Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 17

University of Jyväskylä
Department of Mathematical
Information Technology
Lecture Notes 17

OHJELMOINTI 1
C#

Martti Hyvönen,
Vesa Lappalainen ja
Antti-Jussi Lakanen

Jyväskylä 2012

Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 17

University of Jyväskylä
Department of Mathematical Information Technology

Lecture Notes 17

OHJELMOINTI 1
C#

Martti Hyvönen,
Vesa Lappalainen ja
Antti-Jussi Lakanen

Korjattu painos

Muutokset 1. painokseen näet osoitteessa
https://trac.cc.jyu.fi/projects/ohj1/wiki/monisteenKorjaukset.

Jyväskylä 2012

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

Copyright © 2012 Martti Hyvönen, Vesa Lappalainen, Antti-Jussi Lakanen ja Jyväskylän yliopisto
ISBN 978-951-39-4434-6 (korjattu painos)
ISSN 1456-9787
Jyväskylän yliopistopaino
Jyväskylä 2012

Sisällys

Esipuhe.....1
1. Mitä ohjelmointi on?.....2
2. Ensimmäinen C#-ohjelma.....3
2.1 Ohjelman kirjoittaminen.....3
2.2 Ohjelman kääntäminen ja ajaminen.....3
2.3 Ohjelman rakenne.....5
2.3.1 Virhetyypit.....6
2.3.2 Kääntäjän virheilmoitusten tulkinta.....6
2.3.3 Tyhjät merkit (White spaces).....7
2.4 Kommentointi.....8
2.4.1 Dokumentointi.....8
3. Algoritmit.....10
3.1 Mikä on algoritmi?.....10
3.2 Tarkentaminen.....10
3.3 Yleistäminen.....11
3.4 Harjoitus.....11
3.5 Peräkkäisyys.....11
4. Yksinkertainen graafinen C#-ohjelma.....12
4.1 Mikä on kirjasto?.....12
4.2 Jypeli-kirjasto.....12
4.3 Esimerkki: Lumiukko.....12
4.3.1 Ohjelman suoritus.....14
4.3.2 Ohjelman oleellisemmat kohdat.....14
4.4 Harjoitus.....17
4.5 Kääntäminen ja luokkakirjastoihin viittaaminen.....17
5. Lähdekoodista prosessorille.....18
5.1 Kääntäminen.....18
5.2 Suorittaminen.....18
6. Aliohjelmat.....19
6.1 Aliohjelman kutsuminen.....22
6.2 Aliohjelman kirjoittaminen.....23
6.3 Aliohjelmien dokumentointi.....28
6.3.1 Huomautus.....29
6.4 Aliohjelmat, metodit ja funktiot.....30
7. Muuttujat.....31
7.1 Muuttujan määrittely.....31
7.2 Alkeistietotyypit.....32
7.3 Arvon asettaminen muuttujaan.....33
7.4 Muuttujan nimeäminen.....34
7.4.1 C#:n varatut sanat.....35
7.5 Muuttujien näkyvyys.....35
7.6 Vakiot.....36
7.7 Operaattorit.....36
7.7.1 Aritmeettiset operaatiot.....36
7.7.2 Vertailuoperaattorit.....37
7.7.3 Arvonmuunto-operaattorit.....37
7.7.4 Aritmeettisten operaatioiden suoritusjärjestys.....38
7.8 Huomautuksia.....38
7.8.1 Kokonaisluvun tallentaminen liukulukumuuttujaan.....38
7.8.2 Lisäys- ja vähennysoperaattoreista.....38
7.9 Esimerkki: Painoindeksi.....39
8. Oliotietotyypit.....40
8.1 Mitä oliot ovat?.....40
8.2 Olion luominen.....40
8.3 Oliotietotyyppien ja alkeistietotyyppien ero.....41

8.4 Metodin kutsuminen....43
8.5 Metodin ja aliohjelman ero....43
8.6 Olion tuhoaminen ja roskienkeruu....43
8.7 Olioluokkien dokumentaatio....44
8.7.1 Konstruktorit....44
8.7.2 Harjoitus....46
8.7.3 Metodit....47
8.7.4 Huomautus: Luokkien dokumentaatioiden googlettaminen....47
8.8 Tyyppimuunnokset....47
9. Aliohjelman paluuarvo....49
9.1 Harjoitus....50
10. Visual Studio 2010....51
10.1 Asennus....51
10.2 Käyttö....51
10.2.1 Ensimmäinen käyttökerta....51
10.2.2 Projektit ja solutionit....51
10.2.3 Visual Studion perusnäkymä....53
10.2.4 Ohjelman kirjoittaminen....54
10.2.5 Ohjelman kääntäminen ja ajaminen....54
10.2.6 Referenssien asettaminen....54
10.3 Debuggaus....54
10.4 Hyödyllisiä ominaisuuksia....55
10.4.1 Syntaksivirheiden etsintä....55
10.4.2 Kooditäydennys, IntelliSense....56
10.4.3 Uudelleenmuotoilu....56
10.5 Lisätietoja Visual Studion käytöstä....56
11. ComTest....57
11.1 Käyttö....57
11.2 Liukulukujen testaaminen....58
12. Merkkijonot....60
12.1 Alustaminen....60
12.2 Hyödyllisiä metodeja ja ominaisuuksia....60
12.3 Muokattavat merkkijonot: StringBuilder....62
12.3.1 Muita StringBuilder-luokan hyödyllisiä metodeja....63
12.4 Huomautus: aritmeettinen + vs. merkkijonoja yhdistelevä +....63
12.5 Vinkki: näppärä tyyppimuunnos String-tyypiksi....63
12.6 Reaalilukujen muotoilu String.Format-metodilla....64
13. Ehtolauseet (Valintalauseet)....66
13.1 Mihin ehtolauseita tarvitaan?....66
13.2 if-rakenne: ”Jos aurinko paistaa, mene ulos.”....66
13.3 Vertailuoperaattorit....67
13.3.1 Huomautus: sijoitusoperaattori (=) ja vertailuoperaattori (==)....68
13.4 Esimerkki: yksinkertaisia if-lauseita....68
13.5 if-else -rakenne....68
13.5.1 Esimerkki: Pariton vai parillinen....69
13.6 Loogiset operaatiot....70
13.6.1 De Morganin lait....71
13.6.2 Osittelulaki....71
13.7 else if -rakenne....72
13.7.1 Esimerkki: Tenttiarvosanan laskeminen....73
13.7.2 Harjoitus....73
13.7.3 Harjoitus....74
13.8 switch-rakenne....74
13.8.1 Esimerkki: Arvosana kirjalliseksi....75
14. Olioiden ja alkeistietotyyppien erot....77
15. Taulukot....80
15.1 Taulukon luominen....80
15.2 Taulukon alkioon viittaaminen....81

15.3 Esimerkki: lumiukon pallot taulukkoon....82
15.4 Esimerkki: arvosana kirjalliseksi....83
15.5 Moniulotteiset taulukot....84
15.5.1 Harjoitus....85
15.6 Taulukon kopioiminen....86
15.7 Esimerkki: Moniulotteiset taulukot käytännössä....86
16. Toistorakenteet (silmukat)....88
16.1 ”Syö niin kauan, kuin puuroa on lautasella”....88
16.2 while-silmukka....89
16.2.1 Huomautus: ikuinen silmukka....89
16.2.2 While-silmukka vuokaaviona....89
16.2.3 Esimerkki: Taulukon tulostaminen....90
16.2.4 Esimerkki: Monta palloa....91
16.3 do-while -silmukka....93
16.3.1 Esimerkki: nimen kysyminen käyttäjältä....94
16.4 for-silmukka....94
16.4.1 Huomautus: while ja for -silmukoiden yhtäläisyydet ja erot....97
16.4.2 Esimerkki: lumiukon pallot keltaiseksi....97
16.4.3 Harjoitus....98
16.4.4 Esimerkki: Keskiarvo-aliohjelma....98
16.4.5 Harjoitus....99
16.4.6 Esimerkki: Taulukon kääntäminen käänteiseen järjestykseen....99
16.4.7 Harjoitus....100
16.4.8 Esimerkki: arvosanan laskeminen taulukoilla....100
16.5 foreach-silmukka....104
16.5.1 Esimerkki: taulukon pallot keltaisiksi....104
16.6 Sisäkkäiset silmukat....105
16.7 Esimerkki: rivi, jolla eniten vapaata tilaa....106
16.8 Silmukan suorituksen kontrollointi break- ja continue-lauseilla....106
16.8.1 break....106
16.8.2 continue....107
16.9 Ohjelmointikielistä puuttuva silmukkarakenne....107
16.10 Yhteenveto....108
17. Merkkijonojen pilkkominen ja muokkaaminen....110
17.1 String.Split()....110
17.2 String.Trim()....110
17.3 Esimerkki: Merkkijonon pilkkominen ja muuttaminen kokonaisluvuiksi....110
18. Järjestäminen....114
19. Olion ulkonäön muuttaminen (Jypeli)....115
19.1 Väri....115
19.2 Koko....115
19.3 Tekstuuri....115
19.4 Olion muoto....116
20. Ohjainten lisääminen peliin (Jypeli)....117
20.1 Näppäimistö....118
20.2 Lopetuspainike ja näppäinohjepainike....118
20.3 Peliohjain....119
20.3.1 Analoginen ”tatti”....119
20.4 Hiiri....120
20.4.1 Näppäimet....120
20.4.2 Hiiren liike....120
20.4.3 Hiiren kuunteleminen vain tietyille peliolioille....121
21. Piirtoalusta (Jypeli)....123
21.1 Esimerkki: Punainen rasti....123
21.2 Esimerkki: Pyörivä jana....124
22. Rekursio....125
22.1 Sierpinskin kolmio....126
22.2 Harjoitus....131

23. Dynaamiset tietorakenteet....132
23.1 Rajapinnat....132
23.2 Listat (List<T>)....132
23.2.1 Tietorakenteen määrittäminen....133
23.2.2 Alkioiden lisääminen ja poistaminen....133
24. Poikkeukset....135
24.1 try-catch....135
24.2 finally-lohko....136
24.3 Yleistä....137
25. Tietojen lukeminen ulkoisesta lähteestä....138
25.1 Tekstin lukeminen tiedostosta....138
25.2 Tekstin lukeminen netistä....139
25.3 Satunnaisluvut....140
26. Lukujen esitys tietokoneessa....142
26.1 Lukujärjestelmät....142
26.2 Paikkajärjestelmät....142
26.3 Binääriluvut....142
26.3.1 Binääriluku 10-järjestelmän luvuksi....143
26.3.2 10-järjestelmän luku binääriluvuksi....143
26.4 Negatiiviset binääriluvut....144
26.4.1 Suora tulkinta....145
26.4.2 1-komplementti....145
26.4.3 2-komplementti....145
26.4.4 2-komplementin yhteenlasku....145
26.5 Lukujärjestelmien suhde toisiinsa....146
26.6 Liukuluku (floating-point)....147
26.6.1 Liukuluvun binääriesityksen muuttaminen 10-järjestelmään....148
26.6.2 10-järjestelmän luku liukuluvun binääriesitykseksi....149
26.6.3 Huomio: doublen lukualue....150
26.6.4 Liukulukujen tarkkuus....150
26.6.5 Intelin prosessorikaan ei ole aina osannut laskea liukulukuja oikein....151
27. ASCII-koodi....152
28. Syntaksin kuvaaminen....154
28.1 BNF....154
28.2 Laajennettu BNF (EBNF)....155
29. Jälkisanat....157
Liite: Sanasto....158
Liite: Yleisimmät virheilmoitukset ja niiden syyt....159

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 opiskelun tueksi tarvitaan jokin hyvä aihetta käsittelevä kirja. Useimmat saatavilla olevat kirjat keskittyvät hyvin paljon tiettyyn ohjelmointikieleen—erityisesti aloittelijoille tarkoitetut. Osin tämä on luonnollista, koska ihmisetkin tarvitsevat jonkin yhteisen kielen kommunikoidakseen toisen kanssa. Siksi ohjelmoinnin aloittaminen ilman, että ensin opetellaan jonkun kielen perusteet, on aika haastavaa.

Jäsentämisen selkeyden takia kirjoissa 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. Tästä syystä monisteesta saa yhden näkemyksen mukaisen pintaraapaisun asioille ja kirjoista ja nettilähteistä asiaa on syvennettävä.

Tässä monisteessa käytetään esimerkkikielenä *C#*-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 ennemmin verrattavissa Savon murteen vaihtamisen Turun murteeseen, kuin suomen kielen vaihtamisen ruotsin kieleen. Toisin sanoen, jos yhdellä kielellä on oppinut ohjelmoimaan, kykenee 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, C++, Java ja nyt 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:

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

Tämä moniste on lähes suora kopio Martti Hyvösen ja Vesa Lappalaisen syksyllä 2009 kirjoittamasta *Ohjelmointi 1* -monisteesta muutamin muutoksin. Monisteen asiasisältö on pysynyt pääosin ennallaan, mutta ohjelmointikielenä on nyt ensimmäistä kertaa C#, joka otettiin käyttöön kevään 2010 pilottikurssilla, jossa hyödynnettiin *Jypeli*-ohjelmointikirjastoa. Suurimmat muutokset liittyvät esimerkkeihin ja harjoituksiin, jotka demonstroivat Jypeli-kirjaston käyttöä. Pilottikurssin opettaja ja tämän monisteen muokkaaja oli Antti-Jussi Lakanen.

Moniste on rakentunut nykyiseen muotoonsa monen eri kirjoittajan kautta aina 80-luvulta alkaen. Suurimman panoksen monisteeseen ovat antaneet Timo Männikkö ja Vesa Lappalainen.

Jyväskylässä 2.1.2012

Martti Hyvönen, Vesa Lappalainen, Antti-Jussi Lakanen

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 ohjeet siitä, kuinka kauan ja kuinka suurella teholla sen tulee toimia.

Kaikissa edellisissä esimerkeissä oli siis kyse 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 C#-ohjelma

2.1 Ohjelman kirjoittaminen

C#-ohjelmia (lausutaan *c sharp*) 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 *lähdekoodia* (tai lyhyesti *koodia*) automaattisesti siten, että lukeminen on helpompaa ja siten ymmärtäminen ja muokkaaminen nopeampaa. Ohjelmoijien suosimia ovat mm. *Vim, Emacs* ja *ConTEXT*, 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 C#-ohjelma ja tallennetaan se vaikka nimellä `HelloWorld.cs`. Tiedoston tarkenteeksi (eli niin sanottu tiedostopääte) on sovittu juuri tuo `.cs`, mikä tulee käytetyn ohjelmointikielen nimestä, joten tälläkin kurssilla käytämme sitä. Kannattaa olla tarkkana tiedostoa tallennettaessa, sillä jotkut tekstieditorit yrittävät oletuksena tallentaa kaikki tiedostot tarkenteella `.txt` ja tällöin tiedoston nimi voi helposti tulla muotoon `HelloWorld.cs.txt`.

```
public class HelloWorld
{
   public static void Main(string[] args)
   {
      System.Console.WriteLine("Hello World!");
   }
}
```

Tämän ohjelman pitäisi tulostaa näytölle teksti

```
Hello World!
```

Voidaksemme kokeilla ohjelmaa käytännössä, täytyy se ensiksi *kääntää* tietokoneen ymmärtämään muotoon.

> *Kääntäminen* = 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 C#-sovelluskehitin. Mikäli käytät Windowsia, niin aluksi riittää hyvin Microsoft .NET SDK (Software Development Kit, suom. kehitystyökalut). Muiden käyttöjärjestelmien tapauksessa sovelluskehittimeksi käy esimerkiksi *Novell Mono*. Hyvin monet tämän kurssin harjoituksista on tehtävissä Mono-sovelluskehittimellä, mutta tämän monisteen ohjeet ja esimerkit tullaan käsittelemään Windows-ympäristössä. Edelleen, Jypeli-kirjaston käyttäminen on mahdollista vain Windows-ympäristössä.

Lisätietoa .NET-kehitystyökaluista ja asentamisesta löytyy kurssin kotisivuilta:

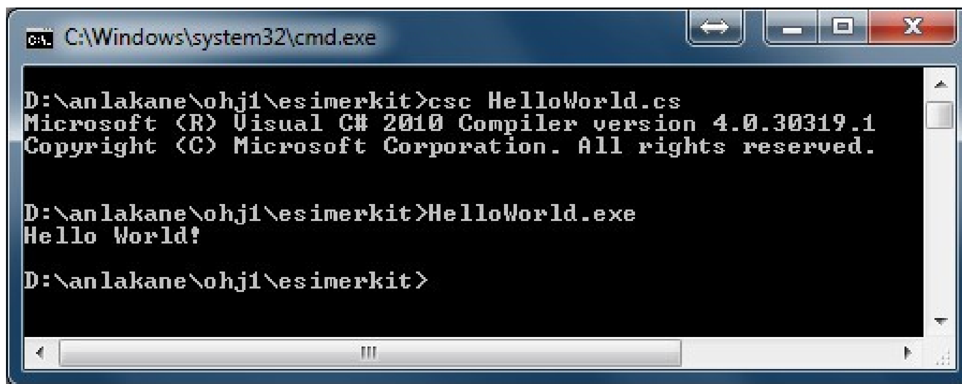
https://trac.cc.jyu.fi/projects/ohj1/wiki/dotnet-tyokalut. Kun sovelluskehitin on asennettu, käynnistetään komentorivi (**Command Prompt**, lyhyemmin **cmd**) ja siirrytään siihen hakemistoon, johon `HelloWorld.cs` tiedosto on tallennettu. Ohjelma käännetään nyt komennolla:

```
csc HelloWorld.cs
```

Komento ”csc” tulee sanoista *C Sharp Compiler* (compiler = kääntäjä). Kääntämisen jälkeen hakemistoon ilmestyy `HelloWorld.exe`-niminen tiedosto, joka voidaan ajaa kuten minkä tahansa ohjelman syöttämällä ohjelman nimi:

```
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 .`cs`-tarkentimen kanssa.

Jos saat virheilmoituksen

```
'csc' is not recognized as an internal or external command, operable program or batch
file.
```

niin kääntäjäohjelmaa `csc.exe` ei silloin löydy niin sanotusta *hakupolusta*. Ohjelman lisääminen hakupolkuun onnistuu komennolla:

```
set PATH=%WINDIR%\Microsoft.NET\Framework\v4.0.30319;%path%
```

Jotta kääntäjää ei tarvitsisi joka kerta lisätä hakupolkuun, voi sen lisätä siihen pysyvästi. Esimerkiksi Windows Vistassa ja 7:ssä tämä tapahtuu seuraavasti.

Klikkaa Oma tietokone -kuvaketta hiiren oikealla painikkeella ja valitse **Ominaisuudet** (Properties). Valitse sitten vasemmalta Advanced system settings ja Advanced-välilehdeltä Environment variables. Ylemmästä laatikosta valitse muuttuja **PATH** ja paina **Muokkaa.** Siirry rivin **Variable value** loppuun, kirjoita puolipiste (;) ja heti perään polku

```
%WINDIR%\Microsoft.NET\Framework\v4.0.30319
```

XP:ssä vastaavasti Oma tietokone → Ominaisuudet → Lisäasetukset → Ympäristömuuttujat.

2.3 Ohjelman rakenne

Ensimmäinen kirjoittamamme ohjelma `HelloWorld.cs` on oikeastaan yksinkertaisin mahdollinen C#-ohjelma.

```
public class HelloWorld
{
```

Yllä olevalla ohjelman ensimmäisellä rivillä määritellään *luokka* (**class**) jonka nimi on `HelloWorld`. Tässä vaiheessa riittää ajatella luokkaa "kotina" *aliohjelmille*. Aliohjelmista puhutaan lisää hieman myöhemmin. Toisaalta luokkaa voidaan verrata "piparkakkumuottiin"–se on rakennusohje olioiden (eli "piparkakkujen") luomista varten. Ohjelman ajamisen aikana olioita syntyy luokkaan kirjoitetun koodin avulla. Olioita voidaan myös tuhota. Yhdellä luokalla voidaan siis tehdä monta samanlaista oliota, aivan kuten yhdellä piparkakkumuotilla voidaan tehdä monta samanlaista (samannäköistä) piparia.

Jokaisessa C#-ohjelmassa on vähintään yksi luokka, mutta luokkia voi olla enemmänkin. Luokan, jonka sisään ohjelma kirjoitetaan, on hyvä olla samanniminen kuin tiedoston nimi. Jos tiedoston nimi on `HelloWorld.cs`, on suositeltavaa, että luokan nimi on myös `HelloWorld`, kuten meidän esimerkissämme. Tässä vaiheessa ei kuitenkaan vielä kannata liikaa vaivata päätänsä sillä, mikä luokka oikeastaan on, se selviää tarkemmin myöhemmin.

> Huomaa! C#:ssa *ei* samasteta isoja ja pieniä kirjaimia. Ole siis tarkkana kirjoittaessasi luokkien nimiä.
>
> Huomaa! Vahva suositus (ja tämän kurssin tapa) on, että luokka alkaa isolla alkukirjaimella, ja ettei skandeja käytetä luokan nimessä.

Luokan edessä oleva `public`-sana on eräs *saantimääre* (eng. *access modifier*). Saantimääreen avulla luokka voidaan asettaa rajoituksetta tai osittain muiden (luokkien) saataville, tai piilottaa kokonaan. Sana `public` tarkoittaa, että luokka on muiden luokkien näkökulmasta *julkinen*, kuten luokat useimmiten ovat. Muita saantimääreitä ovat `protected`, `internal` ja `private`.

Määreen voi myös jättää kirjoittamatta luokan eteen, jolloin luokan määreeksi tulee automaattisesti `internal`. Puhumme aliohjelmista myöhemmin, mutta mainittakoon, että vastaavasti, jos aliohjelmasta jättää määreen kirjoittamatta, tulee siitä `private`. Tällä kurssilla kuitenkin harjoitellaan kirjoittamaan julkisia luokkia (ja aliohjelmia), jolloin `public`-sana kirjoitetaan aina luokan ja aliohjelman eteen.

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

```
public static void Main(string[] args)
{
```

Rivillä kolme määritellään (tai oikeammin *esitellään*) uusi aliohjelma nimeltä `Main`. Nimensä ansiosta se on tämän luokan pääohjelma. Sanat `static` ja `void` kuuluvat aina `Main`-aliohjelman esittelyyn. Paneudumme niihin tarkemmin hieman myöhemmin, mutta sanottakoon tässä kohtaa, että `static` tarkoittaa, että aliohjelma on *luokkakohtainen* (vastakohtana *oliokohtainen*, jolloin

`static`-sanaa ei kirjoiteta). Vastaavasti `void` merkitsee, ettei aliohjelma palauta mitään tietoa.

Samoin kuin luokan, niin myös pääohjelman sisältö kirjoitetaan aaltosulkeiden sisään. C#:ssa ohjelmoijan kirjoittaman koodin suorittaminen alkaa aina käynnistettävän luokan pääohjelmasta. Toki sisäisesti ehtii tapahtua paljon asioita jo ennen tätä.

```
System.Console.WriteLine("Hello World!");
```

Rivillä neljä tulostetaan näytölle *Hello World!*. C#:ssa tämä tapahtuu pyytämällä .NET-ympäristön mukana tulevan luokkakirjaston `System`-luokkakirjaston `Console`-luokkaa tulostamaan `WriteLine()`-*metodilla* (**method**).

> Huomaa! Viitattaessa aliohjelmiin on kirjallisuudessa usein tapana kirjoittaa aliohjelman nimen perään sulut. Kirjoitustyyli korostaa, että kyseessä on aliohjelma, mutta asiayhteydestä riippuen sulut voi myös jättää kirjoittamatta. Tässä monisteessa käytetään pääsääntöisesti jälkimmäistä tapaa, tilanteesta riippuen.

Kirjastoista, olioista ja metodeista puhutaan lisää kohdassa 4.1 ja luvussa 8. Tulostettava merkkijono kirjoitetaan sulkeiden sisälle lainausmerkkeihin (shift + 2). Tämä rivi on myös tämän ohjelman ainoa *lause* (**statement**). Lauseiden voidaan ajatella olevan yksittäisiä toimenpiteitä, joista ohjelma koostuu. Jokainen lause päättyy C#:ssa puolipisteeseen. Koska lauseen loppuminen ilmoitetaan puolipisteellä, ei C#:n *syntaksissa* (**syntax**) ”tyhjillä merkeillä” (**white space**), kuten rivinvaihdoilla ja välilyönneillä ole merkitystä ohjelman toiminnan kannalta. Ohjelmakoodin luettavuuden kannalta niillä on kuitenkin suuri merkitys. Huomaa, että puolipisteen unohtaminen on yksi yleisimmistä ohjelmointivirheistä ja tarkemmin sanottuna *syntaksivirheistä.*

> *Syntaksi* = Tietyn ohjelmointikielen (esim. C#:n) 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 tunnettu esimerkki loogisesta virheestä:

http://money.cnn.com/magazines/fortune/fortune_archive/2000/02/07/272831/index.htm.

2.3.2 Kääntäjän virheilmoitusten tulkinta

Alla on esimerkki syntaksivirheestä `HelloWorld`-ohjelmassa.

```
public class HelloWorld
{
  public static void Main(string[] args)
  {
     System.Console.Writeline("Hello World!");
  }
}
```

Ohjelmassa on pieni kirjoitusvirhe, joka on (ilman apuvälineitä) melko hankala huomata. Tutkitaan `csc`-kääntäjän antamaa virheilmoitusta.

```
HelloWorld.cs(5,17): error CS0117: 'System.Console' does not contain a
        definition for 'Writeline'
```

Kääntäjä kertoo, että tiedostossa `HelloWorld.cs` rivillä 5 ja sarakkeessa 17 on seuraava virhe: `System.Console`-luokka ei tunne `Writeline`-komentoa. Tämä onkin aivan totta, sillä `WriteLine` kirjoitetaan isolla L:llä. Korjattuamme tuon ohjelma toimii jälleen.

Valitettavasti virheilmoituksen sisältö ei aina kuvaa ongelmaa kovinkaan hyvin. Alla olevassa esimerkissä on erehdytty laittamaan puolipisteen väärään paikkaan.

```
public class HelloWorld
{
  public static void Main(string[] args);
  {
     System.Console.Writeline("Hello World!");
  }
}
```

Virheilmoitus, tai oikeammin virheilmoitukset, näyttävät seuraavalta.

```
HelloWorld.cs(4,3): error CS1519: Invalid token '{' in class, struct, or
        interface member declaration
HelloWorld.cs(5,26): error CS1519: Invalid token '(' in class, struct, or
        interface member declaration
HelloWorld.cs(7,1): error CS1022: Type or namespace definition, or end-of-file
        expected
```

Ensimmäinen virheilmoitus osoittaa riville 4, vaikka todellisuudessa ongelma on rivillä 3. Toisin sanoen, näistä virheilmoituksista ei ole meille tässä tilanteessa lainkaan apua, päinvastoin, ne kehottavat tekemään jotain, mitä emme halua.

2.3.3 Tyhjät merkit (White spaces)

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

```
public class HelloWorld
                          {


     public static void Main(string[] args)
  {
System.Console.WriteLine("Hello World!");
    }


}
```

Edelleen, koodi voitaisiin kirjoittaa myös seuraavasti.

```
public class HelloWorld { public static void Main(string[] args) {
    System.Console.WriteLine("Hello World!"); } }
```

Vaikka molemmat yllä olevista esimerkeistä ovat syntaksiltaan oikein, eli ne noudattavat C#:n kielioppisääntöjä, on niiden luettavuus huomattavasti heikompi kuin alkuperäisen ohjelmamme. C#:ssa on yhteisesti sovittuja 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. Linkkejä koodauskäytänteisiin löytyy kurssin wiki-sivulta osoitteesta

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

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

```
System.Console.WriteLine("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.Console.WriteLine("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 koodin kirjoittajaa itseään varten että tulevia ohjelman lukijoita ja ylläpitäjiä 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. C#:ssa on kolmenlaisia kommentteja.

```
// 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 (`*/`). Huomaa, ettei asteriskin ja vinoviivan väliin tule välilyöntiä.

2.4.1 Dokumentointi

Kolmas kommenttityyppi on *dokumentaatiokommentti*. Dokumentaatiokommenteissa on tietty syntaksi, ja tätä noudattamalla voidaan dokumentaatiokommentit muuttaa sellaiseen muotoon, että kommentteihin perustuvaa yhteenvetoa on mahdollista tarkastella nettiselaimen avulla.

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

Dokumentaatiokommentit kirjoitetaan siten, että rivin alussa on aina aina kolme vinoviivaa (shift + 7). Jokainen seuraava dokumentaatiokommenttirivi aloitetaan siis myöskin kolmella vinoviivalla.

```
/// Tämä
/// on
/// dokumentaatiokommentti
```

Dokumentoiminen tapahtuu *tagien* avulla. Jos olet joskus kirjoittanut HTML-sivuja, on merkintätapa sinulle tuttu. Dokumentaatiokommentit alkavat aloitustagilla, muotoa `<esimerkki>`, jonka perään tulee kommentin asiasisältö. Kommentti loppuu lopetustagiin, muotoa `</esimerkki>`, siis muuten sama kuin aloitustagi, mutta ensimmäisen kulmasulun jälkeen on yksi vinoviiva.

C#-tageja ovat esimerkiksi `<summary>`, jolla ilmoitetaan pieni yhteenveto kommenttia seuraavasta koodilohkosta (esimerkiksi pääohjelma tai metodi). Yhteenveto päättyy `</summary>` -lopetustagiin.

Ohjelman kääntämisen yhteydessä dokumentaatiotagit voidaan kirjoittaa erilliseen *XML*-tiedostoon, josta ne voidaan edelleen muuntaa helposti selattaviksi HTML-sivuiksi. Tageja voi keksiä itsekin lisää, mutta tämän kurssin tarpeisiin riittää hyvin suositeltujen tagien luettelo. Tiedot suositelluista tageista löytyvät C#:n dokumentaatiosta:

http://msdn.microsoft.com/en-us/library/5ast78ax.aspx.

Voisimme kirjoittaa nyt C#-kommentit `HelloWorld`-ohjelman alkuun seuraavasti:

```
/// @author  Antti-Jussi Lakanen
/// @version 22.12.2011

/// <summary>
/// Esimerkkiohjelma, joka tulostaa tekstin "Hello World!"
/// </summary>

public class HelloWorld
{
  /// <summary>
  /// Pääohjelma, joka hoitaa varsinaisen tulostamisen.
  /// </summary>
  /// <param name="args">Ei käytössä</param>
  public static void Main(string[] args)
  { // Suoritus alkaa siis tästä
    System.Console.WriteLine("Hello World!"); // Tämä lause tulostaa ruudulle
  } // Ohjelman suoritus päättyy tähän
}
```

Ohjelman alussa kerrotaan kohteen tekijän nimi. Tämän jälkeen tulee ensimmäinen dokumentaatiokommentti (huomaa kolme vinoviivaa), joka on lyhyt ja ytimekäs kuvaus tästä luokasta. Huomaa, että jossain dokumentaation tiivistelmissä näytetään vain tuo ensimmäinen virke. [DOC] [HYV]

Dokumentointi on erittäin keskeinen osa ohjelmistotyötä. Luokkien ja koodirivien määrän kasvaessa dokumentointi helpottaa niin omaa työskentelyä kuin tulevien käyttäjien ja ylläpitäjien tehtävää. Dokumentoinnin tärkeys näkyy muun muassa siinä, että jopa 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:

```
Kahvin keittäminen:

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. Esimerkiksi pannukahvinkeitossa 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 kahvinporot:
  3.1.  Mittaa kahvinporot.
  3.2.  Sekoita kahvinporot kiehuvaan veteen.
4.  Anna tasaantua:
  4.1.  Odota, kunnes suurin osa valmiista kahvista 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.

3.4 Harjoitus

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?

3.5 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 C#-ohjelmointikielellä seuraavassa luvussa.

4. Yksinkertainen graafinen C#-ohjelma

Seuraavissa esimerkeissä käytetään Jyväskylän yliopistossa kehitettyä *Jypeli-ohjelmointikirjastoa.* Kirjaston voit ladata koneelle osoitteesta

https://trac.cc.jyu.fi/projects/npo/wiki/LataaJypeli,

josta löytyy myös ohjeet kirjaston asennukseen ja käyttöön. Huomaa, että tietokoneellasi tulee olla asennettuna .NET Framework 4 sekä XNA Game Studio 4, jotta graafinen ohjelma voidaan kääntää. .NET-frameworkin asennusohjeet löytyvät osoitteesta

https://trac.cc.jyu.fi/projects/ohj1/wiki/dotnet-tyokalut.

4.1 Mikä on kirjasto?

C#-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.

C#-ohjelmoijan kannalta oleellisin kirjasto on `.NET Framework` luokkakirjasto. Luokkakirjaston *dokumentaatioon* (**documentation**) kannattaa tutustua, sillä sieltä löytyy monia todella hyödyllisiä metodeja. Dokumentaatio löytyy Microsoftin sivuilta osoitteesta

http://msdn.microsoft.com/en-us/library/ms229335.aspx.

> *Dokumentaatio* = Sisältää tiedot kaikista kirjaston luokista ja niiden metodeista (ja aliohjelmista). Löytyy useimmiten ainakin WWW-muodossa.

4.2 Jypeli-kirjasto

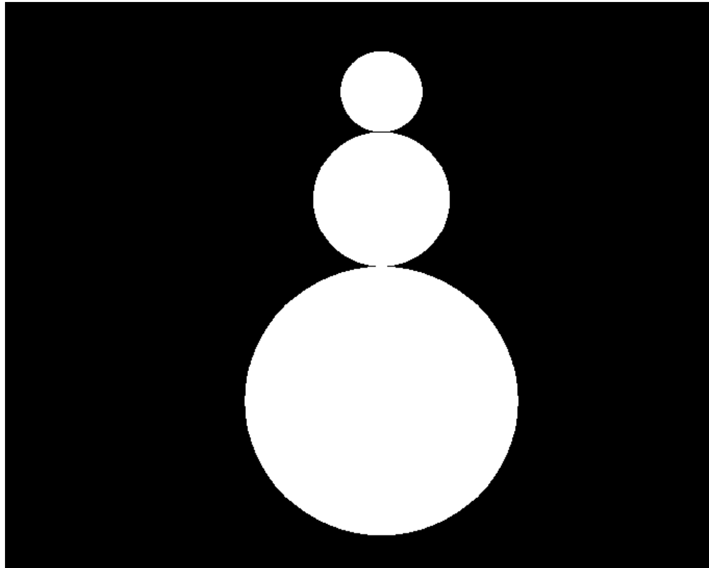
Jypeli-kirjaston kehittäminen aloitettiin Jyväskylän yliopistossa keväällä 2009. Tämän monisteen esimerkeissä käytetään versiota 4. Jypeli-kirjastoon on kirjoitettu valmiita luokkia ja metodeja siten, että esimerkiksi fysiikan ja matematiikan ilmiöiden, sekä pelihahmojen ja liikkeiden ohjelmointi lopulliseen ohjelmaan on helpompaa.

4.3 Esimerkki: Lumiukko

Piirretään lumiukko käyttämällä Jypeli-kirjastoa. Mallikoodin vasemmassa reunassa juoksee myös rivinumerointi, joka ei kuulu koodiin, mutta helpottaa lukemista.

```
01 /// @author  Antti-Jussi Lakanen, Martti Hyvönen
02 /// @version 22.12.2011
03
04
05 // Otetaan käyttöön Jyväskylän yliopiston Jypeli-kirjasto
06 using Jypeli;
07
08 /// <summary>
09 /// Luokka, jossa harjoitellaan piirtämistä lisäämällä ympyröitä ruudulle
10 /// </summary>
11 public class Lumiukko : PhysicsGame
12 {
13   /// <summary>
14   /// Pääohjelmassa laitetaan "peli" käyntiin Jypelille tyypilliseen tapaan
15   /// </summary>
16   /// <param name="args">Ei käytössä</param>
17   public static void Main(string[] args)
18   {
19     using (Lumiukko peli = new Lumiukko())
20     {
21       peli.Run();
22     }
23   }
24
25   /// <summary>
26   /// Piirretään oliot ja zoomataan kamera niin että kenttä näkyy kokonaan.
27   /// </summary>
28   public override void Begin()
29   {
30     Camera.ZoomToLevel();
31     Level.BackgroundColor = Color.Black;
32
33     PhysicsObject p1 = new PhysicsObject(2*100.0, 2*100.0, Shape.Circle);
34     p1.Y = Level.Bottom + 200.0;
35     Add(p1);
36
37     PhysicsObject p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
38     p2.Y = p1.Y + 100 + 50;
39     Add(p2);
40
41     PhysicsObject p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
42     p3.Y = p2.Y + 50 + 30;
43     Add(p3);
44
45   }
46 }
```

Ajettaessa ohjelman tulisi piirtää yksinkertainen lumiukko keskelle ruutua, kuten alla olevassa kuvassa.



Kuva 2: Lumiukko Jypeli-kirjaston avulla piirrettynä

4.3.1 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.

4.3.2 Ohjelman oleellisemmat kohdat

Tarkastellaan ohjelman oleellisempia kohtia.

```
06 using Jypeli;
```

Aluksi meidän täytyy kertoa kääntäjälle, että haluamme ottaa käyttöön koko Jypeli-kirjaston. Nyt Jypeli-kirjaston kaikki luokat (ja niiden metodit) ovat käytettävissämme.

```
08 /// <summary>
09 /// Luokka, jossa harjoitellaan piirtämistä lisäämällä ympyröitä ruudulle
10 /// </summary>
11 public class Lumiukko : PhysicsGame
12 {
```

Rivit 8-10 ovat dokumentaatiokommentteja. Rivillä 11 luodaan `Lumiukko`-luokka, joka hieman poikkeaa `HelloWorld`-esimerkin tavasta luoda uusi luokka. Tässä kohtaa käytämme ensimmäisen kerran Jypeli-kirjastoa, ja koodissa kerrommekin, että `Lumiukko`-luokka, jota juuri olemme tekemässä, ”perustuu” Jypeli-kirjastossa olevaan `PhysicsGame`-luokkaan. Täsmällisemmin sanottuna `Lumiukko`-luokka peritään `PhysicsGame`-luokasta. Tuon `PhysicsGame`-luokan avulla objektien piirtäminen, myöhemmin liikuttelu ruudulla ja fysiikan lakien hyödyntäminen on vaivatonta.

```
13   /// <summary>
14   /// Pääohjelmassa laitetaan "peli" käyntiin Jypelille tyypilliseen tapaan.
15   /// </summary>
16   /// <param name="args">Ei käytössä</param>
17   public static void Main(String[] args)
18   {
19     using (Lumiukko peli = new Lumiukko())
20     {
21       peli.Run();
22     }
23   }
```

Myös `Main`-metodi, eli pääohjelma, on Jypeli-peleissä käytännössä aina tällainen vakiomuotoinen, joten jatkossa siihen ei tarvi juurikaan koskea. Ohitamme tässä vaiheessa pääohjelman sisällön mainitsemalla vain, että pääohjelmassa `Lumiukko`-luokasta tehdään uusi olio (eli uusi ”peli”), joka sitten laitetaan käyntiin `peli.Run()`-kohdassa. Jypeli-kirjaston rakenteesta johtuen kaikki varsinainen peliin liittyvä koodi kirjoitetaan omaan aliohjelmaansa, `Begin`-aliohjelmaan, jota käsittelemme seuraavaksi.

Tarkasti ottaen `Begin` alkaa riviltä 29. Ensimmäinen lause on kirjoitettu riville 30.

```
30     Camera.ZoomToLevel();
31     Level.BackgroundColor = Color.Black;
```

Näistä kahdesta rivistä ensimmäisellä kutsutaan `Camera`-luokan `ZoomToLevel`-aliohjelmaa, joka pitää huolen siitä, että ”kamera” on kohdistettuna ja zoomattuna oikeaan kohtaan. Aliohjelma ei ota vastaan parametreja, joten sulkujen sisältö jää tyhjäksi. Toisella rivillä muutetaan taustan väri.

```
33     PhysicsObject p1 = new PhysicsObject(2*100.0, 2*100.0, Shape.Circle);
34     p1.Y = Level.Bottom + 200.0;
35     Add(p1);
```

Näiden kolmen rivin aikana luomme uuden fysiikkaolio-ympyrän, annamme sille säteen, y-koordinaatin, sekä lisäämme sen ”pelikentälle”, eli näkyvälle alueelle valmiissa ohjelmassa.

Tarkemmin sanottuna luomme uuden `PhysicsObject`-*olion* eli `PhysicsObject`-luokan *ilmentymän*, jonka nimeksi annamme `p1`. `PhysicsObject`-oliot ovat pelialueella liikkuvia olioita, jotka noudattavat fysiikan lakeja. Sulkujen sisään laitamme tiedon siitä, millaisen objektin haluamme luoda – tässä tapauksessa leveys ja korkeus (Jypeli-mitoissa, ei pikseleissä), sekä olion muoto. Teemme siis ympyrän, jonka säde on 100 (leveys 2 * 100 ja korkeus 2 * 100). Muita `Shape`-kokoelmasta löytyviä muotoja ovat muiden muassa kolmio, neliö, sydän jne. Olioista puhutaan lisää luvussa 8.

Kokeile itse muuttaa olion muotoa!

Seuraavalla rivillä asetetaan olion paikka Y-arvon avulla. Huomaa että Y kirjoitetaan isolla kirjaimella. Tämä on p1-olion ominaisuus, attribuutti. X-koordinaattia meidän ei tarvitse tässä erikseen asettaa, se on oletusarvoisesti 0 ja se kelpaa meille. Saadaksemme ympyrät piirrettyä oikeille paikoilleen, täytyy meidän laskea koordinaattien paikat. Oletuksena ikkunan keskipiste on koordinaatiston origo eli piste (0, 0). x-koordinaatin arvot kasvavat oikealle ja y:n arvot ylöspäin, samoin kuin ”normaalissa” koulusta tutussa koordinaatistossa.

Peliolio täytyy aina lisätä kentälle, ennen kuin se saadaan näkyviin. Tämä tapahtuu `Add`-metodin avulla, joka ottaa parametrina kentälle lisättävän olion nimen (tässä `p1`).

Metodeille annettavia tietoja sanotaan *parametreiksi* (**parameter**). `ZoomToLevel`-metodi ei ota vastaan yhtään parametria, mutta `Add`-metodi sen sijaan ottaa yhden parametrin: `PhysicsObject`-tyyppisen olion. `Add`-metodille voidaan antaa toinenkin parametri: *taso*, jolle olio lisätään. Tasojen

avulla voidaan hallita, mitkä oliot lisätään päällimmäiseksi. Tasoparametri voidaan kuitenkin jättää antamatta, jolloin ohjelma itse päättää tasojen parhaan järjestyksen. Parametrit kirjoitetaan metodin nimen perään sulkeisiin ja ne erotetaan toisistaan pilkuilla.

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

Seuraavien rivien aikana luomme vielä kaksi ympyrää vastaavalla tavalla, mutta vaihtaen sädettä ja ympyrän koordinaatteja.

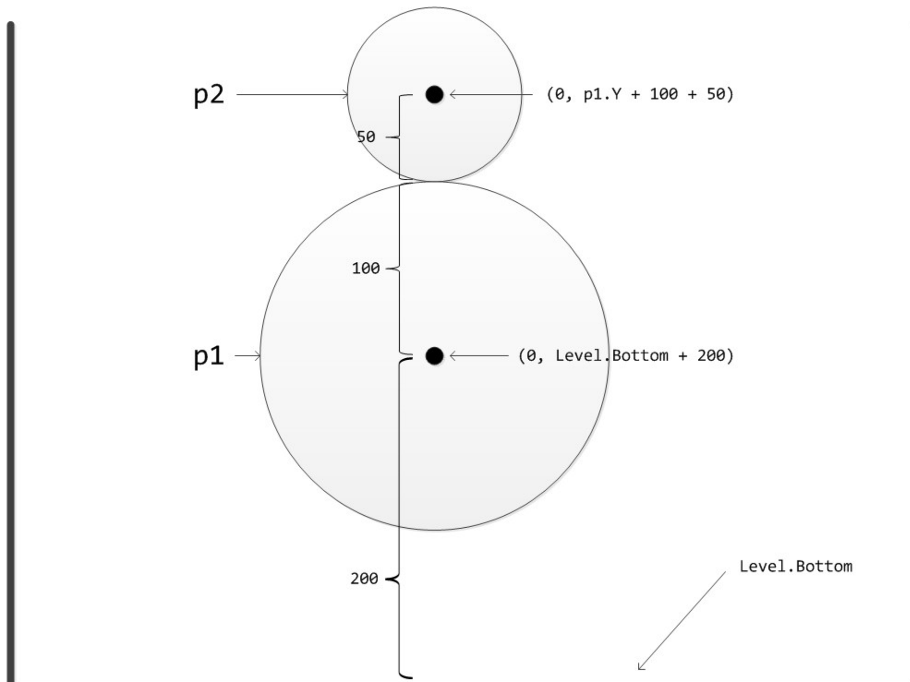
Esimerkissä koordinaattien laskemiseen on käytetty C#:n *aritmeettisia operaatioita*. Voisimme tietenkin laskea koordinaattien pisteet myös itse, mutta miksi tehdä niin jos tietokone voi laskea pisteet puolestamme? C#:n aritmeettiset perusoperaatiot ovat summa (+), vähennys (-), kerto (*), jako (/) ja jakojäännös (%). Aritmeettisista operaatioista puhutaan lisää muuttujien yhteydessä kohdassa 7.7.1.

Keskimmäinen ympyrä tulee alimman ympyrän yläpuolelle niin, että ympyrät sivuavat toisiaan. Keskimmäisen ympyrän keskipiste sijoittuu siis siten, että sen x-koordinaatti on 0 ja y-koordinaatti on *alimman ympyrän paikka* + *alimman ympyrän säde* + *keskimmäisen ympyrän säde*. Kun haluamme, että keskimmäisen ympyrän säde on 50, niin silloin keskimmäisen ympyrän keskipiste tulee kohtaan (0, p1.Y + 100 + 50) ja se piirretään lauseella:

```
PhysicsObject p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
p2.Y = p1.Y + 100 + 50;
Add(p2);
```

Huomaa, että fysiikkaolion `Y`-ominaisuuden asettamisen (*set*) lisäksi voimme myös lukea tai pyytää (*get*) kyseisen ominaisuuden arvon. Yllä teemme sen kirjoittamalla yksinkertaisesti sijoitusoperaattorin oikealle puolelle `p1.Y`.

Seuraava kuva havainnollistaa ensimmäisen ja toisen pallon asettelua.



Kuva 3: Lumiukon kaksi ensimmäistä palloa asemoituina paikoilleen.

Ylin ympyrä sivuaa sitten taas keskimmäistä ympyrää. Harjoitustehtäväksi jätetään laskea ylimmän ympyrän koordinaatit, kun ympyrän säde on 30.

Kaikki tiedot luokista, luokkien metodeista sekä siitä mitä parametreja metodeille tulee antaa löydät käyttämäsi kirjaston dokumentaatiosta. Jypelin luokkadokumentaatio löytyy osoitteesta: `http://kurssit.it.jyu.fi/npo/material/latest/documentation/html/`.

4.4 Harjoitus

Etsi Jypeli-kirjaston dokumentaatiosta `RandomGen`-luokka. Mitä tietoa löydät `NextInt(int min, int max)`-metodista? Mitä muita metodeja luokassa on?

4.5 Kääntäminen ja luokkakirjastoihin viittaaminen

Jotta `Lumiukko`-esimerkkiohjelma voitaisiin nyt kääntää C#-kääntäjällä, tulee Jypeli-kirjasto olla tallennettuna tietokoneelle. Jypeli käyttää XNA-kirjaston lisäksi vapaan lähdekoodin fysiikka- ja matematiikkakirjastoja. Fysiikka- ja matematiikkakirjastot on sisäänrakennettuina Jypeli-kirjastoon.

Ennen kääntämistä kopioi seuraavat tiedostot kurssin kotisivuilta (`https://trac.cc.jyu.fi/projects/ohj1/wiki/csharpCommandLine`) samaan kansioon `Lumiukko.cs`-tiedoston kanssa.

- `Jypeli4.dll`

Meidän täytyy vielä välittää kääntäjälle tieto siitä, että Jypeli-kirjastoa tarvitaan `Lumiukko`-koodin kääntämiseen. Lisäksi annetaan kääntäjälle tieto siitä, että ohjelma tehdään *32-bittisille* järjestelmille (x86). Tämä tehdään `csc`-ohjelman `/reference`-parametrin avulla. Lisäksi tarvitaan referenssi Jypelin käyttämään XNA-kirjastoon. Kirjoita nyt komentoriville

```
csc Lumiukko.cs
/reference:Jypeli4.dll;"%XNAGSv4%\References\Windows\x86\Microsoft.Xna.Framework.Game.dll"
/platform:x86
```

Jos käyttöjärjestelmäsi ei tunnista `csc`-komentoa, niin kertaa luvussa 2 olevat ohjeet komennon asettamisesta `PATH`-ympäristömuuttujan poluksi.

Vinkki! Yllä esitelty kääntämiskomento on varsin pitkä. Asioiden helpottamiseksi voit kirjoittaa tiedoston `csk.bat`, joka sisältää seuraavan tekstin (komento on yksi pitkä rivi):

```
@"%WINDIR%\Microsoft.NET\Framework\v4.0.30319\csc" %*
/reference:Jypeli4.dll;"%XNAGSv4%References\Windows\x86\Microsoft.Xna.Framework.Game.dll";"%XNAGSv4%Re
ferences\Windows\x86\Microsoft.Xna.Framework.dll" /platform:x86  /define:WINDOWS
```

Tämä asettaa puolestasi reference ja platform -parametrit. Varmista, että tekemäsi `csk.bat`-tiedosto on ”polussa”. Tämän jälkeen kääntäminen onnistuu yksinkertaisemmin: `csk OhjelmanNimi.cs`

5. Lähdekoodista prosessorille

5.1 Kääntäminen

Tarkastellaan nyt tarkemmin sitä kuinka C#-lähdekoodi muuttuu lopulta prosessorin ymmärtämään muotoon. Kun ohjelmoija luo ohjelman lähdekoodin, joka käyttää *.NET Framework* -ympäristöä, tapahtuu kääntäminen sisäisesti kahdessa vaiheessa. Ohjelma käännetään ensin eräänlaiselle välikielelle, *MSIL*:lle (**Microsoft Intermediate Language**), joka ei ole vielä suoritettavissa millään käyttöjärjestelmällä. Tästä välivaiheen koodista käännetään ajon aikana valmis ohjelma halutulle käyttöjärjestelmälle, kuten Mac Os X:lle tai Linuxille, niin sanotulla *JIT-kääntäjällä* (**Just-In-Time**). JIT-kääntäjä muuntaa välivaiheen koodin juuri halutulle käyttöjärjestelmälle sopivaksi koodiksi nimenomaan ohjelmaa ajettaessa – tästä tulee nimi ”just-in-time”.

Ennen ensimmäistä kääntämistä kääntäjä tarkastaa, että koodi on syntaksiltaan oikein. [VES][KOS]

Kääntäminen tehtiin Windowsissa komentorivillä (**Command Prompt**) käyttämällä komentoa

```
csc Tiedostonnimi.cs
```

tai hyödyntämällä edellisessä luvussa esiteltyä komentojonoa

```
csk Tiedostonnimi.cs
```

5.2 Suorittaminen

C#:n tuottaa siis lähdekoodista suoritettavan (tai ”ajettavan”) tiedoston. Tämä tiedosto on käyttöjärjestelmäriippuvainen, ja suoritettavissa vain sillä alustalla, johon käännös on tehty. Toisin sanoen, Windows-ympäristössä käännetyt ohjelmat eivät ole ajettavissa OS X -käyttöjärjestelmässä, ja toisin päin.

Toisin kuin C#, eräät toiset ohjelmointikielet tuottavat käyttöjärjestelmäriippumatonta koodia. Esimerkiksi *Java*-kielessä kääntäjän tuottama tiedosto on niin sanottua *tavukoodia*, joka on käyttöjärjestelmäriippumatonta koodia. Tavukoodin suorittamiseen tarvitaan Java-virtuaalikone (**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. [VES][KOS]

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.

```
01 using Jypeli;
02
03 /// <summary>
04 /// Piirretään lumiukko.
05 /// </summary>
06 public class Lumiukko : PhysicsGame
07 {
08   /// <summary>
09   /// Pääohjelmassa peli käyntiin.
10   /// </summary>
11   /// <param name="args">Ei käytössä.</param>
12   public static void Main(String[] args)
13   {
14     using (Lumiukko game = new Lumiukko())
15     {
16       game.Run();
17     }
18   }
19
20   /// <summary>
21   /// Aliohjelma, jossa
22   /// piirretään ympyrät.
23   /// </summary>
24   public override void Begin()
25   {
26     Camera.ZoomToLevel();
27     Level.BackgroundColor = Color.Black;
28
29     PhysicsObject p1, p2, p3;
30
31     // Eka ukko
32     p1 = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
33     p1.Y = Level.Bottom + 200.0;
34     Add(p1);
35
36     p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
37     p2.Y = p1.Y + 100 + 50;
38     Add(p2);
39
40     p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
41     p3.Y = p2.Y + 50 + 30;
42     Add(p3);
43
44     // Toinen ukko
45     p1 = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
46     p1.X = 200;
47     p1.Y = Level.Bottom + 300.0;
48     Add(p1);
49
50     p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
51     p2.X = 200;
52     p2.Y = p1.Y + 100 + 50;
53     Add(p2);
54
```

```
55     p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
56     p3.X = 200;
57     p3.Y = p2.Y + 50 + 30;
58     Add(p3);
59   }
60 }
```

Huomataan, että ensimmäisen ja toisen lumiukon piirtäminen tapahtuu lähes samanlaisilla koodinpätkillä. Itse asiassa ainoa ero on, että jälkimmäisen lumiukon pallot saavat ensimmäisestä lumiukosta eroavat koordinaatit. 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. Määritellään heti alussa myös `p1`, `p2` ja `p3` `PhysicsObject`-olioiksi. Rivinumerointi on tässä jätetty pois selvyyden vuoksi. Luvun lopussa korjattu ohjelma esitellään kokonaisuudessaan rivinumeroinnin kanssa.

```
double x, y;
PhysicsObject p1, p2, p3;

// Tehdään ensimmäinen lumiukko
x = 0; y = Level.Bottom + 200.0;
p1 = new PhysicsObject(2*100.0, 2*100.0, Shape.Circle);
p1.X = x;
p1.Y = y;
Add(p1);

p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
p2.X = x;
p2.Y = y + 100 + 50; // y + 1. pallon säde + 2. pallon säde
Add(p2);

p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
p3.X = x;
p3.Y = y + 100 + 2 * 50 + 30; // y + 1. pallon säde + 2. halk. + 3. säde
Add(p3);
```

Vastaavasti toiselle lumiukolle: asetetaan vain x:n ja y:n arvot oikeiksi.

```
// Tehdään toinen lumiukko
x = 200; y = Level.Bottom + 300.0;
p1 = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
p1.X = x;
p1.Y = y;
Add(p1);

p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
p2.X = x;
p2.Y = y + 100 + 50;
Add(p2);

p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
p3.X = x;
p3.Y = y + 100 + 2*50 + 30;
Add(p3);
```

Tarkastellaan nyt muutoksia hieman tarkemmin.

```
double x, y;
```

Yllä olevalla rivillä esitellään kaksi *liukulukutyyppistä muuttujaa*. Liukuluku on eräs tapa esittää *reaalilukuja* tietokoneissa. C#:ssa jokaisella muuttujalla on oltava tyyppi ja eräs liukulukutyyppi C#:ssa on `double`. Muuttujista ja niiden tyypeistä puhutaan lisää luvussa 7.

Liukuluku (**floating point**) = Tietokoneissa käytettävä esitysmuoto reaaliluvuille. Tarkempaa tietoa liukuluvuista löytyy luvusta 26.

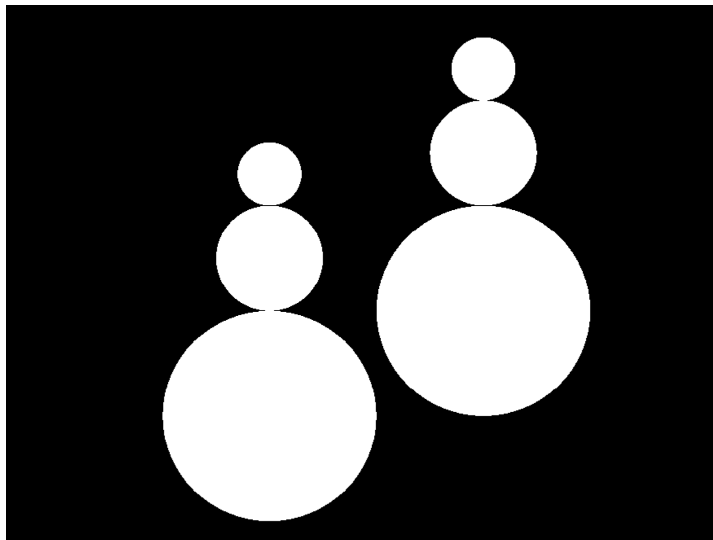
```
x = 0; y = Level.Bottom + 200.0;
```

Yllä olevalla rivillä on kaksi lausetta. Ensimmäisellä asetetaan muuttujaan `x` arvo `0` ja toisella muuttujaan `y` arvo `50`. Nyt voimme käyttää lumiukon pallojen laskentaan näitä muuttujia.

```
x = 300; y = Level.Bottom + 300.0;
```

Vastaavasti yllä olevalla rivillä asetetaan nyt muuttujiin uudet arvot, joita käytetään seuraavan lumiukon pallojen paikkojen laskemiseen. Huomaa, että y-koordinaatti saa negatiivisen arvon, jolloin lumiukon alimman pallon keskipiste painuu kuvaruudun keskitason alapuolelle.

Nyt alimman pallon x-koordinaatiksi sijoitetaankin *muuttuja* `x`, ja vastaavasti y-koordinaatin arvoksi asetetaan *muuttuja* `y`, ja muiden pallojen sijainnit lasketaan ensimmäisen pallon koordinaattien perusteella.



Kuva 4: Kaksi lumiukkoa.

Näiden muutosten jälkeen molempien lumiukkojen varsinainen piirtäminen tapahtuu nyt **täysin samalla koodilla**.

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, jos koodia kirjoittaessa joutuu tekemään suoraa kopiointia, 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 kohdistuvatko muutokset kohdistuvat vain yhteen paikkaan (hyvä) vai joudutaanko muutoksia tekemään useaan paikkaan (huono).

6.1 Aliohjelman kutsuminen

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

Sovitaan, että aliohjelmamme piirtää aina samankokoisen lumiukon haluamaamme pisteeseen. Mitkä ovat ne välttämättömät tiedot, jotka aliohjelma tarvitsee piirtääkseen lumiukon?

Aliohjelma tarvitsee tiedon *mihin* pisteeseen lumiukko piirretään. Viedään siis parametrina lumiukon alimman pallon keskipiste. Muiden pallojen paikat voidaan laskea tämän pisteen avulla. Lisäksi tarvitaan yksi `Game`-tyyppinen parametri, jotta aliohjelmaamme voisi kutsua myös toisesta ohjelmasta. Nämä parametrit riittävät lumiukon piirtämiseen.

Kun aliohjelmaa käytetään ohjelmassa, sanotaan, että aliohjelmaa *kutsutaan*. Kutsu tapahtuu kirjoittamalla aliohjelman nimi ja antamalla sille parametrit. Aliohjelmakutsun erottaa metodikutsusta vain se, että metodi liittyy aina tiettyyn olioon. Esimerkiksi `pallo`-olio `p1` voitaisiin poistaa pelikentällä kutsumalla metodia `Destroy()`, eli kirjoittaisimme:

```
p1.Destroy();
```

Toisin sanoen metodeja kutsuttaessa täytyy ensin kirjoittaa sen olion nimi, jonka metodia kutsutaan, ja sen jälkeen pisteellä erotettuna kirjoittaa haluttu metodin nimi. Sulkujen sisään tulee luonnollisesti tarvittavat parametrit. Yllä olevan esimerkin `Destroy`-metodi ei ota vastaan yhtään parametria.

Päätetään, että aliohjelman nimi on `PiirraLumiukko`. Sovitaan myös, että aliohjelman ensimmäinen parametri on peli johon lumiukko ilmestyy (kirjoitetaan `this`), toinen parametri on lumiukon alimman pallon keskipisteen x-koordinaatti ja kolmas parametri lumiukon alimman pallon keskipisteen y-koordinaatti. Tällöin kentälle voitaisiin piirtää lumiukko, jonka alimman pallon keskipiste on (0, Level.Bottom + 200.0), seuraavalla kutsulla:

```
PiirraLumiukko(this, 0, Level.Bottom + 200.0);
```

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

```
Lumiukot.PiirraLumiukko(this, 0, Level.Bottom + 200.0);
```

Vaikka tämä muoto muistuttaa jo melko paljon metodin kutsua on ero kuitenkin selvä. Metodia kutsuttaessa toimenpide tehdään aina *tietylle oliolle*, kuten `p1.Destroy()` tuhoaa juuri sen pallon, johon `p1`-olio viittaa. Pallojahan voi tietenkin olla myös muita erinimisiä (kuten esimerkissämme onkin). Alla olevassa aliohjelmakutsussa kuitenkin käytetään vain luokasta `Lumiukot` löytyvää `PiirraLumiukko`-aliohjelmaa.

Jos olisimme toteuttaneet jo varsinaisen aliohjelman, piirtäisi `Begin` meille nyt kaksi lumiukkoa.

```
/// <summary>
```

```
/// Kutsutaan PiirraLumiukko-aliohjelmaa
/// sopivilla parametreilla.
/// </summary>
public override void Begin()
{
  Camera.ZoomToLevel();
  Level.BackgroundColor = Color.Black;

  PiirraLumiukko(this, 0, Level.Bottom + 200.0);
  PiirraLumiukko(this, 200.0, Level.Bottom + 300.0);
}
```

Koska `PiirraLumiukko`-aliohjelmaa ei luonnollisesti vielä ole olemassa, ei ohjelmamme vielä toimi. Seuraavaksi meidän täytyy toteuttaa itse aliohjelma, jotta kutsut alkavat toimimaan.

Usein ohjelman toteutuksessa on viisasta edetä juuri tässä järjestyksessä: suunnitellaan aliohjelmakutsu ensiksi, kirjoitetaan kutsu sille kuuluvalle paikalle, ja vasta sitten toteutetaan varsinainen aliohjelman kirjoittaminen.

6.2 Aliohjelman kirjoittaminen

Ennen varsinaista aliohjelman toiminnallisuuden kirjoittamista täytyy aliohjelmalle tehdä määrittely (kutsutaan myös esittelyksi, **declaration**). Kirjoitetaan määrittely aliohjelmalle, jonka kutsun jo teimme edellisessä alaluvussa.

Lisätään ohjelmaamme aliohjelman runko. Dokumentoidaan aliohjelma myös saman tien.

```
/// <summary>
/// Kutsutaan PiirraLumiukko-aliohjelmaa
/// sopivilla parametreilla.
/// </summary>
public override void Begin()
{
  Camera.ZoomToLevel();
  Level.BackgroundColor = Color.Black;

  PiirraLumiukko(this, 0, Level.Bottom + 200.0);
  PiirraLumiukko(this, 200.0, Level.Bottom + 300.0);
}

/// <summary>
/// Aliohjelma piirtää lumiukon
/// annettuun paikkaan.
/// </summary>
/// <param name="peli">Peli, johon lumiukko tehdään.</param>
/// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
/// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
public static void PiirraLumiukko(Game peli, double x, double y)
{
}
```

Alla oleva kuva selvittää aliohjelmakutsun ja aliohjelman määrittelyn sekä vastinparametrien yhteyttä.

```
  ...
  PiirraLumiukko(this, 0, Level.Bottom + 200.0);
  PiirraLumiukko(this, 200.0, Level.Bottom + 300.0);
}

/// <summary> ...
public static void PiirraLumiukko(Game peli, double x, double y)
...
```

Kuva 5: Aliohjelmakutsu ja aliohjelman vastinparametrit.

Aliohjelman toteutuksen ensimmäistä riviä

```
public static void PiirraLumiukko(Game peli, double x, double y)
```

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**), sillä jos emme määrittelisi aliohjelmaa staattiseksi, olisi se oikeastaan metodi, eli olion toiminto (ks. luku 8.5).

Aliohjelmalle on annettu myös palautusarvoksi `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. `void`-määrityksen jälkeen aliohjelmalle on annettu nimeksi `PiirraLumiukko`.

> Huomaa! C#:ssa aliohjelmat kirjoitetaan tyypillisesti isolla alkukirjaimella.

> Huomaa! Aliohjelmien (ja metodien) nimien tulisi olla verbejä tai tekemistä ilmaisevia lauseita, esimerkiksi `LuoPallo`, `Siirry`, `TormattiinEsteeseen`.

Aliohjelman nimen jälkeen ilmoitetaan sulkeiden sisässä aliohjelman parametrit. Jokaista parametria ennen on ilmoitettava myös parametrin *tietotyyppi*. Parametrinä annettiin lumiukon alimman pallon x- ja y-koordinaatit. Molempien tietotyyppi on `double`, joten myös vastinparametrien tyyppien tulee olla `double`. Annetaan myös nimet kuvaavasti `x` ja `y`.

Tietotyypeistä voit lukea lisää kohdasta 7.2 ja luvusta 8.

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

Aliohjelmakutsulla ja aliohjelman määrittelyllä on siis hyvin vahva yhteys keskenään. Aliohjelmakutsussa annetut tiedot "sijoitetaan" kullakin kutsukerralla aliohjelman määrittelyrivillä esitellyille vastinparametreille. Toisin sanoen, aliohjelmakutsun yhteydessä tapahtuu väljästi sanottuna seuraavaa.

```
aliohjelman Game = this;
aliohjelman x = 200.0;
aliohjelman y = Level.Bottom + 300;
```

Voimme nyt kokeilla ajaa ohjelmaamme. Se toimii (lähtee käyntiin), mutta ei tietenkään vielä piirrä

lumiukkoja, eikä pitäisikään, sillä luomamme aliohjelma on ”tyhjä”. Lisätään aaltosulkujen väliin varsinainen koodi, joka pallojen piirtämiseen tarvitaan.

Pieni muutos aikaisempaan versioon kuitenkin tarvitaan. Rivit, joilla pallot lisätään kentälle, muutetaan muotoon

```
peli.Add(...);
```

missä pisteiden paikalle tulee pallo-olion muuttujan nimi.

```
/// <summary>
/// Kutsutaan PiirraLumiukko-aliohjelmaa
/// sopivilla parametreilla.
/// </summary>
public override void Begin()
{
  Camera.ZoomToLevel();
  Level.BackgroundColor = Color.Black;

  PiirraLumiukko(this, 0, Level.Bottom + 200.0);
  PiirraLumiukko(this, 200.0, Level.Bottom + 300.0);
}

/// <summary>
/// Aliohjelma piirtää lumiukon
/// annettuun paikkaan.
/// </summary>
/// <param name="g">Peli, johon lumiukko tehdään.</param>
/// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
/// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
public static void PiirraLumiukko(Game peli, double x, double y)
{
  PhysicsObject p1, p2, p3;
  p1 = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
  p1.X = x;
  p1.Y = y;
  peli.Add(p1);

  p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
  p2.X = x;
  p2.Y = p1.Y + 100 + 50;
  peli.Add(p2);

  p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
  p3.X = x;
  p3.Y = p2.Y + 50 + 30;
  peli.Add(p3);
}
```

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 sisältöä edellisen esimerkin koodiin, on se täysin sama.

C#:ssa on tapana aloittaa aliohjelmien ja metodien nimet isolla kirjaimella ja nimessä esiintyvä jokainen uusi sana alkamaan isolla kirjaimella. Kirjoitustavasta käytetään termiä **PascalCasing**. Muuttujat kirjoitetaan pienellä alkukirjaimella, ja jokainen seuraava sana isolla alkukirjaimella: esimerkiksi `double autonNopeus`. Tästä käytetään nimeä **camelCasing**. Lisää C#:n nimeämiskäytännöistä voit lukea sivulta

http://msdn.microsoft.com/en-us/library/ms229043.aspx.

Tarkastellaan seuraavaksi mitä aliohjelmakutsussa tapahtuu.

```
PiirraLumiukko(this, 0, Level.Bottom + 200.0);
```

Yllä olevalla kutsulla aliohjelman `peli`-nimiseen muuttujaan sijoitetaan `this`, eli kyseessä oleva peli, `x`-nimiseen muuttujaan sijoitetaan arvo 0 (liukulukuun voi sijoittaa kokonaislukuarvon) ja aliohjelman muuttujaan `y` arvo `Level.Bottom + 200.0`. Voisimme sijoittaa tietenkin minkä tahansa muunkin liukuluvun.

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: kokonaislukuarvo 0 ja kokonaislukuarvo 50 ja muuttujan nimi. Ne voisivat kuitenkin olla kuinka monimutkaisia lausekkeita tahansa, esimerkiksi näin:

```
PiirraLumiukko(this, 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 6: Lauseen ja lausekkeen ero

Koska määrittelimme koordinaattien parametrien tyypiksi `double`, voisimme yhtä hyvin antaa parametreiksi mitä tahansa muitakin desimaalilukuja. Täytyy muistaa, että C#:ssa desimaaliluvuissa käytetään pistettä erottamaan kokonaisosa desimaaliosasta.

Kokonaisuudessaan ohjelma näyttää nyt seuraavalta:

```
01 /// @author Antti-Jussi Lakanen
02 /// @version 22.12.2011
03
04
05 using Jypeli;
06
07 /// <summary>
08 /// Piirretään lumiukkoja ja harjoitellaan aliohjelman käyttöä.
09 /// </summary>
10 public class Lumiukot : PhysicsGame
11 {
12  /// <summary>
13  /// Pääohjelmassa laitetaan "peli" käyntiin.
14  /// </summary>
15  /// <param name="args">Ei käytössä</param>
16  public static void Main(String[] args)
17  {
18    using (Lumiukot peli = new Lumiukot())
19    {
20     peli.Run();
21    }
22  }
23
24  /// <summary>
25  /// Kutsutaan PiirraLumiukko-aliohjelmaa
26  /// sopivilla parametreilla.
27  /// </summary>
28  public override void Begin()
29  {
30   Camera.ZoomToLevel();
31   Level.BackgroundColor = Color.Black;
32
33   PiirraLumiukko(this, 0, Level.Bottom + 200.0);
34   PiirraLumiukko(this, 200.0, Level.Bottom + 300.0);
35  }
36
37  /// <summary>
38  /// Aliohjelma piirtää lumiukon
39  /// annettuun paikkaan.
40  /// </summary>
41  /// <param name="peli">Peliluokka</param>
42  /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
43  /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
44  public static void PiirraLumiukko(Game peli, double x, double y)
45  {
46    PhysicsObject p1, p2, p3;
47    p1 = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
48    p1.X = x;
49    p1.Y = y;
50    peli.Add(p1);
51
52    p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
53    p2.X = x;
54    p2.Y = p1.Y + 100 + 50;
55    peli.Add(p2);
56
57    p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
58    p3.X = x;
59    p3.Y = p2.Y + 50 + 30;
60    peli.Add(p3);
61  }
62 }
```

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 ensimmäisen kutsun puolipisteeseen ja sitten pääohjelma jatkuu kutsumalla toista lumiukon piirtämistä.

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

> Huomaa! 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 seuraavat asiat: Lyhyt kuvaus aliohjelman toiminnasta, selitys kaikista parametreista sekä selitys mahdollisesta paluuarvosta. Nämä asiat kuvataan tägien avulla seuraavasti:

- Dokumentaatiokommentin alkuun laitetaan `<summary>`-tagien väliin lyhyt ja selkeä kuvaus aliohjelman toiminnasta.
- Jokainen parametri selitetään omien `<param>`-tagien väliin ja
- paluuarvo `<returns>`-tagien väliin.

`PiirraLumiukko`-aliohjelman dokumentaatiokommentit on edellisessä esimerkissämme riveillä 36-41.

```
36 /// <summary>
37 /// Aliohjelma piirtää lumiukon
38 /// annettuun paikkaan.
39 /// </summary>
40 /// <param name="g">Peliluokka</param>
41 /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
42 /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
```

Doxygen-työkalun (ks. http://en.wikipedia.org/wiki/Doxygen) tuottama HTML-sivu tästä luokasta näyttäisi nyt seuraavalta:

Lumiukot

Main Page | Classes | Files

Class List | Class Index | Class Members

- Lumiukot
  - Class List
    - Lumiukot
  - Class Index
  - Class Members
  - File List

Piirretään lumiukkoja ja harjoitellaan aliohjelman käyttöä. More...

List of all members.

Public Member Functions

override void **Begin** ()
Kutsutaan PiirraLumiukko-aliohjelmaa sopivilla parametreilla.

Static Public Member Functions

static void **Main** (string[] args)
Pääohjelmassa laitetaan "peli" käyntiin.

static void **PiirraLumiukko** (Game g, double x, double y)
Aliohjelma piirtää lumiukon annettuun paikkaan.

Detailed Description

Piirretään lumiukkoja ja harjoitellaan aliohjelman käyttöä.

Definition at line 10 of file Lumiukot.cs.

Member Function Documentation

```
override void Lumiukot.Begin ( )
```

Kutsutaan PiirraLumiukko-aliohjelmaa sopivilla parametreilla.

Definition at line 28 of file Lumiukot.cs.

```
static void Lumiukot.Main ( string[] args ) [static]
```

Pääohjelmassa laitetaan "peli" käyntiin.

Parameters:
args Ei käytössä

Definition at line 16 of file Lumiukot.cs.

```
static void Lumiukot.PiirraLumiukko ( Game   g,
                                      double x,
                                      double y
                                    )        [static]
```

Aliohjelma piirtää lumiukon annettuun paikkaan.

Parameters:
g Peliluokka
x Lumiukon alimman pallon x-koordinaatti.
y Lumiukon alimman pallon y-koordinaatti.

Definition at line 43 of file Lumiukot.cs.

Lumiukot

Kuva 7: Osa `Lumiukot`-luokan dokumentaatiosta

Dokumentaatiossa näkyy kaikki luokan aliohjelmat ja metodit. Huomaa, että Doxygen nimittää sekä aliohjelmia että metodeja jäsenfunktioiksi (**member functions**). Kuten sanottu, nimitykset vaihtelevat kirjallisuudessa, ja tässä kohtaa käytössä on hieman harvinaisempi nimeämistapa. Kysymys on kuitenkin samasta asiasta, josta me tällä kurssilla käytämme nimeä aliohjelmat ja metodit.

Jokaisesta aliohjelmasta ja metodista löytyy lisäksi tarkemmat tiedot **Detailed Description** -kohdasta. Aliohjelman `PiirraLumiukko` dokumentaatio parametreineen näkyy kuvan alaosassa.

6.3.1 Huomautus

Kaikki `PiirraLumiukko`-aliohjelmassa tarvittava tieto välitettiin parametrien avulla, eikä aliohjelman suorituksen aikana tarvittu aliohjelman ulkopuolisia tietoja. Tämä on tyypillistä aliohjelmille, ja usein lisäksi toivottava ominaisuus.

6.4 Aliohjelmat, metodit ja funktiot

Kuten ehkä huomasit, aliohjelmilla ja metodeilla on paljon yhteistä. Monissa 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. Jypelin dokumentaatiosta tutkit `RandomGen`-luokan staattisia metodeja, millä voidaan luoda esimerkiksi satunnaisia lukuja. Yksittäinen pallo poistettiin metodilla `Destroy`, joka on olion toiminto.

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

Aliohjelmamme `PiirraLumiukko` 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, kuten Doxygen teki myös C#:n tapauksessa.

7. Muuttujat

Muuttujat (**variable**) toimivat ohjelmassa tietovarastoina erilaisille asioille. Muuttuja on kuin pieni laatikko, johon voidaan varastoida asioita, esimerkiksi lukuja, sanoja, tietoa ohjelman käyttäjästä ja paljon muuta. Ilman muuttujia järkevä tiedon käsittely olisi oikeastaan mahdotonta. Olemme jo ohimennen käyttäneetkin muuttujia, esimerkiksi Lumiukko-esimerkissä teimme `PhysicsObject`-tyyppisiä muuttujia `p1`, `p2` ja `p3`. Vastaavasti Lumiukko-aliohjelman parametrit (`Game peli`, `double x`, `double y`) ovat myös muuttujia: `Game`-tyyppinen oliomuuttuja `peli`, sekä `double`-alkeistietotyyppiset muuttujat `x` ja `y`.

Termi *muuttuja* on lainattu ohjelmointiin matematiikasta, mutta niitä ei tule kuitenkaan sekoittaa keskenään – muuttuja matematiikassa ja muuttuja ohjelmoinnissa tarkoittavat hieman eri asioita. Tulet huomaamaan tämän seuraavien kappaleiden aikana.

Muuttujien arvot tallennetaan keskusmuistiin tai rekistereihin, mutta ohjelmointikielissä voimme antaa kullekin muuttujalle nimen (**identifier**), jotta muuttujan arvon käsittely olisi helpompaa. 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

Kun matemaatikko sanoo, että ”`n` on yhtäsuuri kuin 1”, tarkoittaa se, että tuo termi (eli muuttuja) `n` on jollain käsittämättömällä tavalla sama kuin luku 1. Matematiikassa muuttujia voidaan esitellä tällä tavalla ”häthätää”.

Ohjelmoijan on kuitenkin tehtävä vastaava asia hieman tarkemmin. C#-kielessä tämä tapahtuisi kirjoittamalla seuraavasti:

```
int n;
n = 1;
```

Ensimmäinen rivi tarkoittaa väljästi sanottuna, että ”lohkaise pieni pala – johon mahtuu `int`-kokoinen arvo – säilytystilaa tietokoneen muistista, ja anna sille nimeksi n”. Toisella rivillä julistetaan, että ”talleta arvo 1 muuttujaan, jonka nimi on n, siten korvaten sen, mitä kyseisessä säilytystilassa mahdollisesti jo on”.

Mikä sitten on tuo edellisen esimerkin `int`?

C#:ssa jokaisella muuttujalla täytyy olla *tietotyyppi* (usein myös lyhyesti *tyyppi*). Tietotyyppi on määriteltävä, jotta ohjelma tietäisi, millaista tietoa muuttujaan tullaan tallentamaan. Toisaalta tietotyyppi on määriteltävä siksi, että ohjelma osaa varata muistista sopivan kokoisen lohkareen muuttujan sisältämää tietoa varten. Esimerkiksi `int`-tyypin tapauksessa tilantarve olisi 32 bittiä (4 tavua), byte-tyypin tapauksessa 8 bittiä (1 tavu) ja `double`-tyypin 64 bittiä (8 tavua). Muuttuja määritellään (**declare**) kirjoittamalla ensiksi tietotyyppi ja sen perään muuttujan nimi. Muuttujan nimet aloitetaan C#ssa pienellä kirjaimella, jonka jälkeen jokainen uusi sana alkaa aina isolla kirjaimella. Kuten aiemmin mainittiin, tämä nimeämistapa on nimeltään **camelCasing**.

```
muuttujanTietotyyppi muuttujanNimi;
```

Tuo mainitsemamme `int` on siis tietotyyppi, ja `int`-tyyppiseen muuttujaan voi tallentaa kokonaislukuja. Muuttujaan `n` voimme laittaa lukuja 1, 2, 3, samoin 0, -1, -2, ja niin edelleen, mutta emme lukua 0.1 tai sanaa ”Moi”.

Henkilön iän voisimme tallentaa seuraavaan muuttujaan:

```
int henkilonIka;
```

Huomaa, että tässä emme aseta muuttujalle mitään arvoa, vain määrittelemme muuttujan `int`-tyyppiseksi ja annamme sille nimen.

Samantyyppisiä muuttujia voidaan määritellä kerralla useampia erottamalla muuttujien nimet pilkulla. Tietotyyppiä `double` käytetään, kun halutaan tallentaa desimaalilukuja.

```
double paino, pituus;
```

Määrittely onnistuu kuitenkin myös erikseen.

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

7.2 Alkeistietotyypit

C#:n tietotyypit voidaan jakaa alkeistietotyyppeihin (**primitive types**) ja oliotietotyyppeihin (**reference types**). Oliotietotyyppeihin kuuluu muun muassa käyttämämme `PhysicsObject`-tyyppi, jota pallot `p1` jne. olivat, sekä merkkijonojen tallennukseen tarkoitettu `String`-olio. Oliotyyppejä käsitellään myöhemmin luvussa 8.

Eri tietotyypit vaativat eri määrän kapasiteettia tietokoneen muistista. Vaikka nykyajan koneissa on paljon muistia, on hyvin tärkeää valita oikean tyyppinen muuttuja kuhunkin tilanteeseen. Suurissa ohjelmissa ongelma korostuu hyvin nopeasti käytettäessä muuttujia, jotka kuluttavat tilanteeseen nähden kohtuuttoman paljon muistikapasiteettia. C#:n alkeistietotyypit on lueteltu alla.

Taulukko 1: C#:n alkeistietotyypit koon mukaan järjestettynä.

C#-merkki	Koko	Selitys	Arvoalue
`bool`	1 bitti	kaksiarvoinen tietotyyppi	true tai false
`sbyte`	8 bittiä	yksi tavu	-128..127
`byte`	8 bittiä	yksi tavu (etumerkitön)	0..255
`char`	16 bittiä	yksi merkki	kaikki merkit
`short`	16 bittiä	pieni kokonaisluku	-32,768..32,767
`ushort`	16 bittiä	pieni kokonaisluku (etumerkitön)	0..65,535
`int`	32 bittiä	kokonaisluku	-2,147,483,648.. 2,147,483,647
`uint`	32 bittiä	kokonaisluku (etumerkitön)	0..4,294,967,295
`float`	32 bittiä	liukuluku	noin 7 desimaalin tarkkuus, $\pm1.5 \times 10^{-45}$.. $\pm3.4 \times 10^{38}$
`long`	64 bittiä	iso kokonaisluku	$-2^{63}..2^{63}-1$
`ulong`	64 bittiä	iso kokonaisluku (etumerkitön)	0.. 18,446,744,073,709,615
`double`	64 bittiä	tarkka liukuluku	noin 15 desimaalin tarkkuus, $\pm5.0 \times 10^{-324}$.. $\pm1.7 \times 10^{308}$
`decimal`	128 bittiä	erittäin tarkka liukuluku	Noin 28 luvun tarkkuus

Tässä monisteessa suositellaan, että desimaalilukujen talletukseen käytettäväksi aina `double`-tietotyyppiä (jossain tapauksissa jopa `decimal`-tyyppiä), 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. 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 tallettamaan enemmän merkitseviä desimaaleja.

Tietyissä sovelluksissa, joissa mahdollisimman suuri tarkkuus on välttämätön (kuten pankki- tai nanotason fysiikkasovellukset), on suositeltavaa käyttää korkeimpaa mahdollista tarkkuutta tarjoavaa `decimal`-tyyppiä. Reaalilukujen esityksestä tietokoneessa puhutaan lisää kohdassa 26.6. [VES][KOS]

7.3 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;
```

Muuttujaan voi asettaa arvon myös jo määrittelyn yhteydessä.

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

Muuttuja täytyy olla määritelty ennen kuin siihen voi asettaa arvoa. Muuttujaan voi asettaa vain määrittelyssä annetun tietotyypin mukaisia arvoja tai sen kanssa *sijoitusyhteensopivia* arvoja. Esimerkiksi liukulukutyyppeihin (`float` ja `double`) voi sijoittaa myös kokonaislukutyyppisiä arvoja, sillä kokonaisluvut on reaalilukujen *osajoukko*.

```
double luku1;
int luku2 = 4;
luku1 = luku2;
```

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

```
//TÄMÄ KOODI EI KÄÄNNY!
int luku1;
double luku2 = 4.0;
luku1 = luku2;
```

Kun `decimal`-tyyppinen muuttuja alustetaan jollain luvulla, tulee luvun perään (ennen puolipistettä) laittaa `m` (tai `M`)-merkki. Samoin `float`-tyyppisten muuttujien alustuksessa perään laitetaan `f` (tai `F`)-merkki.

```
decimal tilinSaldo = 3498.98m;
float lampotila = -4.8f;
```

Huomaa, että `char`-tyyppiseen muuttujaan sijoitetaan arvo laittamalla merkki heittomerkkien väliin, esimerkiksi näin.

```
char ekaKirjain = 'k';
```

Näin sen erottaa myöhemmin käsiteltävästä `String`-tyyppiseen muuttujaan sijoittamisesta, jossa sijoitettava merkkijono laitetaan lainausmerkkien väliin, esimerkiksi seuraavasti.

```
String omaNimi = "Antti-Jussi";
```

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.

C#:ssa 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.Console.WriteLine(ika);
  }
}
```

Virheilmoitus näyttää tältä:

```
Esimerkki.cs(4,40): error CS0165: Use of unassigned local variable
        'ika'
```

Kääntäjä kertoo, että `ika`-nimistä muuttuja yritetään käyttää, vaikka sille ei ole osoitettu mitään arvoa. Tämä ei ole sallittua, joten ohjelman kääntämisyritys päättyy tähän.

7.4 Muuttujan nimeäminen

Muuttujan nimen täytyy olla siihen tallennettavaa tietoa kuvaava. Yleensä pelkkä yksi kirjain on huono nimi muuttujalle, sillä se harvoin kuvaa kovin hyvin muuttujaa. Kuvaava muuttujan nimi selkeyttää koodia ja vähentää kommentoimisen tarvetta. Lyhyt muuttujan nimi ei ole itseisarvo. Vielä parikymmentä vuotta sitten se saattoi olla sitä, koska se nopeutti koodin kirjoittamista. Nykyaikaisia kehitysympäristöjä käytettäessä tämä ei enää pidä paikkaansa, sillä editorit osaavat täydentää muuttujan nimen samalla kun koodia kirjoitetaan, joten niitä ei käytännössä koskaan tarvitse kirjoittaa kokonaan, paitsi tietysti ensimmäisen kerran.

Yksikirjaimisia muuttujien nimiäkin voi perustellusti käyttää, jos niillä on esimerkiksi jo matematiikasta tai fysiikasta ennestään tuttu merkitys. Nimet `x` ja `y` ovat hyviä kuvaamaan koordinaatteja. Nimi `l` (eng. **length**) viittaa pituuteen ja `r` (eng. **radius**) säteeseen. Fysikaalisessa ohjelmassa `s` voi hyvin kuvata matkaa.

> Huomaa! Muuttujan nimi ei saa C#:ssa koskaan alkaa numerolla.

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

```
int polkupyoranRenkaanKoko;
```

C#:ssa muuttujan nimi voi sisältää ääkkösiä, mutta niiden käyttö ei suositella, koska siirtyminen *koodistosta* toiseen aiheuttaa usein ylimääräisiä ongelmia.

> *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 seitsemänbittiseen ASCII-koodistoon, joka ei tue skandeja. ASCII-koodistosta puhutaan lisää luvussa 27.

7.4.1 C#:n varatut sanat

Muuttujan nimi ei saa olla mikään ohjelmointikielen varatuista sanoista, eli sanoista joilla on C#:ssa joku muu merkitys.

Taulukko 2: C#:n varatut sanat.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

7.5 Muuttujien näkyvyys

Muuttujaa 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.

Luokan sisällä muuttuja voidaan määritellä myös niin, että se näkyy kaikkialla, siis kaikille aliohjelmille. Kun muuttuja on näkyvissä kaikille ohjelman osille, sanotaan sitä *globaaliksi muuttujaksi* (**global variable**). Globaaleja muuttujia tulee välttää aina kun mahdollista.

7.6 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. C#:ssa vakio määritellään muuten kuten muuttuja, mutta muuttujan tyypin eteen kirjoitetaan lisämääre `const`.

```
const int KUUKAUSIEN_LKM = 12;
```

Tällä kurssilla vakiot kirjoitetaan suuraakkosin siten, että sanat erotetaan alaviivalla (_). Näin ne erottaa helposti muuttujien nimistä, jotka alkavat pienellä kirjaimella. Muitakin kirjoitustapoja on, esimerkiksi Pascal Casing on toinen yleisesti käytetty vakioiden kirjoitusohje.

7.7 Operaattorit

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

C#:ssa on myös vertailuoperaattoreita (**comparison operators**), loogisia operaattoreita, bittikohtaisia operaattoreita (**bitwise operators**), arvonmuunto-operaattoreita (**shortcut operators**), sijoitusoperaattori `=` , `is`-operaattori sekä ehto-operaattori `?` . Tässä luvussa käsitellään näistä tärkeimmät.

7.7.1 Aritmeettiset operaatiot

C#:ssa peruslaskutoimituksia suoritetaan aritmeettisilla operaatiolla, joista + ja – tulivatkin esille aikaisemmissa esimerkeissä. Aritmeettisia operaattoreita on viisi.

Taulukko 3: Aritmeettiset operaatiot.

Operaattori	Toiminto	Esimerkki
`+`	yhteenlasku	`System.Console.WriteLine(1+2); // tulostaa 3`
`-`	vähennyslasku	`System.Console.WriteLine(1-2); // tulostaa -1`
`*`	kertolasku	`System.Console.WriteLine(2*3); // tulostaa 6`

`/`	jakolasku	`System.Console.WriteLine(6 / 2); // tulostaa 3`
		`System.Console.WriteLine(7 / 2); //` **`Huom! 3`**
		`System.Console.WriteLine(7 / 2.0); // 3.5`
		`System.Console.WriteLine(7.0 / 2); // 3.5`

`%`	jakojäännös (modulo)	`System.Console.WriteLine(18 % 7); // tulostaa 4`

7.7.2 Vertailuoperaattorit

Vertailuoperaattoreiden avulla verrataan muuttujien arvoja keskenään. Vertailuoperaattorit palauttavat totuusarvon (`true` tai `false`). Vertailuoperaattoreita on kuusi. Lisää vertailuoperaattoreista luvussa 13.

7.7.3 Arvonmuunto-operaattorit

Arvonmuunto-operaattoreiden avulla laskutoimitukset voidaan esittää tiiviimmässä muodossa: esimerkiksi `x++;` (4 merkkiä) tarkoittaa samaa asiaa kuin `x = x+1;` (6 merkkiä). Niiden avulla voidaan myös alustaa muuttujia.

Taulukko 4: Arvonmuunto-operaattorit.

Operaattori	Toiminto	Esimerkki

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

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

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

`-=`	Vähennys-operaatio	`int luku = 0;`
		`luku -= 2; // luku muuttujan arvo on -2`
		`luku -= 1 // luku muuttujan arvon -3`

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

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

`%=`	Jakojäännös-operaatio	`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.

> Huomaa! Arvonmuunto-operaattorit ovat ns. sivuvaikutuksellisia operaattoreita. Toisin sanoen, operaatio muuttaa muuttujan arvoa toisin kuin esimerkiksi aritmeettiset operaatiot. Seuraava esimerkki havainnollistaa asiaa.

```
int luku1 = 5;
int luku2 = 5;
System.Console.WriteLine(++luku1); // tulostaa 6;
System.Console.WriteLine(luku2 + 1 ); // tulostaa 6;
System.Console.WriteLine(luku1); // 6
System.Console.WriteLine(luku2); // 5
```

7.7.4 Aritmeettisten operaatioiden suoritusjärjestys

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

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

7.8 Huomautuksia

7.8.1 Kokonaisluvun tallentaminen liukulukumuuttujaan

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.Console.WriteLine(laskunTulos); //tulostaa 2
```

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

```
double laskunTulos = 5 / 2.0;
System.Console.WriteLine(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.Console.WriteLine(laskunTulos); //tulostaa 1

laskunTulos = 5 / 6;
System.Console.WriteLine(laskunTulos); //tulostaa 0

laskunTulos = 7 / 3;
System.Console.WriteLine(laskunTulos); //tulostaa 2
```

Kokonaisluvuilla laskettaessa lukuja ei siis pyöristetä lähimpään kokonaislukuun, vaan desimaaliosa menee C#:n jakolaskuissa ikään kuin "hukkaan".

7.8.2 Lisäys- ja vähennysoperaattoreista

On neljä tapaa kasvattaa luvun arvoa yhdellä.

```
++a;
a++; // idiomi
a += 1;
a = a + 1; // huonoin muutettavuuden ja kirjoittamisen kannalta
```

Ohjelmoinnissa *idiomilla* tarkoitetaan tapaa jolla asia yleensä kannattaa tehdä. Näistä `a++` on ohjelmoinnissa vakiintunut tapa ja (yleensä) suositeltavin, siis idiomi. Kuitenkin, jos lukua `a` pitäisikin kasvattaa (tai vähentää) kahdella tai kolmella, ei tämä tapa enää toimisi. Seuraavassa esimerkissä tarkastellaan eri tapoja kahdella vähentämiseksi. Siihen on kolme vaihtoehtoista tapaa.

```
a -= 2;
a += -2;  // lisättävä luku voisi olla muuttuja. Luku voi olla myös negatiivinen
a = a - 2;
```

Tässä tapauksessa `+=` -operaattorin käyttö olisi suositeltavinta, sillä lisättävä luku voi olla positiivinen tai negatiivinen (tai nolla), joten `+=` -operaattori ei tässä rajoita sitä, millaisia lukuja `a`-muuttujaan voidaan lisätä.

7.9 Esimerkki: Painoindeksi

Tehdään ohjelma joka laskee painoindeksin. Painoindeksi lasketaan jakamalla paino (kg) pituuden (m) neliöllä, eli kaavalla

```
paino / pituus * pituus
```

C#:lla painoindeksi saadaan siis laskettua seuraavasti.

```
/*
 * Author: Antti-Jussi Lakanen
 */

/// <summary>
/// Ohjelma, joka laskee painoindeksin
/// pituuden (m) ja painon (kg) perusteella.
/// </summary>
public class Painoindeksi
{
    /// <summary>
    /// Pääohjelma, jossa painoindeksi tulostetaan ruudulle.
    /// </summary>
    /// <param name="args">Ei käytössä.</param>
    public static void Main(string[] args)
    {
      double pituus = 1.83;
      double paino = 75.0;
      double painoindeksi = paino / (pituus*pituus);
      System.Console.WriteLine(painoindeksi);
   }
}
```

8. Oliotietotyypit

C#:n alkeistietotyypit antavat melko rajoittuneet puitteet ohjelmointiin. Niillä pystytään tallentamaan ainoastaan lukuja ja yksittäisiä kirjaimia. Vähänkin monimutkaisemmissa ohjelmissa tarvitaan kehittyneempiä rakenteita tiedon tallennukseen. C#:ssa, Javassa ja muissa oliokielissä tällaisen rakenteen tarjoavat oliot. C#:ssa jo merkkijonokin (`String`) toteutetaan oliona.

8.1 Mitä oliot ovat?

Olio (**object**) on tietorakenne, jolla pyritään ohjelmoinnissa kuvaamaan mahdollisimman tarkasti reaalimaailman ilmiöitä. Luokkapohjaisissa kielissä (kuten C#, Java ja 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 saada samasta luokasta luoduilla olioilla eri arvoja. Attribuuttien arvot muodostavat olion tilan. Huomaa kuitenkin, että vaikka oliolla olisi sama tila, sen *identiteetti* on eri. Esimerkiksi, kaksi täsmälleen samannäköistä palloa voi olla samassa paikassa (näyttää yhdeltä pallolta), mutta todellisuudessa ne ovat kaksi eri palloa.

Olioita voi joko tehdä itse tai käyttää jostain kirjastosta löytyviä valmiita olioita. Omien olioluokkien tekeminen ei kuulu vielä Ohjelmointi 1 -kurssin asioihin, mutta käyttäminen kyllä. Tarkastellaan seuraavaksi luokan ja olion suhdetta, sekä kuinka oliota käytetään.

Luokan ja olion suhdetta voisi kuvata seuraavalla esimerkillä. Olkoon luentosalissa useita ihmisiä. Kaikki luentosalissa olijat ovat ihmisiä. Heillä on tietyt samat ominaisuudet, jotka ovat kaikilla ihmisillä, kuten pää, kaksi silmää ja muitakin ruumiinosia. Kuitenkin jokainen salissa olija on erilainen ihmisen ilmentymä, eli jokaisella oliolla on oma identiteetti – eiväthän he ole yksi ja sama vaan heitä on useita. Eri ihmisillä voi olla erilainen tukka ja eriväriset silmät ja oma puhetyyli. 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`, `tehtava`, `osasto`, `palkka`. Luokalla täytyisi olla myös ainakin seuraavat metodit: `MaksaPalkka`, `MuutaTehtava`, `MuutaOsasto`, `MuutaPalkka`. Jokainen työntekijä olisi nyt omanlaisensa `Tyontekija`-luokan ilmentymä eli olio.

8.2 Olion luominen

```
Tyontekija tyontekija = new Tyontekija("Teppo Tunari", "Projektipäällikkö",
"Tutkimusosasto", 5000);
```

Olioviite määritellään kirjoittamalla ensiksi sen luokan nimi, josta olio luodaan. Seuraavaksi kirjoitetaan nimi, jonka haluamme oliolle antaa. Nimen jälkeen tulee yhtäsuuruusmerkki, jonka jälkeen oliota luotaessa kirjoitetaan sana `new` ilmoittamaan, että luodaan uusi olio. Tämä `new`-operaattori varaa tilan tietokoneen muistista oliota varten.

Seuraavaksi kirjoitetaan luokan nimi uudelleen, jonka perään kirjoitetaan sulkuihin mahdolliset olion luontiin liittyvät parametrit. Parametrit riippuvat siitä kuinka luokan *konstruktori* (**constructor**, muodostaja) on toteutettu. Konstruktori 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,..., parametriN);
```

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

> Ennen kuin oliolle on varattu tila tietokoneen muistista `new`-operaattorilla, ei sitä voi käyttää. Ennen `new`-operaattorin käyttöä olion arvo on `null`, joka tarkoittaa, että olio on käyttökelvoton, ”mitätön”. Tällaisen olion käyttö aiheuttaa virheilmoituksen. Olio voidaan myös tarkoituksellisesti merkitä käyttökelvottomaksi asettamalla `olionNimi = null`.

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 `PhysicsObject`-luokan olion seuraavasti.

```
PhysicsObject p1 = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
```

Itse asiassa oliomuuttuja on C#:ssa ainoastaan *viite* varsinaiseen olioon. Siksi niitä kutsutaankin usein myös *viitemuuttujiksi.* Viitemuuttujat eroavat oleellisesti alkeistietotyyppisistä muuttujista.

8.3 Oliotietotyyppien ja alkeistietotyyppien ero

C#:ssa on kahden mallisia rakenteita, joihin tietoja voidaan tallentaa. Tapauksesta riippuen tiedot tallennetaan joko *alkeistietotyyppeihin* tai *oliotietotyyppeihin*. Oliotietotyypit eroavat alkeistietotyypeistä siinä, että ne ovat *viitteitä* tiettyyn olioon, ja siksi niitä kutsutaan myös nimellä viitetyypit tai viitemuuttujat.

- Alkeistietotyypit säilyttävät sisältämänsä tiedon yhdessä paikassa tietokoneen muistissa (nimeltään *pino*).
- Viitetyypit sisältävät viitteen johonkin toiseen paikkaan muistissa (kutsutaan nimellä *keko*), missä varsinainen data sijaitsee. Viittaus olioon sijaitsee kuitenkin pinossa.

Yleensä meidän ei tarvitse olla kovin huolissamme siitä käytämmekö alkeistietotyyppiä (kuten `int`, `double` tai `char`) vai oliotyyppiä (kuten `String`). Yleisesti ottaen tärkein ero on siinä, että alkeistietotyyppien tulee (tiettyjä poikkeuksia lukuun ottamatta) aina sisältää jokin arvo, mutta oliotietotyypit voivat olla `null`-arvoisia (eli ”ei-minkään” arvoisia). Jäljempänä esimerkkejä alkeistietotyyppien ja viitetyyppien eroista.

Samaan olioon voi viitata useampi muuttuja. Vertaa alla olevia koodinpätkiä.

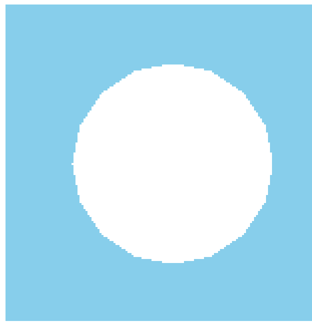
```
int luku1 = 10;
int luku2 = luku1;
luku1 = 0;
Console.WriteLine(luku2); //tulostaa 10
```

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:

```
PhysicsObject p1 = new PhysicsObject(2*100.0, 2*100.0, Shape.Circle);
Add(p1);
p1.X = -50;

PhysicsObject p2 = p1;
p2.X = 100;
```

Yllä oleva koodi piirtää seuraavan kuvan:

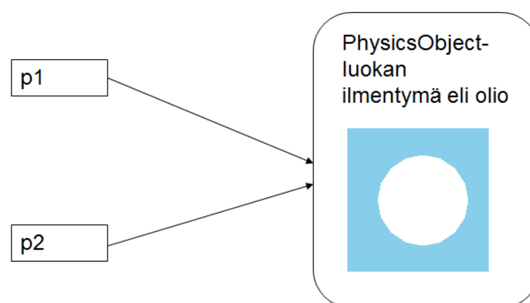


Kuva 8: Molemmat muuttujat, p1 ja p2, liikuttelevat samaa ympyrää. Lopputuloksena ympyrä seisoo pisteessä x=100.

Nopeasti voisi olettaa, että ikkunassamme näkyisi nyt vain kaksi samanlaista ympyrää eri paikoissa. Näin ei kuitenkaan ole, vaan molemmat `PhysicsObject`-oliot viittaavat samaan ympyrään, jonka säde on 50. Tämä johtuu siitä, että muuttujat `p1` ja `p2` ovat olioviitteitä, jotka viittaavat (ts. osoittavat) samaan olioon.

```
PhysicsObject p2 = p1;
```

Toisin sanoen yllä olevalla rivillä ei luoda uutta `PhysicsObject`-oliota, vaan ainoastaan uusi olioviite, joka viittaa nyt samaan olioon kuin `p1`.



Kuva 9: Sekä p1 että p2 viittaavat samaan olioon.

Oliomuuttuja = Viite todelliseen olioon. Samaan olioon voi olla useitakin viitteitä.

Viitteitä käsitellään tarkemmin luvussa 14.

8.4 Metodin kutsuminen

Jokaisella tietystä luokasta luodulla oliolla on käytössä kaikki tämän luokan julkiset metodit. Metodikutsussa käsketään oliota tekemään jotain. Voisimme esimerkiksi käskeä `PhysicsObject`-oliota liikkumaan, tai `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,...parametriN);
```

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

```
akuAnkka.MuutaPalkka(3500);
```

Tai laittaa `p1`-olion (oletetaan että `p1` on `PhysicsObject`-olio) liikkeelle käyttäen Hit-metodia.

```
p1.Hit(new Vector(1000.0, 500.0));
```

`String`-luokasta löytyy esimerkiksi `Contains`-metodi, joka palauttaa arvon `True` tai `False`. Parametrina `Contains`-metodille annetaan merkkijono, ja metodi etsii oliosta antamaamme merkkijonoa vastaavia ilmentymiä. Jos olio sisältää merkkijonon (yhden tai useamman kerran) palautetaan `True`. Muutoin palautetaan `False`. Alla esimerkki.

```
String lause = "Pekka meni kauppaan";
Console.WriteLine(lause.Contains("eni")); // Tulostaa True
```

8.5 Metodin ja aliohjelman ero

Aliohjelma esitellään static-tyyppiseksi, mikäli aliohjelma ei käytä mitään muita tietoja kuin parametreinä tuodut tiedot. Esimerkiksi luvussa 20.4.2 on seuraava aliohjelma.

```
private void KuunteleLiiketta(AnalogState hiirenTila)
{
   pallo.X = Mouse.PositionOnWorld.X;
   pallo.Y = Mouse.PositionOnWorld.Y;

   Vector hiirenLiike = hiirenTila.MouseMovement;
}
```

Tässä tarvitaan hiiren tilan lisäksi pelioliossa esitellyn pallon tietoja, joten enää ei ole kyse stattisesta aliohjelmasta ja siksi `static`-sana jätetään pois. Metodi sen sijaan pystyy käyttämään *olion* omia "ominaisuuksia", attribuutteja, metodeja ja ns. ominaisuus-kenttiä (**property fields**).

8.6 Olion tuhoaminen ja roskienkeruu

Kun olioon ei enää viittaa yhtään muuttujaa, täytyy olion käyttämät muistipaikat vapauttaa muuhun käyttöön. Oliot poistetaan muistista puhdistusoperaation avulla. Tästä huolehtii C#:n automaattinen roskienkeruu (**garbage collection**). Kun olioon ei ole enää viitteitä, se merkitään poistettavaksi ja aina tietyin väliajoin *puhdistusoperaattori* (kutsutaan usein myös nimellä roskienkerääjä, **garbage collector**) vapauttaa merkittyjen 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.

Yleensä C#-ohjelmoijan ei tarvitse huolehtia muistin vapauttamisesta, mutta on tiettyjä tilanteita, joissa voidaan itse joutua poistamaan oliot. Yksi esimerkki tällaisesta tilanteesta on tiedostojen käsittely: Jos olio on avannut tiedoston, olisi ennen olion tuhoamista järkevää sulkea tiedosto. Tällöin samassa yhteydessä olion tuhottavaksi merkitsemisen kanssa suoritettaisiin myös tiedoston sulkeminen. Tämä tehdään esittelemällä *hajotin* (**destructor**), joka on luokan metodi ja jonka tehtävänä on tyhjentää olio kaikesta sen sisältämästä tiedosta sekä vapauttaa sen sisältämät rakenteet, kuten kytkökset avoinna oleviin tiedostoihin.

8.7 Olioluokkien dokumentaatio

Luokan dokumentaatio sisältää tiedot luokasta, luokan konstruktoreista ja metodeista. Luokkien dokumentaatioissa on yleensä linkkejä esimerkkeihin, kuten myös `String`-luokan tapauksessa. Tutustutaan nyt tarkemmin `String`-luokan dokumentaatioon. `String`-luokan dokumentaatio löytyy sivulta http://msdn.microsoft.com/en-us/library/system.string.aspx, ja lista *jäsenistä* eli käytössä olevista konstruktoreista, attribuuteista (**fields**), ominaisuuksista (**property**) ja metodeista sivulta http://msdn.microsoft.com/en-us/library/system.string_members.aspx.

Olemme kiinnostuneita tässä vaiheessa kohdista `String Constructor` ja `String Methods` (sivun vasemmassa osassa hierarkiapuussa). Klikkaa kohdasta `String Constructor` saadaksesi lisätietoa luokan konstruktoreista tai `String Methods` saadaksesi tietoja käytössä olevista metodeista.

8.7.1 Konstruktorit

Avaa luokan `String` sivu `String Constructor`. Tämä kohta sisältää tiedot kaikista luokan konstruktoreista. Konstruktoreita voi olla useita, kunhan niiden parametrit eroavat toisistaan. Jokaisella konstruktorilla on oma sivu, ja sivulla kunkin ohjelmointikielen kohdalla oma versionsa, sillä .NET Framework käsittää useita ohjelmointikieliä. Me olemme luonnollisesti tässä vaiheessa kiinnostuneita vain C#-kielisistä versioista.

Kunkin konstruktorin kohdalla on lyhyesti kerrottu mitä se tekee, ja sen jälkeen minkä tyyppisiä ja montako parametria konstruktori ottaa vastaan. Kaikista konstruktoreista saa lisätietoa klikkaamalla konstruktorin esittelyriviä. Esimerkiksi linkki

```
[C#] public String(char[]);
```

Vie sivulle (http://msdn.microsoft.com/en-us/library/ttyxaek9.aspx) jossa konstruktorista `public String(char[])` kerrotaan lisätietoja ja annetaan käyttöesimerkkejä.

Huomaa, että monet `String`-luokan konstruktoreista on merkitty `unsafe`-merkinnällä, jolloin niitä ei tulisi käyttää omassa koodissa. Tällaiset konstruktorit on tarkoitettu ainoastaan järjestelmien keskinäiseen viestintään.

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 lisää luvussa 15.

`String`-luokan olio on C#:n ehkä yleisin olio, ja on itse asiassa kokoelma (taulukko) perättäisiä yksittäisiä `char`-tyyppisiä merkkejä. Se voidaan luoda seuraavasti.

```
String nimi = new string(new char [] {'J', 'a', 'n', 'n', 'e'});
```

```
System.Console.WriteLine(nimi); // Tulostaa Janne
```

Näin kirjoittaminen on tietenkin usein melko vaivalloista. `String`-luokan olio voidaan kuitenkin poikkeuksellisesti luoda myös alkeistietotyyppisten muuttujien määrittelyä muistuttavalla tavalla. Alla oleva oleva lause on vastaava kuin edellisessä kohdassa, mutta lyhyempi kirjoittaa.

```
String nimi = "Janne";
```

Huomaa, että merkkijonon ympärille tulee lainausmerkit. Näppäimistöltä lainausmerkit saadaan näppäinyhdistelmällä `Shift+2`. Vastaavasti merkkijono voitaisiin kuitenkin alustaa myös muilla `String`-luokan konstruktoreilla, joita on pitkä lista.

Jos taas tutkimme `PhysicsObject`-luokan dokumentaatiota (löytyy osoitteesta http://kurssit.it.jyu.fi/npo/material/latest/documentation/html/ → Luokat → PhysicsObject), löydämme useita eri konstruktoria (ks. kohta *Julkiset jäsenfunktiot*, jotka alkavat sanalla `PhysicsObject`). Konstruktoreista järjestyksessä toinen saa parametrina kaksi lukua ja muodon. Tätä konstruktoria käytimme jo lumiukkoesimerkissä.

Voisimme kuitenkin olla antamatta muotoa (ensimmäinen konstruktori) ja määritellä muodon vasta myöhemmin fysiikkaolion `Shape`-ominaisuuden avulla.

Home | Library | Learn | Downloads | Support | Sign in | Suomi - Suomi

Search MSDN with Bing

- MSDN Library
- .NET Development
- .NET Framework 4
- .NET Framework Class Library
- System
- String Class
  - String Constructor
    - String Constructor (Char*)
    - **String Constructor (Char[])**
    - String Constructor (SByte*)
    - String Constructor (Char, Int32)
    - String Constructor (Char*, Int32, Int32)
    - String Constructor (Char[], Int32, Int32)
    - String Constructor (SByte*, Int32, Int32)
    - String Constructor (SByte*, Int32, Int32, En

Community Content

Add code samples and tips to enhance this topic.

More...

String Constructor (Char[])

msdn

.NET Framework 4 | Other Versions

Initializes a new instance of the String class to the value indicated by an array of Unicode characters.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

VB | C# | C++ | F# | JScript

```
public String(
        char[] value
)
```

Parameters

value
Type: System.Char[]
An array of Unicode characters.

Kuva 10: Tiedot luokan konstruktoreista löytyvät MSDN-dokumentaatioissa Constructor-kohdasta.

Julkiset jäsenfunktiot

	PhysicsObject (double width, double height) Luo uuden fysiikkaolion.
	PhysicsObject (double width, double height, **Shape** shape) Luo uuden fysiikkaolion.
	PhysicsObject (**Image** image) Luo uuden fysiikkaolion. Kappaleen koko ja ulkonäkö ladataan parametrina annetusta kuvasta.
	PhysicsObject (double width, double height, **Shape** shape, **CollisionShapeQuality** quality) Luo uuden fysiikkaolion asettaen laadun törmäyskappaleelle. Käytä tätä rakentajaa vain, jos törmäystunnistuksen laatu ei ole tyydyttävää.
	PhysicsObject (double width, double height, **Shape** shape, double maxDistanceBetweenVertices, double gridSpacing) Luo uuden fysiikkaolion antaen parametreja fysiikan laskentaan. Käytä tätä rakentajaa vain, jos on tarvetta kokeilla eri parametrien vaikutusta törmäyksen laatuun.
	PhysicsObject (**RaySegment** raySegment) Luo fysiikkaolion, jonka muotona on säde.

Kuva 11: Jypeli-kirjaston luokan konstruktorit löytyvät Julkiset jäsenfunktiot -otsikon alta.

8.7.2 Harjoitus

Tutki muita konstruktoreja. Mitä niistä selviää dokumentaation perusteella? Mikä on oletusmuoto?

8.7.3 Metodit

Kohta Methods (http://msdn.microsoft.com/en-us/library/system.string_methods.aspx) sisältää tiedot kaikista luokan metodeista. Jokaisella metodilla on taulukossa oma rivi, ja rivillä lyhyt kuvaus, mitä metodi tekee. Klikattuasi jotain metodia saat siitä tarkemmat tiedot. Tällä sivulla kerrotaan mm. minkä tyyppisen parametrin metodi ottaa, ja minkä tyyppisen arvon metodi palauttaa. Esimerkiksi `String`-luokassa käyttämämme `ToUpper`-metodi, joka siis palauttaa `String`-tyyppisen arvon.

8.7.4 Huomautus: Luokkien dokumentaatioiden googlettaminen

Huomaa, että kun haet luokkien dokumentaatioita hakukoneilla, saattavat tulokset viitata .NET Frameworkin vanhempiin versioihin (esimerkiksi 1.0 tai 2.0). Kirjoitushetkellä uusin .NET Framework-versio on 4, ja onkin syytä varmistua että löytämäsi dokumentaatio koskee juuri oikeaa versiota. Voit esimerkiksi käyttää hakutermissä versionumeroa tähän tapaan: ”`c# string documentation .net 4`”. Versionumeron näkee otsikon alapuolella. Voit halutessasi vaihtaa johonkin toiseen versioon klikkaamalla Other Versions -pudotusvalikkoa.

8.8 Tyyppimuunnokset

C#:ssa yhteen muuttujaan voi tallentaa vain yhtä tyyppiä. Tämän takia meidän täytyy joskus muuttaa esimerkiksi `String`-tyyppinen muuttuja `int`-tyyppiseksi tai `double`-tyyppinen muuttuja `int`-tyyppiseksi ja niin edelleen. Kun muuttujan tyyppi vaihdetaan toiseksi, sanotaan sitä tyyppimuunnokseksi (**cast**).

Kaikilla alkeistietotyypeillä sekä C#:n oliotyypeillä on `ToString`-metodi, jolla olio voidaan muuttaa merkkijonoksi. Alla esimerkki `int`-luvun muuttamisesta merkkijonoksi.

```
// kokonaisluku merkkijonoksi
int kokonaisluku = 24;
String intMerkkijonona = kokonaisluku.ToString();

// liukuluku merkkijonoksi
double liukuluku = 0.562;
String doubleMerkkijonona = liukuluku.ToString();
```

Merkkijonon muuttaminen alkeistietotyypiksi onnistuu sen sijaan jokaiselle alkeistietotyypille tehdystä luokasta löytyvällä metodilla. Alkeistietotyypithän eivät ole olioita, joten niillä ei ole metodeita. C#:sta löytyy kuitenkin jokaista alkeistietotyyppiä vastaava *rakenne* (**structure**), joista löytyy alkeistietotyyppien käsittelyyn hyödyllisiä metodeita. Rakenteet sijaitsevat `System`-nimiavaruudessa, ja tästä syystä ohjelman alussa tarvitaan lause

```
using System;
```

Alkeistietotyyppejä vastaavat rakenteet löytyy seuraavasta taulukosta.

Taulukko 5: Alkeistietotyypit ja niitä vastaavat rakenteet.

Alkeistieto-tyyppi	Rakenne
`bool`	`Boolean`
`byte`	`Byte`
`char`	`Char`
`short`	`Int16`
`int`	`Int32`
`long`	`Int64`
`ulong`	`UInt64`
`float`	`Single`
`double`	`Double`

Huomaa, että rakenteen ja alkeistietotyypin nimet ovat C#:ssa synonyymejä. Seuraavat rivit tuottavat saman lopputuloksen (mikäli `System`-nimiavaruus on otettu käyttöön `using`-lauseella).

```
int luku1 = 5;
Int32 luku2 = 5;
```

Vastaavasti kaikki rakenteiden metodit ovat käytössä, kirjoittipa alkeistietotyypin tai rakenteen nimen. Tästä esimerkki seuraavaksi.

Merkkijonon (`String`) muuttaminen `int`-tyypiksi onnistuu C#:n `int.Parse`-funktiolla seuraavasti.

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

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

Jos luvun parsiminen (jäsentäminen, muuttaminen) ei onnistu, aiheuttaa se niin sanotun *poikkeuksen*. `double`-luvun parsiminen onnistuu vastaavasti `Double`-rakenteesta (iso D-kirjain) löytyvällä `Parse`-funktiolla.

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

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 ”kysyjälle” paluuarvona. Tällöin aliohjelmaa voidaan käyttää myös tilanteessa, jossa keskiarvoa ei haluta tulostaa, vaan sitä tarvitaan johonkin muuhun laskentaan. Paluuarvon palauttaminen tapahtuu `return`-lauseella, ja `return`-lause lopettaa aina aliohjelman suorittamisen.

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

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

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 `keskiarvo`. Kolmannella rivillä lasketaan parametrien `a` ja `b` summa ja jaetaan se kahdella muuttujaan `keskiarvo`. Neljännellä rivillä palautetaan keskiarvo-muuttujan arvo.

Aliohjelmaa voitaisiin nyt käyttää pääohjelmassa esimerkiksi alla olevalla tavalla.

```
double keskiarvo;
keskiarvo = Keskiarvo(3, 4);
Console.WriteLine("Keskiarvo = " + keskiarvo);
```

Tai lyhyemmin kirjoitettuna:

```
Console.WriteLine("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 keskiarvo = (a + b) / 2.0;
    return keskiarvo;
}
```

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 viimeisin tapa on selkein ja lyhin.

Aliohjelmassa voi olla myös useita `return`-lauseita. Tästä esimerkki kohdassa: 13.5.1.

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. C#:ssa on olemassa kolmaskin keino on olemassa, jota ei käsitellä tällä kurssilla: `ref`- ja `out`-parametrit, katso

http://msdn.microsoft.com/en-us/library/t3c3bfhx(v=vs.80).aspx.

Metodeita ja aliohjelmia, jotka ottavat vastaan parametreja palauttavat arvon 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
Console.WriteLine(tulos); //tulostaa 3.5
Console.WriteLine(tulos); //tulostaa 3.5
```

ja tämän

```
Console.WriteLine(Keskiarvo(5, 2)); //tämäkin tulostaa 3.5
Console.WriteLine(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.

9.1 Harjoitus

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. Visual Studio 2010

Vaikka ohjelmakoodia voi kirjoittaa pelkällä editorillakin, ohjelmien koon kasvaessa alkaa kaipaamaan työvälineiltä hieman enemmän ominaisuuksia. Peruseditoreja enemmän ominaisuuksia tarjoavat sovelluskehittimet eli IDE:t (**Integrated Development Environment**). C#:lle tehtyjä ilmaisia sovelluskehittimiä ovat muun muassa

- Visual Studio 2010 Express -tuoteperhe,
- Mono ja
- SnippetCompiler.

Tässä monisteessa tutustumme tarkemmin *Visual Studio 2010 Express* -tuoteperheeseen kuuluvaan *Visual C# 2010 Express* -kehittimeen. Muita perheeseen kuuluvia ilmaisia C#-kehittimiä ovat mm. *Visual Studio 2010 Express for Windows Phone* sekä *Visual Web Developer 2010 Express*

Kaikki ohjeet on testattu toimiviksi Visual C# 2010 Express sekä Visual Studio 2010 Ultimate -versioilla, luonnollisesti Windows-ympäristössä.

10.1 Asennus

Visual Studion (VS) maksullinen – ja Jyväskylän yliopiston opiskelijoille maksuton – Ultimate-versio, on ladattavissa MSDNAA-palvelusta. Edellä mainittu Express-versio on saatavilla ilmaiseksi Microsoftin verkkosivuilta. Express-versio riittää muilta osin tämän kurssin tarpeisiin, mutta siitä puuttuvat testausominaisuudet (ks. luku 11). Muistathan, että VS on saatavissa vain Windows-käyttöjärjestelmille. Linux- ja Mac OS X -ympäristöihin on saatavilla ainakin Mono-sovelluskehitin, mutta tällä kurssilla käytettävä Jypeli-ohjelmointikirjasto vaatii toimiakseen Windows-käyttöjärjestelmän.

10.2 Käyttö

10.2.1 Ensimmäinen käyttökerta

Kun käynnistät VS:n ensimmäisen kerran, kysyy VS millä oletusasetuksilla haluat valikoita ja toimintoja käyttää. Valitse Visual C# Development Settings.

Aluksi rivinumerot eivät ole käytössä. Koodin seuraamisen helpottamiseksi otetaan ne käyttöön kohdasta Tools → Options → Text Editor → C# → Line Numbers. Jos Error list-ikkuna ei ole jo näkyvissä ruudun alareunassa, kannattaa se ottaa käyttöön: View → Error List.

10.2.2 Projektit ja solutionit

Visual Studiossa on *solutioneja* ja *projekteja*. Yhdessä solutionissa voi olla monta projektia ja jokaisen projektin täytyy kuulua jonkin solutioniin. Uuden ohjelman kirjoittaminen alkaa aina projektin perustamisella johonkin solutioniin, jolloin myös solution luodaan samalla, ellei sitä ole jo tehty.

Hierarkia ei ole kiveen hakattu, mutta solutioneja ja projekteja voi hahmottaa esimerkiksi seuraavasti:

- Jokainen demokerta (tai luento, ohjauskerta tms) on uusi solution, eli kun aloitat tekemään *ensimmäistä* demotehtävää (ohjaustehtävää) niin toimitaan seuraavasti
    - Klikkaa File → New project (**Ctrl+Shift+N**)
    - Valitse projektin tyyppi: Jos teet konsoliohjelmaa niin valitse Visual C# -kohdasta Console Application. Jos teet Jypeli-peliä, valitse Visual C# → Jypeli-kohdasta

haluamasi projektimalli (esimerkiksi `FysiikkaPeli`).
    - Anna projektille (eli tehtävälle) nimi kohtaan Name, esimerkiksi `Lumiukko` tai `HelloWorld`. Tarkista demotehtävien nimeämiskäytäntö opettajalta.
    - Laita Location-kohtaan demotehtäviesi juuripolku, eli vaikkapa `C:\MyTemp\<omatunnus>\ohj1\demot`. Laita `omatunnus` kohtaan luonnollisesti yliopiston mikroverkon käyttäjätunnuksesi. Älä käytä ääkkösiä tai välilyöntejä kansioiden eikä tiedostojen nimissä, tulemme yleensä toimeen myös ilman niitä, mutta ongelmia ääkköset ja erikoismerkit aiheuttavat usein.

> Huomaa! Tarkista luennoitsijalta demotehtävien tallennuspaikka yliopiston mikroverkossa.

    - Valitse Solution-valikosta Create new Solution ja paina Create directory for solution (tärkeä)
    - Laita Solution Name kohtaan `demoN` (N = demokerran numero), esimerkiksi `demo3`
    - Nyt sinulle on luotu uusi solution ja yksi projekti solutionin sisään.
- Kun haluat *lisätä* demotehtäviä (projekteja) tiettyyn demokertaan (solutioniin) toimi näin
    - Avaa sen demokerran solution (`.sln`-tiedosto) johon haluat tehtävän lisätä (ellei se ole jo auki Visual Studiossa).
    - File → New project *tai* Solution Explorerissa klikkaa hiiren oikealla solutionin päälle, ja klikkaa Add → New project
    - Anna projektille nimi, jätä Location ennalleen, ja valitse Solution-valikosta Add to Solution. Solution Name -kohta himmenee ja Location-kohtaan pitäisi tulla automaattisesti lisäteksti `demoN`.

Näin tehtynä kaikki yhden demokerran tehtävät löytyvät "saman katon alta" eli yhden solutionin kaikki projektit menevät samaan kansioon. Resurssienhallinnassa hakemistopuusi voisi näyttää esimerkiksi tältä.

```
ohj1
 |
 +--demot
 |  +--demo1
 |  |  +--HelloWorld
 |  |  +--Lumiukko
 |  |  '-demo1.sln
 |  '--demo2
 |     +--Lumiukko2
 |     '--LukujenLaskemista
 '--ohjaukset
    +--ohjaus1
    |  +--HelloWorld
    |  '--Lumiukko
    '--ohjaus2
```

Yksittäisten projektien kansioihin (tässä `HelloWorld`, `Lumiukko`, `Lumiukko2`, ja niin edelleen) syntyy myös läjä erilaisia tiedostoja. Tutkitaan vielä, millainen rakenne yksittäisen projektin kansioon on syntynyt. Esimerkkinä vaikkapa `HelloWorld`-kansio.

```
HelloWorld
 |
 +--bin
 |  +--Debug
 |     +-HelloWorld.vshost.exe
 |     +-HelloWorld.vshost.exe.manifest
 +--obj
 |  +--x86
 |     +-(...)
 +--Properties
 |  +-AssemblyInfo.cs
```

```
+-HelloWorld.csproj
'-Program.cs
```

Lähdekoodi sijaitsee `Program.cs`-tiedostossa. Kuten huomaat, projektiin kuuluu kuitenkin monia muitakin tiedostoja ja kansioita. `bin`- ja `obj`-kansiot sisältävät ohjelman käännökseen liittyviä väliaikaistiedostoja. `Properties`-kansio taas sisältää julkaisuun liittyvää tietoa esimerkiksi tekijänoikeuksista ja merkistöstä. Huomaa, että varsinainen projektitiedosto on `.csproj`-päätteinen.

Solution toimii tavallaan liimana näiden eri projektitiedostojen välille yhdistäen ne Visual Studiossa. Tämän johdosta monen eri projektin käsittely yhtä aikaa on kätevää.

Viimeisenä tutkitaan Jypeli-projektimallista luotua projektia, sillä sen kansiorakenne eroaa hieman konsoliohjelmaan syntyvästä kansiorakenteesta, esimerkkinä `Lumiukko`-projekti.

```
Lumiukko
 |
 +--bin
 |  +-(...)
 +--obj
 |  +-(...)
 +--Properties
 |  +-AssemblyInfo.cs
 +-Game.ico
 +-GameThumbnail.png
 +-Lumiukko.csproj
 +-Ohjelma.cs
 '-Peli.cs
LumiukkoContent
 |
 +--bin
 |  +-(...)
 +--obj
 |  +-(...)
 '-LumiukkoContent.contentproj
```

Projekti sisältää yhden kansion sijaan kaksi kansiota (`Lumiukko` ja `LumiukkoContent`), joista jälkimmäinen on varattu sisällön, kuten äänien ja kuvien, tuomiselle peliin. `Lumiukko`-kansiosta löytyy edelleen kaksi kooditiedostoa, `Ohjelma.cs` ja `Peli.cs`, joista ensimmäinen on varattu `Main`-metodille, ja jälkimmäinen sisältää varsinaisen pelin lähdekoodin. `Ohjelma.cs`-tiedostoa ei käytännössä juurikaan tarvitse muokata.

`Peli.cs` sisältää lähdekoodin, ja se on yleensä se tiedosto, joka pyydetään palautettavaksi demotehtävissä. Muita tiedostoja ei tarvitse palauttaa, ellei niitä erikseen pyydetä. Huomaa, että tiedoston nimen voi ja tulee muuttaa vastaamaan luokan nimeä.

10.2.3 Visual Studion perusnäkymä

Kun olet luonut ensimmäisen projektisi, pitäisi edessäsi olla VS:n perusnäkymä. VS voi vaikuttaa aluksi melko monimutkaiselta, mutta sen peruskäytön oppii nopeasti. Jos jotakin työkalua ei tarvi ja se vie mielestäsi tilaa ruudulla, sen voi yleensä piilottaa. Kannattaakin opetella käyttämään ruututilaa tehokkaasti ja poistaa turhat visuaaliset elementit käytöstä kun niitä ei tarvita.

Oikealla on Solution Explorer, jossa näkyy kaikki projektit, sekä projektien sisältämät tiedostot ja viitteet muihin kooditiedostoihin.

Alhaalla on Error List (mikäli aktivoit sen päälle aikaisemmin), joka näyttää koodin kirjoittamisen aikana havaittuja virheitä, sekä käännöksessä tapahtuvia virheitä. Error List on yksi ohjelmoijan tärkeimmistä työkaluista ja Visual Studion ehdoton vahvuus – syntaksivirheet saadaan kiinni ilman, että ohjelmoijan tarvitsee erikseen edes kääntää tekemäänsä ohjelmaa. Voit tuplaklikata Error

Listissä virhettä, ja VS vie kursorin suoraan siihen pisteeseen, jossa olettaa virheen olevan.

Keskellä näkyy koodin kirjoitusikkuna. Eri kooditiedostot aukeavat välilehtiin, joita voi selata näppäinyhdistelmällä **Ctrl+Tab** (ja **Shift+Ctrl+Tab**).

10.2.4 Ohjelman kirjoittaminen

Jokainen C#-ohjelma kirjoitetaan luokan sisään. Luotaessa uusi konsoliohjelma, tekee Visual Studio meille valmiin kooditiedoston, jossa on pieni pätkä valmista ohjelmakoodia. Oletusarvoisesti konsoliohjelmaan luodaan yksi kooditiedoston nimeltä `Program.cs`, jossa on valmiina muutama `using`-lause, nimiavauus eli `namespace` (projektia vastaavalla nimellä), luokka sekä pääohjelma eli `Main`-metodi.

Aivan aluksi kannattaa muuttaa kooditiedoston nimi johonkin hieman kuvaavampaan klikkaamalla Solution Explorerissa tiedoston päällä hiiren oikealla napilla ja valitsemalla Rename. Kun tiedoston nimi on muutettu, kysyy Visual Studio, haluatko vyöryttää tekemäsi muutokset myös muualle ohjelmaan. Vastaa tähän kyllä.

Nyt meillä on edessä uusi luokan raakile `Main`-metodeineen ja editori, jolla ohjelma voidaan kirjoittaa.

10.2.5 Ohjelman kääntäminen ja ajaminen

Kun ohjelma on kirjoitettu, sen ajaminen onnistuu ylhäältä Debug-napista (vihreä kolmio) tai klikkaamalla **F5** (debug). Nappia painamalla VS kääntää ohjelman automaattisesti ja suorittaa heti sen jälkeen ns. debuggaustilassa. Ohjelma voidaan ajaa myös nopeammin ilman debuggausta – mikä ei yleensä ole suositeltavaa – painamalla **Ctrl+F5** (start without debugging). Jos haluamme lopettaa ohjelman suorituksen jostain syystä kesken kaiken, onnistuu se painamalla **Shift+F5**.

10.2.6 Referenssien asettaminen

Oman tai jonkun muun tekemän luokkakirjaston (`.dll`-tiedosto), esimerkiksi kurssilla käytettävä Jypeli voi lisätä VS:oon seuraavasti. Valitse Solution Explorerista References ja klikkaa hiiren oikealla napilla Add reference. Mene Browse-välilehdelle ja hae haluamasi kirjasto, esimerkiksi `Jypeli4.dll`. Nyt voit kirjoittaa kooditiedoston alkuun lauseen

```
using Jypeli;
```

jolloin Jypeli-kirjasto on käytössäsi.

10.3 Debuggaus

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 Wikipediasta:

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

Virheet ohjelmakoodissa ovat tosiasia. Ne tulevat aina olemaan mukanasi, vaikka olisit kuinka hyvä ohjelmoija tahansa. Hyvä ohjelmoija kuitenkin tiedostaa tämän asian, ja valmistautuu siihen hankkimalla työkalut virheiden jäljittämistä ja korjaamista varten. On olemassa pieniä virheitä, jotka eivät vaikuta ohjelman toimintaan millään tavalla (kuten kirjoitusvirhe painonapissa), mutta on olemassa myös virheitä, jotka ovat erittäin vakavia. Tällaiset virheet kaatavat ohjelman. *Syntaksivirheet* voivat olla pieniä, mutta estävät ohjelmaa kääntymästä. *Loogiset virheet* eivät jää

kääntäjän kouriin, mutta aiheuttavat ongelmia ohjelman ajon aikana.

Ehkäpä ohjelmasi ei onnistu lisäämään oikeaa tietoa tietokantaan, koska tarvittava kenttä puuttuu, tai lisää väärän tiedon jossain olosuhteissa. Tällaiset virheet, jossa sovelluksen logiikka on jollain tavalla pielessä, ovat semanttisia virheitä tai loogisia virheitä.

Varsinkin monimutkaisemmista ohjelmista loogisen virheen löytäminen on välillä vaikeaa, koska ohjelma ei kenties millään tavalla ilmoita virheestä – huomaat vain lopputuloksesta virheen kuitenkin tapahtuneen.

Tähän erinomaisena apuna on VS:n virheidenjäljitystoiminto eli *debuggaus*. 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.

VS:ssa 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. Aseta keskeytyskohta kursorin kohdalle painamalla **F9** tai klikkaamalla koodi-ikkunassa rivinumeroiden vasemmalle puolelle (harmaalle alueelle). Keskeytyskohta pitäisi näkyä punaisena pallona ja rivillä oleva lause värjättynä punaisella.

Kun keskeytyskohta on asetettu, klikataan ylhäältä Debug-nappia tai painetaan **F5**. Visual Studiossa ohjelman ”käynnistäminen” ja debuggauksen käynnistäminen ovat sama asia. Toki ohjelma voidaan käynnistää ilman debug-tilaakin, mutta silloin mahdolliset ohjelman virheet eivät tule Visual Studion tietoon, ja esimerkiksi nollalla jakaminen kaataa ajetun ohjelman.

Ohjelman suoritus on nyt pysähtynyt siihen kohtaan mihin asetimme keskeytyskohdan. Avaa Locals-välilehti alhaalta, ellei se ole jo auki. Debuggaus-näkymässä Locals-paneelissa näkyy kaikki tällä hetkellä näkyvillä olevat muuttujat (paikalliset, eli lokaalit muuttujat) ja niiden arvot. Keskellä näkyy ohjelman koodi, ja keltaisella se rivi missä kohtaa ohjelmaa ollaan suorittamassa. Vasemalla näkyy myös keltainen nuoli joka osoittaa sen hetkisen rivinumeron.

Ohjelman suoritukseen rivi riviltä on nyt kaksi eri komentoa: Step Into (**F11**) ja Step Over (**F10**). 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 Debug → Continue (**F5**)-napilla tai keskeyttää ohjelman suorituksen Terminate (**Shift+F5**)-napilla.

10.4 Hyödyllisiä ominaisuuksia

10.4.1 Syntaksivirheiden etsintä

Visual Studio huomaa osan syntaksivirheistä, joten osa virheistä voidaan korjata jo ennen kääntämistä – tarkemmin sanottuna VS kääntää jatkuvasti koodia havaitakseen ja ilmoittaakseen mahdolliset virheet. Kun VS löytää virheen, ilmestyy Error List -paneeliin virheilmoitus ja tieto siitä rivistä, jolla virheellinen koodi sijaitsee. Lisäksi jos virhe paikallistuu alleviivaa VS virheellisen koodinpätkän punaisella aaltoviivalla. Viemällä hiiri punaisen rastin päälle, VS kertoo

tarkemmin mikä kyseisessä kohdassa on vikana. Huomaa, että VS ei välttämättä paikallista virhettä täysin oikein. Usein virhe voi olla myös edellisellä tai seuraavalla rivillä.

10.4.2 Kooditäydennys, IntelliSense

IntelliSense on yksi VS:n parhaista ominaisuuksista. IntelliSense on monipuolinen automaattinen koodin täydentäjä, sekä dokumentaatiotulkki.

Yksi dokumentaatioon perustuva IntelliSensen ominaisuus on parametrilistojen selaus kuormitetuissa aliohjelmissa. Kirjoitetaan esimerkiksi

```
String nimi = "Kalle";
```

Kun tämän jälkeen kirjoitetaan `nimi` ja piste ”.”, ilmestyy lista niistä funktioaliohjelmista ja metodeista, jotka kyseisellä oliolla ovat käytössä. Aliohjelman valinnan jälkeen klikkaa kaarisulku auki, jolloin pienten nuolten avulla voi selata kyseessä olevan aliohjelman eri ”versioita”, eli samannimisiä aliohjelmia eri parametrimäärillä varustettuna. Lisäksi saadaan lyhyt kuvaus metodin toiminnasta ja jopa esimerkkejä käytöstä.

IntelliSense auttaa myös kirjoittamaan nopeammin ja erityisesti ehkäisemään kirjoitusvirheiden syntymistä. Jos ei ole konekirjoituksen Kimi Räikkönen, voi koodia kirjoittaessa helpottaa elämää painamalla **Ctrl+Space**. Tällöin VS yrittää arvata (perustuen kirjoittamaasi tekstiin sekä aiemmin kirjoittamaasi koodiin), mitä haluat kirjoittaa. Jos mahdollisia vaihtoehtoja on monta, näyttää VS vaihtoehdot listana.

10.4.3 Uudelleenmuotoilu

Visual Studio pyrkii muotoilemaan koodia "kauniiksi" kirjoittamisen yhteydessä. Käyttäjä voi kuitenkin tarkoituksellisesti tai vahingossa rikkoa VS:n sisäisiä muotoilusäännöksiä. Tällöin koodi voidaan palauttaa vastaamaan säännöksiä painamalla Edit → Advanced → Format Document tai näppäinyhdistelmällä **Ctrl+E, D** (pidä **Ctrl** pohjassa, klikkaa ensin **E** ja sitten **D**).

Jos välttämättä haluat, voit muokata VS:n koodimuotoilun oletussääntöjä valikosta:

Tools → Options → Text editor → C# → Formatting.

10.5 Lisätietoja Visual Studion käytöstä

Kurssin Wikistä löytyy lisää vinkkejä ja neuvoja Visual Studion käyttöön:

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

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

Jo melko yksinkertaisten ohjelmien testaaminen ruudulle 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 aliohjelmalle 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.

Visual Studion Pro-, Premium- ja Ultimate-versioissa on olemassa monipuoliset testausominaisuudet, mikä mahdollistaa ns. yksikkötestien (**unit tests**) kirjoittamisen. Ongelmana on, että näiden VS:n tukemien testitiedostojen kirjoittaminen on melko työlästä. Tähän on apuna Jyväskylän yliopiston tietotekniikan laitoksella kehitetty ComTest-työkalu, joka hyödyntää Visual Studion sisäänrakennettua testausjärjestelmää, mutta madaltaa huomattavasti kynnystä kirjoittaa ohjelmaansa yksikkötestejä. ComTest-testaustyökalun idea on, että testit voidaan kirjoittaa yksinkertaisella syntaksilla aliohjelmien ja metodien dokumentaatiokommentteihin suoraan ohjelman kooditiedostoon, joista sitten luodaan varsinaiset Visual Studio -testiprojektit ja -tiedostot. Samalla kirjoitetut testit toimivat dokumentaatiossa esimerkkinä aliohjelman tai metodin toiminnasta. ComTest:n asennusohjeet löytyy sivulta:

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

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

11.1 Käyttö

ComTestistä johtuen luokan, jossa aliohjelmia halutaan testata, on oltava julkisuusmääreellä `public`, muutoin testaaminen ei onnistu. Samoin jokaisen testattavan aliohjelman on oltava `public`-aliohjelma.

Kirjoita aliohjelmaan dokumentaatiokommentit ja kommentoi aliohjelma. Siirry dokumentaatiokommentin alaosaan, laita yksi tyhjä rivi (ilman kauttaviivoja) ja kirjoita ”comt” ja painaa **Tab+Tab** (kaksi kertaa sarkan-näppäintä). Tällöin Visual Studio luo valmiiksi paikan, johon testit kirjoitetaan. Dokumentaatiokommentteihin pitäisi ilmestyä seuraavat rivi.

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

Testit kirjoitetaan `pre`-tagien sisälle. Ylläoleva syntaksi on Doxygen-tyokalua (ja muita automaattisia dokumentointityökaluja) varten.

Aliohjelmat ja metodit testataan yksinkertaisesti antamalla niille parametreja ja kirjoittamalla mitä niiden odotetaan palauttavan annetuilla parametreilla. ComTest-testeissä käytetään erityistä vertailuoperaattoria, jossa on kolme yhtä suuri kuin -merkkiä (`===`). Tämä tarkoittaa, että arvon pitää olla sekä samaa tyyppiä, että samansisältöinen. Kirjoitetaan `Yhdista`-aliohjelma, joka yhdistää kahden annetun ei-negatiivisen luvun numerot toisiinsa. Tehdään aliohjelmalle samalla testit. Aliohjelman toteutuksessa on (naiivisti) oletettu, että parametrina annettavat luvut täyttävät varmasti annetun ehdon (ei-negatiivinen). Myöhemmin opimme käsittelemään myös sellaiset tilanteet, joissa tästä ehdosta ollaan poikettu.

Huomaa, että ComTest-testeihin kirjoitetuissa aliohjelmakutsuissa luokan nimi täytyy antaa ennen aliohjelman nimeä. Tässä luokan nimeksi on laitettu `Laskuja`.

```
/// <summary>
/// Yhdistää kahden annetun ei-negatiivisen
/// luvun numerot toisiinsa.
/// </summary>
/// <param name="a">Ensimmäinen luku</param>
/// <param name="b">Toinen luku</param>
/// <returns>Yhdistetty luku</returns>
/// <example>
/// <pre name="test">
/// Laskuja.Yhdista(0, 0) === 0;
/// Laskuja.Yhdista(1, 0) === 10;
/// Laskuja.Yhdista(0, 1) === 1;
/// Laskuja.Yhdista(1, 1) === 11;
/// Laskuja.Yhdista(13, 2) === 132;
/// Laskuja.Yhdista(10, 0) === 100;
/// </pre>
/// </example>
public static int Yhdista(int a, int b)
{
    String ab = a.ToString() + b;
    int tulos = int.Parse(ab);
    return tulos;
}
```

Tarkastellaan testejä nyt hieman tarkemmin.

```
Laskuja.Yhdista(0, 0) === 0;
```

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

```
Laskuja.Yhdista(1, 0) === 10;
```

Seuraavaksi testataan, että jos parametreistä ensimmäinen on luku 1 ja toinen luku 0, niin näiden yhdistelmä on 10, joten aliohjelman tulee palauttaa luku 10. Nollan ja ykkösen yhdistelmä antaisi 01, mutta sitä vastaava luku on tietenkin 1, joten se on luku jota palautuksena odotamme, ja näin jatketaan.

Varsinaisen VS-testin voi nyt luoda ja ajaa painamalla **Ctrl+Shift+Q** tai valikosta Tools → ComTest. Jos *Test Results* -välilehti (oletuksena näytön alareunassa, ilmestyy ajettaessa ComTest) näyttää vihreää ja lukee *Passed*, testit menivät oikein. Punaisen ympyrän tapauksessa testit menivät joko väärin, tai sitten testitiedostossa on virheitä.

Myös testit täytyy testata. Voihan olla, että kirjoittamissamme testeissä on myös virheitä. Tämä onkin syytä testata kirjoittamalla testeihin virhe tarkoituksella. Tällöin Test Results -välilehdellä pitäisi tietenkin näkyä punainen ympyrä. Jos näin ei ole, on joku testeistä väärin, tai aliohjelmassa on virhe. 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 `Yhdista`-aliohjelmassa ääritilanteita ovat lähinnä nollat, kummallakin puolella erikseen ja yhdessä. Muutoin testiarvot on valittu melko sattumanvaraisesti.

11.2 Liukulukujen testaaminen

Liukulukuja (`double` ja `float`) testataan ComTest:n vertailuoperaattorilla, jossa on kolme aaltoviivaa (~~~). Tämä johtuu siitä, että kaikkia reaalilukuja ei pystytä esittämään tietokoneella

tarkasti, joten täytyy sallia pieni virhemarginaali. Tehdään `Keskiarvo`-aliohjelma, joka osaa laskea kahden `double`-tyyppisen luvun keskiarvon ja kirjoitetaan sille samalla dokumentaatiokommentit ja ComTest-testit.

```
/// <summary>
/// Aliohjelma laskee parametrinaan saamiensa kahden
/// double-tyyppisen luvun keskiarvon.
/// </summary>
/// <param name="a">Ensimmäinen luku</param>
/// <param name="b">Toinen luku</param>
/// <returns>Lukujen keskiarvo</returns>
/// <example>
/// <pre name="test">
/// Laskuja.Keskiarvo(0.0, 0.0)   ~~~  0.0;
/// Laskuja.Keskiarvo(1.2, 0.0)   ~~~  0.6;
/// Laskuja.Keskiarvo(0.8, 0.2)   ~~~  0.5;
/// Laskuja.Keskiarvo(-0.1, 0.1)  ~~~  0.0;
/// Laskuja.Keskiarvo(-1.5, -2.5) ~~~ -2.0;
/// </pre>
/// </example>
public static double Keskiarvo(double a, double b)
{
    return (a + b) / 2.0;
}
```

Liukulukuja testattaessa täytyy parametrit antaa desimaaliosan kanssa. Esimerkiksi jos yllä olevassa esimerkissä ensimmäinen testi olisi muotoa `Keskiarvo(0, 0) ~~~ 0.0`, niin tällöin ajettaisiin aliohjelma `Keskiarvo(int x, int y)`, *jos* sellainen olisi olemassa.

12. Merkkijonot

Tässä luvussa tutustumme tarkemmin merkkijonoihin. Merkkijonot voidaan jakaa *muuttumattomiin* ja *muokattaviin*. C#:n muuttumaton merkkijono on tyypiltään `String`, johon olemmekin jo hieman tutustuneet olioiden yhteydessä. Muuttumatonta merkkijonoa ei voi muuttaa luomisen jälkeen. Muokattavan merkkijonon, `StringBuilder`-olion, käsittely on joissakin tilanteessa mielekkäämpää. Vaikka `String`-olioita ei voikaan muuttaa, pärjäämme sillä monissa tilanteissa.

> Huomaa, että `string` (pienellä alkukirjaimella kirjoitettuna) on `System.String`-luokan alias, joten `string` ja `String` voidaan samaistaa muuttujien tyyppimäärittelyssä, vaikka tarkasti ottaen toinen on alias ja toinen luokan nimi. Yksinkertaisuuden vuoksi jatkossa puhutaan pääsääntöisesti vain `String`-tyypistä sillä oletuksella, että System-nimiavaruus on otettu käyttöön lauseella `using System;`

12.1 Alustaminen

Merkkijono on *kokoelma peräkkäisiä merkkejä*. Tarkalleen ottaen merkkijono toteutetaan C#:ssa 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(new char[] {'J', 'a', 'n', 'n', 'e'});
String henkilo2 = "Kalle Korhonen";
```

Jälkimmäinen tapa muistuttaa enemmän alkeistietotyyppien alustamista, mutta merkkijonot ovat C#:ssa siis aina *olioita*.

12.2 Hyödyllisiä metodeja ja ominaisuuksia

`String`-luokassa on paljon hyödyllisiä metodeja, joista käsitellään nyt muutama. Kaikki metodit näet C#:n MSDN-dokumentaatiosta.

- `Equals(String)` Palauttaa tosi jos kaksi merkkijonoa ovat sisällöltään samat merkkikoko huomioon ottaen. Muutoin palauttaa epätosi.

```
if (etunimi.Equals("Aku")) Console.WriteLine("Löytyi!");
```

- `Compare(String, String, Boolean)` Vertaa merkkijonon aakkosjärjestystä toiseen merkkijonoon. Palauttaa arvon 0 jos merkkijonot ovat samat, nollaa pienemmän arvon jos ensimmäinen merkkijono on aakkosjärjestykseltään ennen kuin toinen ja nollaa suuremman arvon jos jälkimmäinen merkkijono on aakkosissa ennen ensimmäistä. Kirjainkoko saadaan merkitseväksi asettamalla kolmanneksi parametriksi `false`, tai jos halutaan unohtaa kirjainkoko, laitetaan kolmanneksi `true`.

```
String s1 = "jAnNe"; String s2 = "JANNE";
if (Compare(s1, s2, true) == 0) Console.WriteLine("Samat tai melkein samat!"); //
samat tai melkein samat
```

- `Contains(String)` Palauttaa totuusarvon sen perusteella esiintyykö parametrin sisältämän merkkijono tutkittavana olevassa merkkijonossa.

```
String henkilo = "Ville Virtanen";
String haettava = "irta";
if (henkilo.Contains(haettava)) Console.WriteLine(haettava + " löytyi!");
```

- `Substring(Int32)` Palauttaa osan merkkijonosta alkaen parametrinaan saamastaan indeksistä.

```
String henkilo = "Ville Virtanen";
String sukunimi = henkilo.Substring(6);
Console.WriteLine(sukunimi); // Virtanen
```

- `Substring(Int32, Int32)` Palauttaa osan merkkijonosta parametrinaan saamiensa indeksien välistä. Ensimmäinen parametri on palautettavan merkkijonon ensimmäisen merkin indeksi ja toinen parametri palautettavien merkkien määrä.

```
String henkilo = "Ville Virtanen";
String etunimi = henkilo.Substring(0, 5);
Console.WriteLine(etunimi); // Ville
```

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

```
String henkilo = "Ville Virtanen";
Console.WriteLine(henkilo.ToLower()); //tulostaa "ville virtanen"
```

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

```
String henkilo = "Ville Virtanen";
Console.WriteLine(henkilo.ToUpper()); //tulostaa "VILLE VIRTANEN"
```

- `Replace(Char, Char)` Korvaa merkkijonon kaikki tietyt merkit toisilla merkeillä. Ensimmäisenä parametrina korvattava merkki ja toisena korvaaja. Huomaa, että parametrit laitetaan `char`-muuttujille tyypilliseen tapaan yksinkertaisten lainausmerkkien sisään.

```
String sana = "katti";
sana = sana.Replace('t', 's');
Console.WriteLine(sana);  //tulostaa "kassi"
```

- `Replace(String, String)` Korvaa merkkijonon kaikki merkkijonoesiintymät toisella merkkijonolla. Ensimmäisenä parametrina korvattava merkkijono ja toisena korvaaja.

```
String sana = "katti kattinen";
sana = sana.Replace("atti", "issa");
Console.WriteLine(sana);  //tulostaa "kissa kissanen"
```

Lisäksi

- `Length` Palauttaa merkkijonon pituuden kokonaislukuna. Huomaa, että tämä EI ole aliohjelma / metodi, vaan merkkijono-olion *ominaisuus*.

```
String henkilo = "Ville";
Console.WriteLine(henkilo.Length); //tulostaa 5
```

Koska merkkijono on kokoelma yksittäisiä `char`-merkkejä, saadaan merkkijonon kukin merkki `char`-tyyppisenä laittamalla halutun merkin paikkaindeksi merkkijono-olion perään hakasulkeiden

sisään, esimerkiksi:

```
String henkilo1 = "Seppo Sirkuttaja";
char kolmasKirjain;
int i = 2;
kolmasKirjain = henkilo1[i];
Console.WriteLine(henkilo1 + " -nimen paikassa " + i + " oleva merkki on " +
kolmasKirjain);
```

> Merkkijonojen indeksointi alkaa nollasta! Merkkijonon ensimmäinen merkki on siis indeksissä 0.

12.3 Muokattavat merkkijonot: StringBuilder

Niin sanottujen muuttumattomien merkkijonojen, eli `String`-tyypin, lisäksi C#:ssa 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. Käytännössä, jos haluamme muuttaa `String`-merkkijonoa, tehdään uusi olio. Jos merkkijonoon tehdään paljon muutoksia (esimerkiksi jonoon lisätään useaan kertaan merkkejä), käy käsittely lopulta hitaaksi – ja tämä hitaus alkaa näkyä melko nopeasti.

C#-kielessä muokattava merkkijonoluokka on `StringBuilder`, joka sijaitsee `System.Text`-nimiavaruudessa. Voit ottaa tuon nimiavaruuden käyttöön kirjoittamalla ohjelman alkuun

```
using System.Text;
```

Merkkijonon perään lisääminen onnistuu `Append`-metodilla. `Append`-metodilla voi lisätä merkkijonon perään muun muassa kaikkia C#:n alkeistietotyyppejä sekä `String`-olioita. Myös kaikkien C#:n 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(); // "" (tyhjä)
yhtalo.Append("f(x): "); // "f(x): "
yhtalo.Append(a);        // "f(x): 3.5"
yhtalo.Append(" + ");    // "f(x): 3.5 + "
yhtalo.Append(b);        // "f(x): 3.5 + 6"
yhtalo.Append('x');      // "f(x): 3.5 + 6x"
yhtalo.Append(" = ");    // "f(x): 3.5 + 6x = "
yhtalo.Append(c);        // "f(x): 3.5 + 6x = 9.5"
```

Tiettyyn paikkaan voidaan lisätä merkkejä ja merkkijonoja `Insert`-metodilla, joka saa parametrikseen indeksin, eli kohdan, johon merkki (tai merkit) lisätään, sekä lisättävän merkin (tai merkit). Indeksointi alkaa jälleen nollasta. `Insert`-metodilla voi lisätä myös kaikkia samoja tietotyyppejä kuin `Append`-metodillakin. Voisimme esimerkiksi lisätä edelliseen esimerkkiin luvun 3.5 perään vielä muuttujan `x`. Sitä ennen tarkastellaan merkkien järjestystä ja indeksointia ja kirjoitetaan kunkin tulostuvan merkin yläpuolelle sen paikkaindeksi.

```
01234567890123456789 0
|----+----|----+----|
f(x): 3,5 + 6x = 9,5
```

Tästä huomaamme, että indeksi, johon haluamme muuttujan `x` lisätä, on 9.

```
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, jolloin merkkijonon pituus kasvaa siis yhdellä. Korvaamiseen on olemassa oma metodi, `Replace`.

12.3.1 Muita StringBuilder-luokan hyödyllisiä metodeja

- `Remove(Int32, Int32)`. Poistaa merkkijonosta merkkejä siten, että ensimmäinen parametri on aloitusindeksi, ja toinen parametri on poistettavien merkkien määrä.
- `ToString()` ja `ToString(Int32, Int32)`. Palauttaa `StringBuilder`-olion sisällön ”tavallisena” `String`-merkkijonona. `ToString`-metodille voi antaa myös kaksi int-lukua parametreina, jolloin palautetaan osa merkkijonosta (ks. `Substring`).

Muut metodit löytyvät `StringBuilder`-luokan MSDN-dokumentaatiosta:

http://msdn.microsoft.com/en-us/library/k5314fdf.aspx.

12.4 Huomautus: aritmeettinen + vs. merkkijonoja yhdistelevä +

Merkkijonoihin voidaan ”+”-merkkiä käyttämällä yhdistellä myös 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
      Console.WriteLine(luku1 + luku2); //tulostaa 7

      //tässä "+"-merkki toimii merkkijonoja yhdistelevänä operaattorina
      Console.WriteLine(luku1 + "" + luku2);  //tulostaa 25

      //Tässä ensimmäinen "+"-merkki toimii aritmeettisena operaattorina
      //ja toinen "+"-merkki merkkijonoja yhdistelevänä operaattorina
      Console.WriteLine(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.

12.5 Vinkki: näppärä tyyppimuunnos String-tyypiksi

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

```
int luku = 23;
bool 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 `bool`-tyyppistä muuttujaa.

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

12.6 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 20 desimaalin tarkkuudella kirjoittamalla `Math.PI`. Huomaa, että `PI` ei ole metodi, joten perään ei tule sulkuja. `PI` on `Math`-luokan julkinen staattinen vakio (**public const double**). Jos haluaisimme muuttaa piin `String`-tyypiksi vain kahden desimaalin tarkkuudella, onnistuisi se seuraavasti:

```
String pii = String.Format("{0:#.##}", Math.PI); // pii = "3,14"
```

Tässä `Format`-metodi saa kaksi parametria. Ensimmäistä parametria sanotaan muotoilumerkkijonoksi (**format string**). Toisena parametrina on sitten muotoiltava arvo. Muotoilumahdollisuuksia on hyvin paljon. Alla muutamia esimerkkejä erilaisten lukujen muotoilusta. Lisää löydät MSDN-dokumentaatiosta kohdasta Formatting types.

Luku	Muotoilujonon määrittely			
	{0:000.0}	{0:###.#}	{0:##0.0}	{0:#0E+0}
1230,1	1230,1	1230,1	1230,1	12E+2
17	017,0	17	17,0	17E+0
0,15	000,2	,2	0,2	15E-2
0	000,0		0,0	00E+0
-26	-026,0	-26	-26,0	-26E+0

Kuva 12: Muotoilujonoilla voidaan muotoilla lukuja monipuolisesti. Tämän esimerkin lähdekoodi löytyy osoitteesta https://trac.cc.jyu.fi/projects/ohj1/browser/luentomonistecs/esimerkit/StringFormat.cs

Esimerkkisarakkeista kolmas, `{0:##0.0}`, on esimerkki siitä, miten erilaisia lukuja saadaan järjestettyä siististi myös päällekkäin desimaalipisteen (tai -pilkun) kohdalta. Risuaita tarkoittaa, että jos luvussa ei ole sen merkin kohdalla numeroa, se jätetään tyhjäksi. Nolla sen sijaan ”pakottaa” numeron sen merkin paikalle, vaikka syötteessä ei sen merkin kohdalla olisikaan mitään: esimerkiksi syötteet 17 ja 0 muuttuvat 17.0:ksi ja 0.0:ksi. Esimerkiksi rahaan liittyvissä sovelluksissa oletuksena desimaalipisteen jälkeen olisi mielekästä olla kaksi nollaa, jolloin ”nollasentit” näytetään joka tapauksessa, myös rahamäärän ollessa tasasumma.

Huomaa, että ylläolevissa esimerkeissä oletetaan, että järjestelmän desimaalierottimena on piste. Tämä on järjestelmäkohtaista ja muutettavissa esimerkiksi Windows 7:ssa Control panel → Region and language → Formats → Additional settings → Decimal symbol.

Muotoilumerkkijono laitetaan lainausmerkkeihin, ja aaltosulkujen sisään. Muotoilujonoja voi olla myös useita, samoin muotoiltavia merkkijonoja. Tästä esimerkki alla.

```
String luvut = String.Format("{0} {2} {1} {0}", 1, 2, 3);
Console.WriteLine(luvut); // Tulostaa: 1 3 2 1
```

Muotoillun jonon (ja sen määrittelyn) voi antaa myös suoraan esimerkiksi `WriteLine`-metodille. Alla edellinen esimerkki lyhyemmin kirjoitettuna.

```
Console.WriteLine("{0} {2} {1} {0}", 1, 2, 3);
```

Lisätietoja merkkijonojen muotoilusta löytyy MSDN-dokumentaatiosta:

http://msdn.microsoft.com/en-us/library/0c899ak8.aspx.

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 syötteestä tai aliohjelmien parametreista, tarvitsemme ehtolauseita.

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

Tavallinen ehtolause sisältää aina sanan ”jos”, ehdon sekä toimenpiteet mitä tehdään, jos ehto on tosi. Arkielämän naiivi 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 C#:sta vastineet. Tutustutaan ensiksi ensimmäiseen eli `if`-rakenteeseen.

Yleisessä muodossa C#:n `if`-rakenne on alla olevan kaltainen:

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

Esimerkki ehtolause: ”Jos aurinko paistaa, mene ulos” voidaan nyt esittää C#:n syntaksin mukaan seuraavasti.

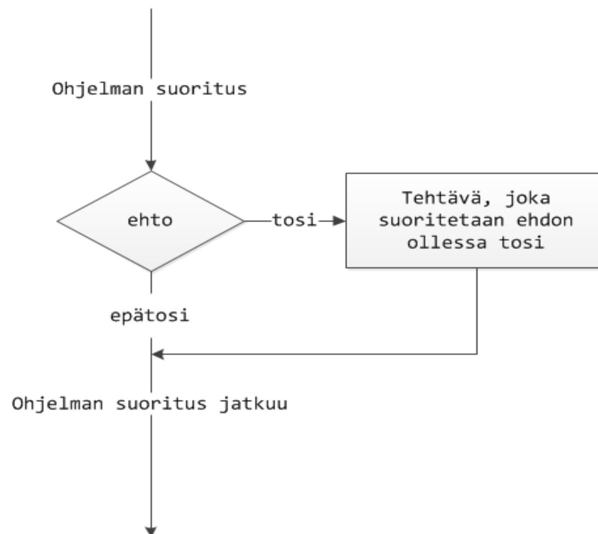
```
if (aurinkoPaistaa) MeneUlos();
```

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

```
if (ehto)
{
  lause1;
  lause2;
  ...
  lauseN;
}
```

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.

Ennen kuin `if`-rakenteesta voidaan antaa esimerkkiä, tarvitsemme hieman tietoa vertailuoperaattoreista.

13.3 Vertailuoperaattorit

Vertailuoperaattoreilla voidaan vertailla aritmeettisia arvoja.

Taulukko 6: C#:n vertailuoperaattorit.

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 (==)

Muistathan, ettei sijoitusoperaattoria (=) voi käyttää vertailuun. Tämä on yksi yleisimmistä ohjelmointivirheistä. Vertailuun aina kaksi yhtä suuri kuin -merkkiä ja sijoitukseen yksi. Tästä seuraava esimerkki.

13.4 Esimerkki: yksinkertaisia if-lauseita

Yhtäsuuruuden vertailuoperaattorissa on kaksi yhtä suuri kuin -merkkiä.

```
if (henkilonIka == 20) Console.WriteLine(”Onneksi olkoon!”);
```

Alla oleva aiheuttaa virheilmoituksen.

```
if (henkilonIka = 20) Console.WriteLine(”Onneksi olkoon!”); // Virhe!
```

Seuraava esimerkki havainnollistaa toisen vertailuoperaattorin käyttöä.

```
if (luku < 0) Console.WriteLine("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. Tällöin perässä oleva lause tai lohko suoritetaan.

13.5 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. C#:n syntaksin mukaiseksi ohjelma saadaan, kun ohjelmointikieleen kuuluvat sanat muutetaan englanniksi.

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

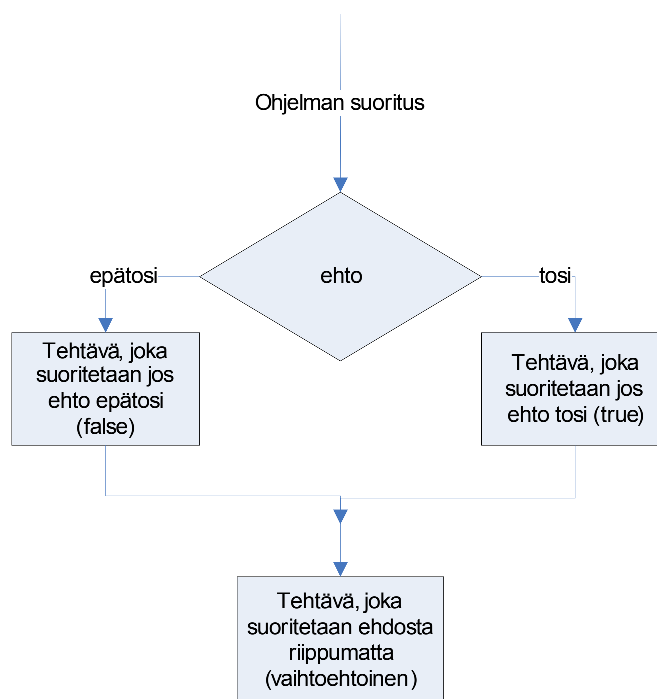
`if-else` -rakenteen yleinen muoto:

```
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.

13.5.1 Esimerkki: Pariton vai parillinen

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

```
public static bool 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, sillä `else`-lauseeseen mennään ohjelmassa nyt vain siinä tapauksessa, että `if`-ehto

ei ollut tosi. Voisimmekin kirjoittaa aliohjelman hieman lyhyemmin seuraavasti:

```
public static bool 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 bool 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 kuten aiemminkin.

13.6 Loogiset operaatiot

Loogisia lausekkeita voidaan myös yhdistellä *loogisilla operaattoreilla*.

Taulukko 7: Loogiset operaatiot.

C#-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)) Console.WriteLine("Luku on suurempi kuin nolla");
```

Ei-operaattori siis palauttaa vastakkaisen `bool`-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) Console.WriteLine("Luku on suurempi kuin nolla");
```

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

```
if ((1 <= luku) && (luku <= 99)) Console.WriteLine("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) Console.WriteLine("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 pienempi kuin -merkkiä (nuolen kärki vasemmalle). Tällöin epäyhtälön operandit ovat samassa järjestyksessä, kuin miten ihmiset mieltävät lukujen suuruusjärjestyksen.

13.6.1 De Morganin lait

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

```
!(p || q) sama asia kuin !p && !q
!(p && q) 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
{
    /// <summary>
    /// Testiohjelma DeMorganin laeille
    /// </summary>
    /// <param name="args">Ei käytössä.</param>
    public static void Main(string[] args)
    {
        bool p = true;
        bool q = true;
        Console.WriteLine(!(p || q) == (!p && !q));
        Console.WriteLine(!(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 ei-operaattorin siirto voikin olla mielekästä. Toinen tällainen laki on osittelulaki.

13.6.2 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)
```

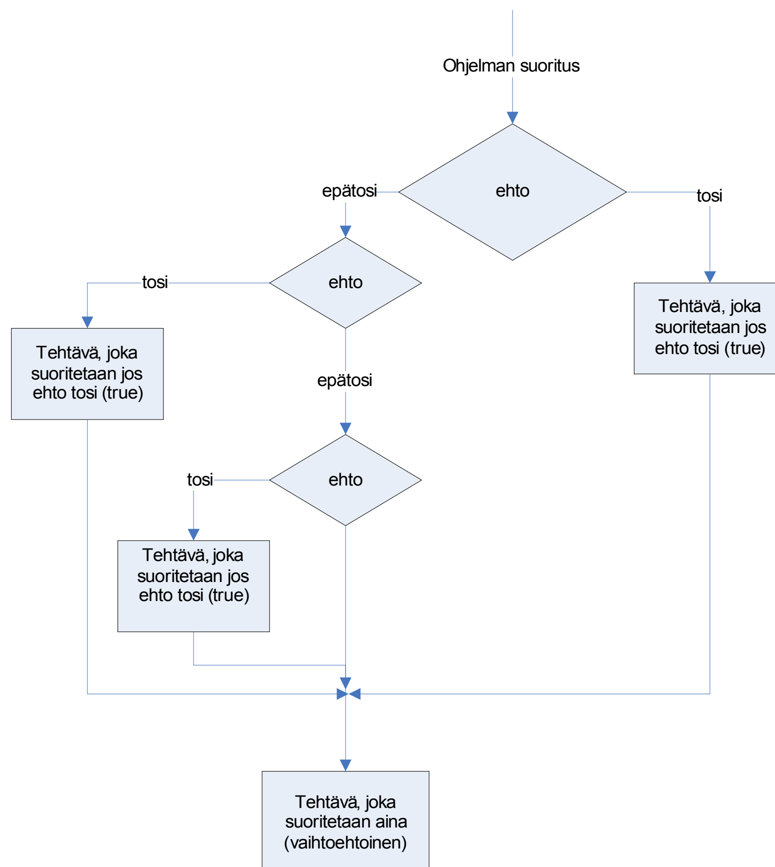
13.7 else if -rakenne

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.

13.7.1 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.

```
using System;
/// <summary>
/// Laskee opiskelijan tenttiarvosanan.
/// </summary>
public class LaskeTenttiArvosana
{
    /// <summary>
    /// Laskee tenttiarvosanan pistevälien mukaan.
    /// </summary>
    /// <param name="maksimipisteet">Tentin maksimipisteet</param>
    /// <param name="lapipaasyraja">Tentin läpipääsyraja</param>
    /// <param name="tenttipisteet">Opiskelijan saamat tenttipisteet</param>
    /// <returns>tenttiarvosana välillä 0-5.</returns>
    public static int LaskeArvosana(int maksimipisteet, int lapipaasyraja,
      int tenttipisteet)
    {
        //Lasketaan eri arvosanoille tasaiset pistevälit
        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;
    }

    /// <summary>
    /// Pääohjelmassa tehdään testitulostuksia
    /// </summary>
    /// <param name="args"></param>
    public static void Main(string[] args)
    {
        //Tehdään muutama testitulostus
        Console.WriteLine(LaskeArvosana(100, 50, 75));
        Console.WriteLine(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 15.

13.7.2 Harjoitus

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

13.7.3 Harjoitus

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

13.8 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 (valitsin) //valitsin on useimmiten joku muuttuja
{
   case arvo1:
      lauseet;
      break;

   case arvo2:
      lauseet;
      break;

   case arvoX:
      lauseet;
      break;

   default:
      lauseet;
     break;
}
```

Jokaisessa `case`-kohdassa sekä `default`-kohdassa on lauseiden jälkeen oltava lause, jolla hypätään pois `switch`-lohkosta. Ylläolevassa esimerkissä hyppylauseena toimi `break`-lause. Toisin kuin joissain esimerkiksi C++:ssa, ei C#:ssa sallita suorituksen siirtymistä tapauksesta (`case`) toiseen, mikäli tapauksessa on yksikin lause. Esimerkiksi seuraava koodi aiheuttaisi virheen.

```
switch (valitsin)
{
    // Seuraava koodi aiheuttaa virheen
    case 1:
        Console.WriteLine("Tapaus 1...");
        // Tähän kuuluisi break-lause tai muu hyppylause!
    case 2:
        Console.WriteLine("... ja/tai tapaus 2");
        break;
}
```

Kuitenkin, tapauksesta toiseen ”valuttaminen” on sallittu, mikäli tapaus *ei* sisällä yhtään lausetta. Seuraavassa on esimerkki tästä.

```
int luku = 3;
switch (luku)
{
    case 0:
    case 1:
        Console.WriteLine("Luku on 0 tai 1");
        break;
    case 2:
        Console.WriteLine("Luku on 2");
        break;
    default:
        Console.WriteLine("Oletustapaus");
```

```
        break;
}
```

13.8.1 Esimerkki: Arvosana kirjalliseksi

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

```
/// <summary>
/// Palauttaa parametrina saamansa numeroarvosanan kirjallisena.
/// </summary>
/// <param name="numero">tenttiarvosana numerona</param>
/// <returns>tenttiarvosana kirjallisena</returns>
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";
        break;
   }
   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, sillä `return`-lauseen ansiosta tapauksesta toiseen valuttaminen ei ole mahdollista.

```
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 (tai kyseinen `case`-osa ei sisällä yhtään lausetta). Muulloin `break`-lauseen poisjättäminen johtaa virheeseen.

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.

14. Olioiden ja alkeistietotyyppien erot

Tehdään ohjelma, jolla demonstroidaan olioiden ja alkeistietotyyppien eroja.

```
/// <summary>
/// Tutkitaan olioviitteiden käyttöä ja käyttäytymistä.
/// </summary>
public class Olioviitteet
{
  /// <summary>
  /// Alustetaan muuttujia ja tulostetaan.
  /// Testaillaan olioiden ja alkeismuuttujien eroja.
  /// </summary>
  /// <param name="args">Ei käytössä.</param>
  public static void Main(String[] args)
  {
    StringBuilder s1 = new StringBuilder("eka");
    StringBuilder s2 = new StringBuilder("eka");

    Console.WriteLine(s1 == s2);       // false
    Console.WriteLine(s1.Equals(s2));  // true

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

    Console.WriteLine(i1 == i2);       // true

    int[] it1 = new int[1]; it1[0] = 3;
    int[] it2 = new int[1]; it2[0] = 3;

    Console.WriteLine(it1 == it2);       // false
    Console.WriteLine(it1.Equals(it2));  // true
    Console.WriteLine(it1[0] == it2[0]); // true

    s2 = s1;
    Console.WriteLine(s1 == s2);         // true
  }
}
```

Tarkastellaan ohjelmaa hieman tarkemmin:

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

Yllä luodaan kaksi `StringBuilder`-luokan ilmentymää.

```
Console.WriteLine(s1 == s2); // false
```

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

```
Console.WriteLine(s1.Equals(s2)); // true
```

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

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

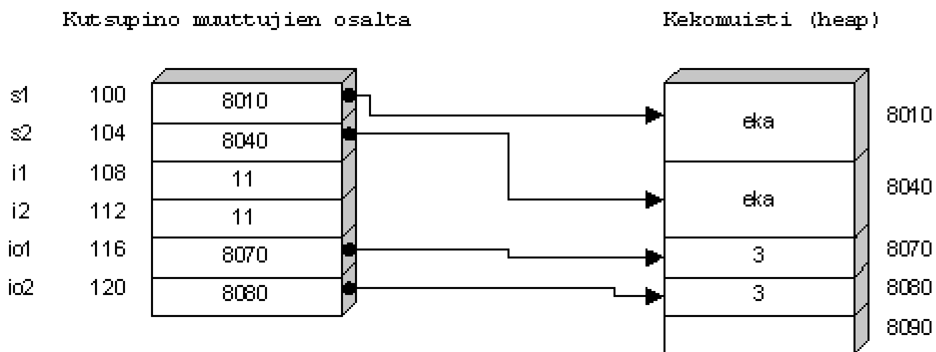
```
(i1 == i2)
```

on totta.

```
int[] it1 = new int[1]; it1[0] = 3;
int[] it2 = new int[1]; it2[0] = 3;
Console.WriteLine(it1 == it2); // false
```

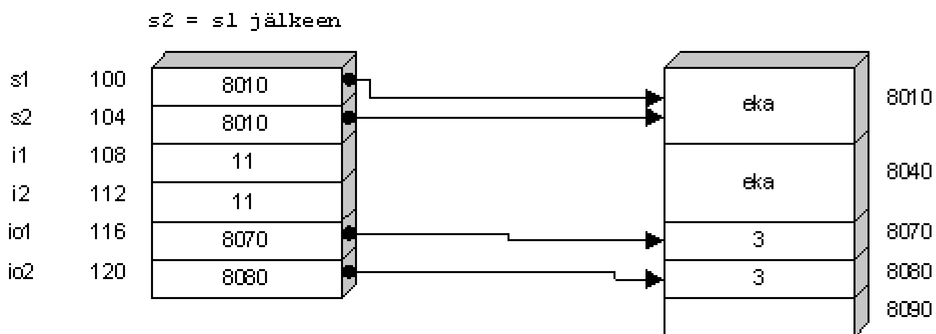
Vastaavasti kuten `StringBuilder`-olioilla yllä oleva tulostus palauttaa `false`. Huomaa, että vaikka taulukko sisältää `int`-tyyppisiä kokonaislukuja (jotka ovat primitiivityyppisiä), niin *kokonaislukutaulukko* on olio. Jälleen verrataan taulukkomuuttujien viitteitä, eikä arvoja joihin muuttujat viittaavat.

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 roskienkeruu (*garbage-collection*, *gc*) seuraavan kerran käynnistyy, "vapautetaan" tällaiset käyttämättömät muistialueet. Tätä automaattista roskienkeruuta on pidetty yhtenä syynä esimerkiksi Javan menestykseen. Samalla täytyy kuitenkin varoittaa, että muisti on

vain yksi resurssi ja automatiikka on olemassa vain muistin hoitamiseksi. Muut resurssit kuten esimerkiksi tiedostot ja tietokannat pitää edelleen hoitaa samalla huolellisuudella kuin muissakin kielissä. [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) Console.WriteLine("s2:n pituus on " + s2.Length);
```

Ilman testiä esimerkissä tulisi `NullPointerException`-poikkeus.

15. Taulukot

Muuttujaan pystytään tallentamaan yksi arvo kerrallaan. Usein ohjelmoinnissa kuitenkin tulee tilanteita, joissa meidän tulisi tallettaa useita samantyyppisiä yhteenkuuluvia arvoja. Jos haluaisimme tallettaa esimerkiksi kaikkien kuukausien päivien lukumäärä, voisimme tietenkin tehdä tämän 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 käsitellään useita samaan asiaan liittyviä jossain mielessä samankaltaisia tai yhteen liittyviä arvoja, on usein syytä ottaa käyttöön *taulukko* (**array**). Taulukko on tietorakenne, johon voi tallentaa useita samantyyppisiä muuttujia. 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.

Taulukon koko täytyy määrittää etukäteen, eikä sitä voi myöhemmin muuttaa[1].

15.1 Taulukon luominen

C#:ssa 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.

1 Array.Resize-metodi ei muuta alkuperäistä taulukkoa, vaan luo uuden taulukon, kopioi alkuperäisen taulukon kaikki alkiot uuteen taulukkoon, ja sen jälkeen korvaa alkuperäisen taulukon (viitteen) uudella taulukolla (viitteellä). Ks. http://msdn.microsoft.com/en-us/library/bb348051.aspx.

```
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, `bool`-tyyppi saa arvon `false` ja oliotyypit (esim. `String`) tyhjän merkkijonoarvon. [MÄN][KOS]

15.2 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.

```
Console.WriteLine(kuukausienPaivienLkm[0]); //tulostuu 31
```

Tai tallentaa tammikuun pituuden edelleen muuttujaan.

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

Taulukkoon viittaava indeksi voi olla myös `int`-tyyppinen lauseke, jolloin `kuukausienPaivienLkm`-taulukkoon viittaaminen onnistuu yhtä hyvin seuraavasti:

```
int indeksi = 0;
Console.WriteLine(kuukausienPaivienLkm[indeksi]); // ensimmäinen alkio
Console.WriteLine(kuukausienPaivienLkm[indeksi + 3]); // neljäs alkio
```

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 `IndexOutOfRangeException`-poikkeuksen. Tällöin kääntäjä tulostaa seuraavan kaltaisen virheilmoituksen ja ohjelman suoritus päättyy.

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the
array.
```

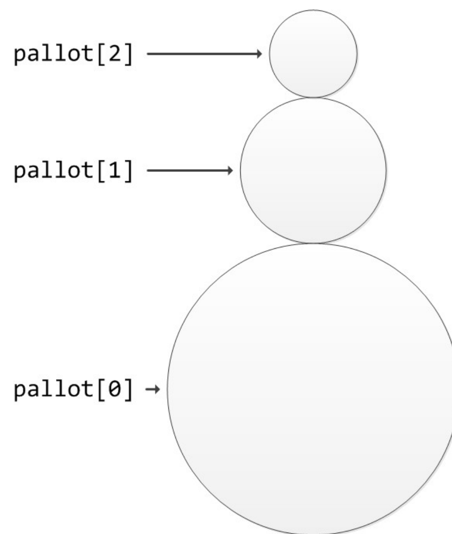
Myöhemmin opitaan kuinka poikkeuksista voidaan toipua ja ohjelman suoritusta jatkaa.

15.3 Esimerkki: lumiukon pallot taulukkoon

Luvussa 4.3 teimme lumiukon kolmesta pallosta. Tehdään sama siten, että laitetaan yksittäiset `PhysicsObject`-oliot taulukkoon.

```
/*
 * Author: Antti-Jussi Lakanen
 */

using Jypeli;
using System;

/// <summary>
/// Lumiukko, jonka pallot ovat taulukossa.
/// </summary>
public class Lumiukko : PhysicsGame
{
    /// <summary>
    /// Pääohjelmassa laitetaan "peli" käyntiin Jypelille tyypilliseen tapaan
    /// </summary>
    /// <param name="args">Ei käytössä</param>
    public static void Main(String[] args)
    {
        using (Lumiukko peli = new Lumiukko())
        {
            peli.Run();
        }
    }

    /// <summary>
    /// Piirretään oliot ja zoomataan kamera niin että kenttä näkyy kokonaan.
    /// </summary>
    public override void Begin()
    {
        Camera.ZoomToLevel();
        Level.BackgroundColor = Color.Black;

        // Lisätään pallot taulukkoon, ja sitten lisätään kentälle
        PhysicsObject[] pallot = new PhysicsObject[3];
        pallot[0] = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
        pallot[0].Y = Level.Bottom + 200.0;
        pallot[1] = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
        pallot[1].Y = pallot[0].Y + 100 + 50;
        pallot[2] = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
        pallot[2].Y = pallot[1].Y + 50 + 30;

        Add(pallot[0]); Add(pallot[1]); Add(pallot[2]);
    }
}
```

Näkyvä lopputulos on sama lumiukko kuin aikaisemminkin. Nyt pallot ovat kuitenkin taulukkorakenteessa.



Kuva 19: Lumiukon pallot ovat pallot-taulukon alkioita.

Nyt taulukon avulla pääsemme käsiksi yksittäisiin pallo-olioihin. Esimerkiksi keskimmäisen pallon värin muuttaminen onnistuisi seuraavasti

```
pallot[1].Color = Color.Yellow;
```

15.4 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.

```
/// <summary>
/// Palauttaa parametrina saamansa numeroarvosanan kirjallisena.
/// </summary>
/// <param name="numero">tenttiarvosana numerona</param>
/// <returns>tenttiarvosana kirjallisena</returns>
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 viisi kirjallista arvosanaa, `switch`-rakenteessa tehtäisiin viisi vertailuoperaatiota. Taulukkoa käyttämällä vertailuoperaatioita ei tehdä yhtään, vaan ainoastaan yksi hakuoperaatio taulukosta.

15.5 Moniulotteiset taulukot

Taulukot voivat olla myös moniulotteisia. Kaksiulotteinen taulukko (eli matriisi) on esimerkki moniulotteisesta taulukosta, joka koostuu vähintään kahdesta samanpituisesta taulukosta. Kaksiulotteisella taulukolla voidaan esittää esimerkiksi tason tai kappaleen pinnan koordinaatteja.

Kaksiulotteinen taulukko määritellään seuraavasti:

```
tyyppi[,] taulukonNimi;
```

Huomaa, että määrittelyssä `[,]` tarkoittaa, että esitelty taulukko on kaksiulotteinen. Vastaavasti `[,,]` tarkoittaisi, että taulukko on kolmiulotteinen ja niin edelleen.

Moniulotteisen taulukon alkioiden määrä tulee aina ilmoittaa ennen taulukon käyttöä. Tämä tapahtuu `new`-operaattorilla seuraavasti:

```
taulukonNimi = new tyyppi[rivienLukumaara, sarakkeidenLukumaara]
```

Esimerkiksi kaksiulotteisen `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";
```

Taulukko näyttäisi nyt seuraavalta:

Kuva 20: `kurssinOpiskelijat`-taulukko

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

```
// tulostaa Ville Virtanen
Console.WriteLine(kurssinOpiskelijat[0,1] + " " + kurssinOpiskelijat[0,0]);
```

Huomaa, että yllä olevassa esimerkissä ”+”- merkki ei toimi aritmeettisena operaattorina, vaan sillä yhdistetään tulostettavia merkkijonoja. C#:ssa ”+”-merkkiä käytetään siis myös merkkijonojen yhdistelyyn.

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:

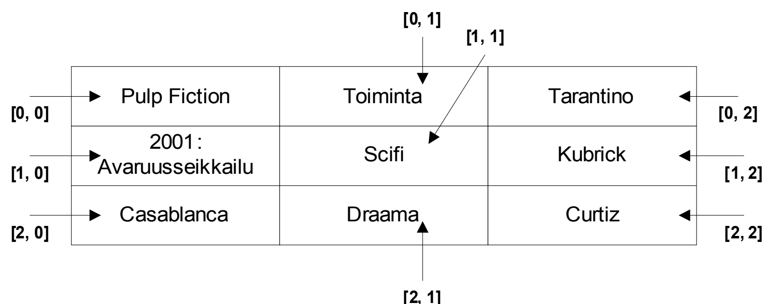
```
// tulostaa Virtanen, Ville
Console.WriteLine(kurssinOpiskelijat[0,0] + ", " + kurssinOpiskelijat[0,1]);
```

Todellisuudessa henkilötietorekisteriä ei kuitenkaan tehdä tällä tavalla. 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 = new String[3,3] { {"Pulp Fiction", "Toiminta", "Tarantino"},
                        {"2001: Avaruusseikkailu", "Scifi", "Kubrick"},
                        {"Casablanca", "Draama", "Curtiz"} };
```

Yllä oleva määrittely luo 3 x 3 kokoisen taulukon:



Kuva 21: Taulukon `elokuvat` sisältö.

Kun taulukko on luotu, sen alkioihin viitataan seuraavalla tavalla.

```
taulukonNimi[rivi-indeksi, sarakeindeksi]
```

Alla oleva esimerkki hahmottaa taulukon alkioihin viittaamista.

```
Console.WriteLine(elokuvat[0, 0]);  //tulostaa "Pulp Fiction"
Console.WriteLine("Tyyppi: " + elokuvat[0, 1]); //tulostaa "Tyyppi:  Toiminta"
Console.WriteLine("Ohjaaja: " + elokuvat[0, 2]); //tulostaa "Ohjaaja: Tarantino"
```

> Tällä tavalla jokaiselle riville tulee yhtä monta saraketta eli alkiota. Jos eri riveille halutaan eri määrä alkioita, voidaan käyttää ns. jagged array -taulukkoja. Lue lisää jagged arrayn MSDN-dokumentaatiosta osoitteesta `http://msdn.microsoft.com/en-us/library/2s05feca(v=vs.80).aspx`.

15.5.1 Harjoitus

Miten tulostat taulukosta Casablanca? Entä Kubrick?

15.6 Taulukon kopioiminen

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

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

taulukko2[0] = 10;
Console.WriteLine(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 = (int[])taulukko.Clone();
```

Huomaa, että sijoituksessa vaaditaan ns. tyyppimuunnos: ennen `taulukko.Clone`-lausetta kirjoitetaan `(int[])`, sulkujen kanssa, joka muuttaa `Clone`-metodin palauttaman ”yleisen” `Object`-olion kokonaislukutaulukoksi.

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

```
kopio_taulukosta[0] = 3;
Console.WriteLine(taulukko[0] + ", " + kopio_taulukosta[0]); // 1, 3
```

15.7 Esimerkki: Moniulotteiset taulukot käytännössä

Kaksiulotteisia taulukoita kutsutaan yleisesti matriiseiksi, ja ne ovat käytössä erityisesti matemaattisissa sovelluksissa kuvaten lineaarifunktioita. Muitakin käyttökohteita matriiseilla kuitenkin on. Esimerkiksi laivanupotuspelin pelikenttä voidaan ajatella 2-ulotteiseksi taulukoksi.

```
using System;

/*
 * Author: Antti-Jussi Lakanen
 */

/// <summary>
/// Moniulotteiset taulukot käytännössä.
/// </summary>
public class Laivanupotus
{
  /// <summary>
  /// Taulukon alustus ja tulostus.
  /// </summary>
  /// <param name="args"></param>
  public static void Main(string[] args)
  {
    int[,] ruudut = { { 1, 0, 2 }, { 0, 0, 3 } };
    Console.WriteLine(ruudut[0, 0] + " " + ruudut[0, 1] + " " + ruudut[0, 2]);
    Console.WriteLine(ruudut[1, 0] + " " + ruudut[1, 1] + " " + ruudut[1, 2]);
    // Tulostaa:
    // 1 0 2
    // 0 0 3
  }
}
```

Vaikka ruudukko oli vielä aika pieni (2 riviä x 3 saraketta), on alkioiden tulostaminen melko työlästä. Esimerkiksi jo 20 x 20 kokoisen taulukon tulostaminen pelkkiä tulostuslauseita peräkkäin laittamalla olisi jo kohtuuttoman iso työ.

Edelleen, yhtenä toiveena voisi olla, että löytäisimme ”ruudukosta” rivin, jolla tyhjiä paikkoja on eniten. Tämä tieto voisi auttaa meitä asemoimaan uuden laivan oikein. Mielivaltaiselle ruudukolle tämä ei vielä meidän tiedoillamme onnistu.

Näihin tehtäviin tarvitsemme toistorakenteita, joka esitellään seuraavassa luvussa.

16. 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:

```
Console.WriteLine(kuukausienPaivienLkm[0]);
Console.WriteLine(kuukausienPaivienLkm[1]);
Console.WriteLine(kuukausienPaivienLkm[2]);
Console.WriteLine(kuukausienPaivienLkm[3]);
Console.WriteLine(kuukausienPaivienLkm[4]);
Console.WriteLine(kuukausienPaivienLkm[5]);
Console.WriteLine(kuukausienPaivienLkm[6]);
Console.WriteLine(kuukausienPaivienLkm[7]);
Console.WriteLine(kuukausienPaivienLkm[8]);
Console.WriteLine(kuukausienPaivienLkm[9]);
Console.WriteLine(kuukausienPaivienLkm[10]);
Console.WriteLine(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**).

Tämä luku on pitkä ja sisältää runsaasti esimerkkejä. Toistorakenteiden hallinta on kuitenkin hyvin tärkeää ohjelmoinnin opettelun alkuvaiheilla.

16.1 ”Syö niin kauan, kuin puuroa on lautasella”

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

```
Syö aamupuuroa niin kauan, kuin 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.
```

C#:ssa on neljän tyyppisiä toistorakenteita:

- `for`
- `while`
- `do-while`
- `foreach`

On tilanteita, joissa voimme vapaasti valita näistä minkä tahansa, mutta useimmiten toistorakenteen valinnan kanssa täytyy olla tarkkana. Jokaisella näistä on tietyt ominaispiirteensä, eivätkä kaikki toistorakenteet sovi kaikkiin mahdollisiin tilanteisiin.

16.2 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`. Ehdon jälkeen voi yksittäisen lauseen sijaan olla myös lohko.

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

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.

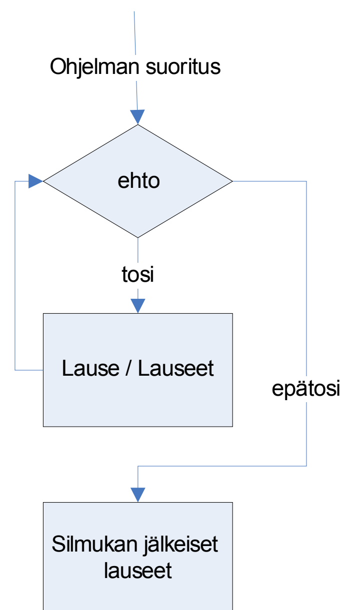
16.2.1 Huomautus: ikuinen silmukka

Huomaa, että jos while-silmukan ehto on aina `true`, on kyseessä *ikuinen silmukka* (**infinite loop**). Ikuinen silmukka on nimensä mukaisesti silmukka, joka ei pääty koskaan. Ikuinen silmukka johtuu siitä, että silmukan ehto ei saa koskaan arvoa `false`. Useimmiten ikuinen silmukka on ohjelmointivirhe, mutta joskus (hallitun) ikuisen silmukan tekeminen on perusteltua. Tällöin silmukasta kuitenkin poistutaan (ennemmin tai myöhemmin) `break`-lauseen avulla. Tällöinhän silmukka ei oikeastaan ole ikuinen, vaikka tällaisesta silmukasta sitä nimitystä usein käytetäänkin. `break`-lauseesta puhutaan tässä luvussa myöhemmin kohdassa 16.8.1.

Ikuinen silmukka on mahdollista tehdä myös muilla toistorakenteilla.

16.2.2 While-silmukka vuokaaviona

`while`-silmukan voisi esittää vuokaaviona seuraavasti.



Kuva 22: while-silmukka vuokaaviona

16.2.3 Esimerkki: Taulukon tulostaminen

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

```
public class Silmukat
{
/// <summary>
/// Tulostaa int-tyyppisen taulukon sisällön.
/// </summary>
/// <param name="taulukko">Tulostettava taulukko</param>
  public static void TulostaTaulukko(int[] taulukko)
  {
    int i = 0;
    while (i < taulukko.Length){
      Console.Write(taulukko[i] + " ");
      i++;
    }
  }

  /// <summary>
  /// Pääohjelma.
  /// </summary>
  /// <param name="args">Ei käytössä.</param>
  public static void Main(String[] args)
  {
    int[] kuukausienPaivienLkm =
      {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    TulostaTaulukko(kuukausienPaivienLkm);
    Console.ReadKey();
  }
}
```

Tarkastellaan `TulostaTaulukko`-aliohjelman sisältöä hieman tarkemmin.

```
int i = 0;
```

Tässä luodaan uusi muuttuja, jolla kontrolloidaan, mitä taulukon alkioita ollaan tulostamassa. Lisäksi sen avulla selvitetään, milloin taulukon kaikki alkiot on tulostettu ruudulle. Muuttuja alustetaan arvoon `0`, sillä taulukon ensimmäinen alkio on aina indeksissä 0. Muuttujalle annetaan

nimeksi `i`. Useimmiten pelkät kirjaimet ovat huonoja muuttujan nimiä, koska ne kuvaavat muuttujaa huonosti. Silmukoissa kuitenkin nimi `i` on vakiinnuttanut asemansa kontrolloimassa silmukoiden kierroksia, joten sitä voidaan hyvällä omallatunnolla käyttää.

```
while (i < taulukko.Length)
```

Aliohjelman toisella rivillä aloitetaan `while`-silmukka. Ehtona on, 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.

```
Console.Write(taulukko[i] + " ");
```

Ensimmäisessä silmukan lauseessa tulostetaan taulukon alkio indeksissä `i`. Perään tulostetaan välilyönti erottamaan eri alkiot toisistaan. `Console.WriteLine`-metodin sijaan käytämme nyt toista `Console`-luokan metodia. `Console.Write`-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. Indeksimuuttujan hallintaan liittyvät virheet ovat tyypillisiä aloittelevan (ja pidemmällekin edistyneen) ohjelmoijan virheitä. Ongelmalliseksi virheen tekee se, ettei se ole syntaksivirhe, jolloin esimerkiksi Visual Studio ei anna tilanteesta virheilmoitusta.

`while`-silmukkaa tulisi käyttää silloin, kun meillä ei ole tarkkaa tietoa silmukan suorituskierrosten lukumäärästä. Koska taulukon koko on tarkalleen tiedossa taulukon luomisen jälkeen, olisi läpikäyminen käytännössä järkevämpää ja tehdä `for`-silmukalla, missä vaara ikuisen silmukan syntymiseen on pienempi. Myöhemmin löytyy järkevämpää käyttöä `while`-silmukalle.

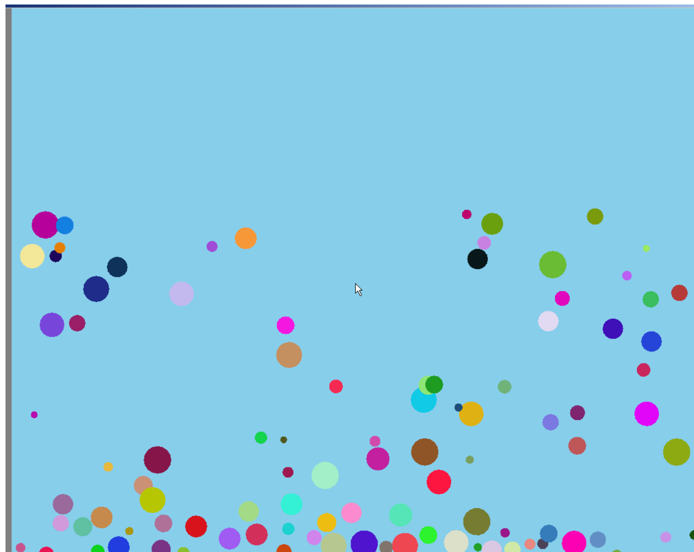
16.2.4 Esimerkki: Monta palloa

Tehdään aliohjelma, joka luo halutun kokoisen pallon haluttuun paikkaan, ja vielä haluamalla värillä. `Main`-metodi on jätetty listauksesta pois.

```
using System;
using Jypeli;

/// <summary>
/// Paljon palloja tippuu alaspäin.
/// </summary>
public class MontaPalloa : PhysicsGame
{
    /// <summary>
    /// Ruudulla näkyvä sisältö.
    /// </summary>
    public override void Begin()
    {
        Level.CreateBorders();
        Gravity = new Vector(0, -500);
        Camera.ZoomToLevel();

        int i = 0;
        while (i < 100)
        {
            int sade = RandomGen.NextInt(5, 20);
            double x = RandomGen.NextDouble(Level.Left + sade, Level.Right - sade);
```

```
            double y = RandomGen.NextDouble(Level.Bottom + sade, Level.Top - sade);
            Color vari = RandomGen.NextColor();
            PhysicsObject pallo = LuoPallo(x, y, vari, sade);
            Add(pallo);
            i++;
        }
    }

    /// <summary>
    /// Luo yksittäisen pallon ja palauttaa sen.
    /// </summary>
    /// <param name="x">Pallon kp x-koordinaatti</param>
    /// <param name="y">Pallon kp y-koordinaatti</param>
    /// <param name="vari">Pallon väri</param>
    /// <param name="sade">Pallon säde</param>
    public static PhysicsObject LuoPallo(double x, double y, Color vari, double sade)
    {
        PhysicsObject pallo = new PhysicsObject(2 * sade, 2 * sade, Shape.Circle);
        pallo.Color = vari;
        pallo.X = x;
        pallo.Y = y;
        return pallo;
    }
}
```

Ajettaessa koodin tulisi piirtää ruudulle sata palloa, jotka putoavat alaspäin kohti kentän reunaa. Katso seuraava kuva.



Kuva 23: Pallot tippuu.

Tutkitaan tarkemmin `LuoPallo`-aliohjelmaa Aliohjelma palauttaa `PhysicsObject`-olion, siis paluuarvon tyyppinä on luonnollisesti `PhysicsObject`. Parametreja ovat

```
double x, double y, Color vari, double sade
```

siis pallon keskipisteen x- ja y-koordinaatit, väri ja säde.

Huomaa, että `LuoPallo` on tässä funktioaliohjelma, joka ei tee ohjelmassa mitään ”näkyvää”. Se vain luo pallon, kuten nimikin kertoo, mutta ei lisää sitä ruudulle. Tästä syystä ei tarvita myöskään `Game`-parametria, joka `Lumiukko`-esimerkissä aikanaan tarvittiin. Sen sijaan lisääminen tehdään `Begin`-aliohjelmassa. Yleisesti ottaen aliohjelmissa ei pidä tehdä enempää kuin mitä

dokumentaatiossa kerrotaan – jopa aliohjelman nimestä pitäisi kaikkein tärkein selvitä.

Jos haluttaisiin, että tämä kyseinen aliohjelma myös lisää pallon ruutuun, tulisi se nimetä jotenkin muuten, esimerkiksi `LisaaPallo` olisi loogisempi vaihtoehto. Silloin palloa ei palautettaisi kysyjälle ja paluuarvon tyypiksi tulisi `void`.

Siirrytään sitten takaisin `Begin`-aliohjelmaan.

```
int i = 0;
while (i < 100)
```

Tässä alustetaan `int`-tyyppinen indeksi `i` nollaksi ja määritetään `while`-sanan jälkeen sulkujen sisään ehto jonka perusteella silmukassa etenemistä jatketaan. Aaltosulut on jätetty listauksesta tarkoituksellisesti pois.

```
int sade = RandomGen.NextInt(5, 20);
double x = RandomGen.NextDouble(Level.Left + sade, Level.Right - sade);
double y = RandomGen.NextDouble(Level.Bottom + sade, Level.Top - sade);
Color vari = RandomGen.NextColor();
PhysicsObject pallo = LuoPallo(x, y, vari, sade);
Add(pallo);
i++;
```

Silmukassa arvotaan ensin kunkin pallon säde `RandomGen`-luokan satunnaislukugeneraattorilla. Ensimmäisenä parametrina `NextInt`-aliohjelmalle annetaan pienin mahdollinen arvottava luku, toisena parametrina luku, jota pienempi arvottavan luvun tulee olla. Toisin sanoen, luvut tulevat olemaan välillä 5–19. Samoin arvotaan `double`-tyyppiset koordinaatit sekä `Color`-tyyppinen väri.

Tämän jälkeen luodaan normaalisti `PhysicsObject`-fysiikkaolio, ja annetaan `LuoPallo`-aliohjelmalle parametrina juuri tekemämme muuttujat, joka sitten palauttaa haluamamme pallon. Pallo lisätään kentälle `Add`-metodin avulla.

```
i++;
```

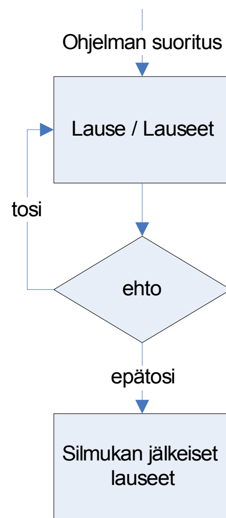
Silmukan jokaisen ”kierroksen” jälkeen indeksin arvoa on lisättävä yhdellä, ettemme joutuisi ikuiseen silmukkaan.

16.3 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. Yleisessä muodossa `do-while` -silmukka on seuraavanlainen:

```
do
{
  lause1;
  lause2;
  (...)
  lauseN;
} while (ehto);
```

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



Kuva 24: `do-while`-silmukka vuokaaviona.

16.3.1 Esimerkki: nimen kysyminen käyttäjältä

Seuraavassa esimerkissä käyttäjää pyydetään syöttämään merkkijono Jos käyttäjä antaa tyhjän jonon, kysytään nimeä uudestaan. Tätä toistetaan niin kauan, kunnes käyttäjä antaa jotain muuta kuin tyhjän jonon.

```
using System;

/// <summary>
/// Harjoitellaan do-while-silmukan käyttöä.
/// </summary>
public class NimenTulostus
{
    /// <summary>
    /// Pyydetään käyttäjältä syöte ja tulostellaan.
    /// </summary>
    /// <param name="args"></param>
    public static void Main(string[] args)
    {
        String nimi;
        do
        {
            Console.Write("Anna nimi > ");
            nimi = Console.ReadLine();
        } while (nimi.Length == 0);
        Console.WriteLine("Hei, " + nimi + "!");
        Console.ReadKey();
    }
}
```

Tämä kuvaa hyvin `do-while`-silmukan olemusta: nimi halutaan kysyä varmasti ainakin kerran, mutta mahdollisesti useamminkin – emme kuitenkaan voi olla varmoja kuinka monta kertaa useammin.

Todellisuudessa nimen oikeellisuuden tarkistaminen olisi tietenkin monimutkaisempaa, mutta idea `do-while`-silmukan osalta olisi täsmälleen vastaava.

16.4 for-silmukka

Kun silmukan suoritusten lukumäärä on ennalta tiedossa, on järkevintä käyttää `for`-silmukkaa.

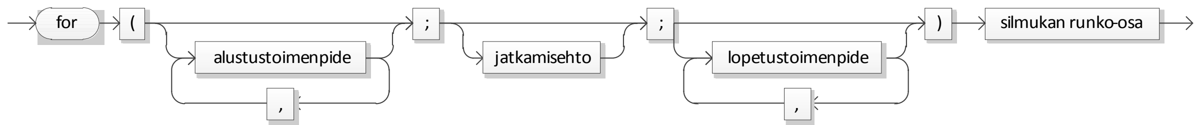
Esimerkiksi taulukoiden käsittelyyn `for`-silmukka on yleensä paras vaihtoehto. Syntaksiltaan `for`-silmukka eroaa selvästi edellisistä. Perinteinen `for`-silmukka on yleisessä muodossa seuraavanlainen:

```
for (muuttujien alustukset; ehto; silmukan lopussa tehtävät toimenpiteet)
{
   lauseet; // silmukan runko-osa
}
```

Silmukan *kontrollilauseke* eli kaarisulkujen sisäpuoli 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.

Alla for-silmukan syntaksi graafisessa ”junarataformaatissa” (ks. luku 28.2) – tosin tarkoitusta varten hieman yksinkertaistettuna.



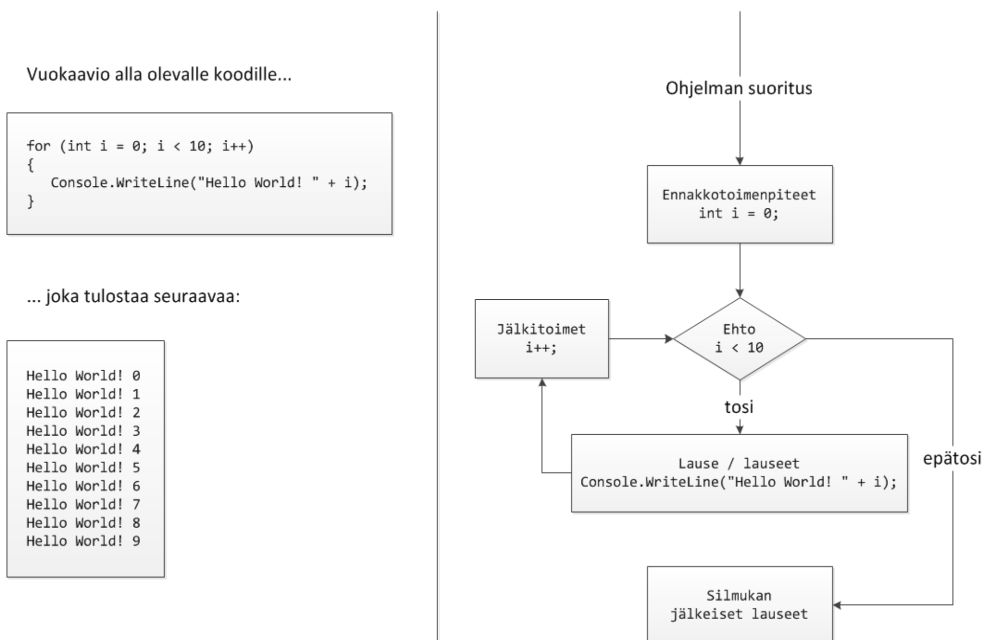
Kuva 25: for-silmukan syntaksi graafisessa "junaratamuodossa".

Alla esimerkki yksinkertaisesta `for`-silmukasta. Siinä tulostetaan 10 kertaa ”Hello World!” ja perään `i`-muuttujan sen hetkinen arvo.

```
for (int i = 0; i < 10; i++)
{
   Console.WriteLine("Hello World " + i);
}
```

Kontrollilausekkeessa alustetaan aluksi muuttujan `i` arvoksi 0. Seuraavaksi ehtona on, että silmukan suoritusta jatketaan niin kauan kuin muuttujan `i` arvo on pienempää kuin luku 10. Lopuksi kontrollilausekkeessa todetaan, että muuttujan `i` arvoa kasvatetaan joka kierroksella yhdellä.

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



Kuva 26: Vuokaavio for-silmukalle

Huomaa, että `i`-muuttujan arvo alkaa nollasta, joka tulostetaan ensimmäisenä Hello World! -tekstin jälkeen. Silmukan runko-osan suorittamisen ehtona on, että i-muuttujan arvon on oltava alle 10, joten kun `i` saavuttaa arvon 10 (9:n kierroksen päätteeksi), poistutaan silmukasta.

Silmukan runko-osassa ei suinkaan aina tarvitse tulostaa mitään. Otetaan esimerkki, missä luodun taulukon alkioihin sijoitetaan aina kahden edellisen alkion sisältämien lukujen summa. Ensimmäisen alkion arvoksi asetetaan ”manuaalisesti” luku 1.

```
int[] luvut = new int[10];
luvut[0] = 1;

for (int i = 1; i < luvut.Length; i++)
{
  luvut[i] = luvut[i - 1] + luvut[Math.Max(1, i - 2)];
}
```

Yllä `Math.Max`-funktio palauttaa kahdesta luvusta suuremman. Mieti, miksi tähän ei voi suoraan kirjoittaa:

```
luvut[i] = luvut[i - 1] + luvut[i – 2];
```

Silmukan jälkeen taulukon sisältö näyttää seuraavalta.

```
      [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
luvut  1   1   2   3   5   8   13  21  34  55
```

Myös for-silmukalla voidaan tehdä "ikuinen" silmukka:

```
for(;;)
{
  // ”ikuisesti” suoritettavat asiat
}
```

16.4.1 Huomautus: while ja for -silmukoiden yhtäläisyydet ja erot

`for`- ja `while`-silmukkarakenteilla voidaan periaatteessa tehdä täsmälleen samat asiat. `for`-silmukan yleisen muodon

```
for (muuttujien alustukset; ehto; silmukan lopussa tehtävät toimenpiteet)
{
   lauseet; // silmukan runko-osa
}
```

voisi tehdä `while`-rakenteella seuraavasti.

```
muuttujien alustukset;
while (ehto)
{
  lauseet; // silmukan runko-osa
  silmukan lopussa tehtävät toimenpiteet;
}
```

Mihin for-silmukkaa sitten tarvitaan?

Nuo kolme osaa – muuttujien alustus, ehto, lopussa tehtävät toimenpiteet – ovat osia, jotka jokaisessa silmukkarakenteessa tarvítaan. `for`-silmukassa ne kirjoitetaan selvästi perättäin yhdelle riville, jolloin ne on helpompi saada kirjoitettua kerralla oikein. Silmukan suoritusta ohjaavat tekijät on myös helpompi lukea yhdeltä riviltä, kuin yrittää selvittää sitä eri riveiltä (silmukkahan voi olla hyvinkin monta koodiriviä pitkä).

16.4.2 Esimerkki: lumiukon pallot keltaiseksi

Palataan esimerkkiin 15.3. Koska pallo-oliot ovat taulukossa, voimme käydä taulukon alkiot läpi silmukan avulla, ja muuttaa kaikkien pallojen värin toiseksi. `Main`-metodi on jätetty listauksesta pois. Pallojen luomiseen käytämme esimerkissä 16.2.4 esiteltyä `LuoPallo`-aliohjelmaa.

```
using System;
using Jypeli;

/// @author  Antti-Jussi Lakanen
/// @version 22.12.2011

/// <summary>
/// Lumiukko, jonka pallot väritetään.
/// </summary>
public class LumiukkoKeltaisetPallot : PhysicsGame
{
    /// <summary>
    /// Piirretään oliot ja zoomataan kamera niin että kenttä näkyy kokonaan.
    /// </summary>
    public override void Begin()
    {
        Camera.ZoomToLevel();
        Level.BackgroundColor = Color.Black;
        // Lisätään pallot taulukkoon, ja sitten lisätään kentälle
        PhysicsObject[] pallot = new PhysicsObject[3];
```

```
        pallot[0] = LuoPallo(0, Level.Bottom + 200, Color.White, 100);
        pallot[1] = LuoPallo(0, pallot[0].Y + 100 + 50, Color.White, 50);
        pallot[2] = LuoPallo(0, pallot[1].Y + 50 + 30, Color.White, 30);
        Add(pallot[0]); Add(pallot[1]); Add(pallot[2]);

        // Muutetaan pallojen väri
        for (int i = 0; i < pallot.Length; i++)
        {
            pallot[i].Color = Color.Yellow;
        }
    }

    /// <summary>Luo yksittäisen pallon ja palauttaa sen.</summary>
    /// <param name="x">Pallon kp x-koordinaatti</param>
    /// <param name="y">Pallon kp y-koordinaatti</param>
    /// <param name="vari">Pallon väri</param>
    /// <param name="sade">Pallon säde</param>
    public static PhysicsObject LuoPallo(double x, double y, Color vari, double sade)
    {
        PhysicsObject pallo = new PhysicsObject(2 * sade, 2 * sade, Shape.Circle);
        pallo.Color = vari;
        pallo.X = x;
        pallo.Y = y;
        return pallo;
    }
}
```

16.4.3 Harjoitus

Tee aliohjelma, joka muuttaa `PhysicsObject`-taulukossa olevien olioiden värin halutuksi.

16.4.4 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 `Keskiarvo`, joka laskee taulukossa olevien kokonaislukujen keskiarvon. Kirjoitetaan sille myös ComTest-testit.

```
/// <summary>
/// Palauttaa parametrina saamansa int-taulukon
/// alkoiden keskiarvon.
/// </summary>
/// <param name="luvut">summattavat luvut</param>
/// <returns>summa</returns>
/// <example>
/// <pre name="test">
/// int[] luvut1 = {0};
/// Laskuja.Keskiarvo(luvut1) ~~~ 0;
/// int[] luvut2 = {3, 3, 3};
/// Laskuja.Keskiarvo(luvut2) ~~~ 3;
/// int[] luvut3 = {3, -3, 3};
/// Laskuja.Keskiarvo(luvut3) ~~~ 1;
/// int[] luvut4 = {-3, -6};
/// Laskuja.Keskiarvo(luvut4) ~~~ -4.5;
/// </pre>
/// </example>
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, on ehdottoman kätevää asettaa myös laskurimuuttuja `i` 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 "`IndexOutOfRangeException`"-poikkeuksen.

```
return summa / luvut.Length;
```

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

16.4.5 Harjoitus

Pohdi, mikä tärkeä tapaus jää huomiotta taulukon keskiarvon laskemisessa.

16.4.6 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.

```
/// <summary>
/// Aliohjelma kääntää kokonaisluku-taulukon alkiot päinvastaiseen
/// järjestykseen.
/// </summary>
/// <param name="taulukko">Käännettävä taulukko.</param>
public static void KaannaTaulukko(int[] taulukko)
{
   int temp;
   for (int vasen = 0, oikea = taulukko.Length-1; vasen < oikea; vasen++, oikea--)
   {
      temp = taulukko[vasen];
      taulukko[vasen] = taulukko[oikea];
      taulukko[oikea] = temp;
   }
}
```

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;
```

Ensimmäiseksi metodissa on alustettu `temp`-niminen muuttuja. Muuttuja 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.

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.

16.4.7 Harjoitus

Tee funktiosta `KaannaTaulukko` sivuvaikutukseton versio.

16.4.8 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. Kirjoitetaan samalla ComTest-testit.

```
using System;
using System.Collections.Generic;

/// @author  Antti-Jussi Lakanen, Martti Hyvönen
/// @version 22.12.2011

/// <summary>
/// Harjoitellaan vielä taulukoiden käyttöä.
/// </summary>
public class Arvosana
{
    /// <summary>
    /// Laskee opiskelijan tenttiarvosanan asteikoilla 0-5.
    /// </summary>
    /// <param name="maksimipisteet">Tentin maksimipisteet.</param>
    /// <param name="lapaisyraja">Tentin läpipääsyraja.</param>
    /// <param name="tenttipisteet">Opiskelijan tenttipisteet.</param>
    /// <returns>Opiskelijan tenttiarvosana.</returns>
```

```
    /// <example>
    /// <pre name="test">
    /// Arvosana.LaskeArvosana(24, 12, 11) === 0;
    /// Arvosana.LaskeArvosana(24, 12, 12) === 1;
    /// Arvosana.LaskeArvosana(24, 12, 13) === 1;
    /// Arvosana.LaskeArvosana(24, 12, 14) === 1;
    /// Arvosana.LaskeArvosana(24, 12, 15) === 2;
    /// Arvosana.LaskeArvosana(24, 12, 19) === 3;
    /// Arvosana.LaskeArvosana(24, 12, 20) === 4;
    /// Arvosana.LaskeArvosana(24, 12, 22) === 5;
    /// Arvosana.LaskeArvosana(24, 12, 28) === 5;
    /// </pre>
    /// </example>
    public static int LaskeArvosana(int maksimipisteet, int lapaisyraja, int
tenttipisteet)
    {
        double[] arvosanaRajat = new double[6];
        double arvosanojenPisteErot = (maksimipisteet - lapaisyraja) / 5.0;

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

        //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;
    }

    /// <summary>
    /// Pääohjelma
    /// </summary>
    /// <param name="args">Ei käytössä.</param>
    public static void Main(String[] args)
    {
        Console.WriteLine(LaskeArvosana(24, 12, 19)); // tulostaa 3
        Console.WriteLine(LaskeArvosana(24, 12, 11)); // tulostaa 0
        Console.ReadKey();
    }
}
```

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

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

Aliohjelman alussa alustetaan tenttiarvosanojen pisterajoille 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 ja niin edelleen. Näin taulukon ensimmäinen indeksi jää käyttämättä, mutta taulukkoon viittaaminen on selkeämpää. Koska pisterajat voivat olla desimaalilukuja, on taulukon oltava tyypiltään `double[]`.

```
double arvosanojenPisteErot = (maksimipisteet - lapaisyraja) / 5.0;
```

Yllä oleva rivi laskee arvosanojen välisen piste-eron. Mieti, miksi jakoviivan alapuolelle on luku laitettava muodossa 5.0, eikä 5.

```
arvosanaRajat[1] = lapaisyraja;
```

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 kokeiltu muutamilla testitulostuksilla. Tarkemmat testit on tehty kuitenkin ComTest-testeinä, joita voidaan testata automaattisesti.

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. Molempien aliohjelmien ComTest-testit ovat myös näkyvillä.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Globalization;

/// @author  Antti-Jussi Lakanen, Martti Hyvönen
/// @version 22.12.2011

/// <summary>
/// Harjoitellaan taulukoiden käyttöä ja lasketaan tenttiarvosanoja.
/// </summary>
public class Arvosanat
{
    /// <summary>
    /// Laskee tenttiarvosanojen pisterajat taulukkoon.
    /// </summary>
    /// <param name="maksimipisteet">tentin maksimipisteet</param>
    /// <param name="lapaisyraja">tentin läpipääsyraja</param>
    /// <returns>arvosanojen pisterajat taulukossa</returns>
    /// <example>
    /// <pre name="test">
    /// double[] rajat1 = Arvosanat.LaskeRajat(24, 12);
    /// String rajat1Jono = Arvosanat.TaulukkoJonoksi(rajat1);
    /// rajat1Jono === "0, 12, 14.4, 16.8, 19.2, 21.6";
    /// double[] rajat2 = Arvosanat.LaskeRajat(30, 15);
    /// String rajat2Jono = Arvosanat.TaulukkoJonoksi(rajat2);
    /// rajat2Jono === "0, 15, 18, 21, 24, 27";
    /// </pre>
    /// </example>
    public static double[] LaskeRajat(int maksimipisteet, int lapaisyraja)
```

```
{
    double[] arvosanaRajat = new double[6];
    double arvosanojenPisteErot = Math.Round((maksimipisteet - lapaisyraja) / 5.0, 1);

    arvosanaRajat[1] = lapaisyraja;

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

/// <summary>
/// Muuttaa annetun taulukon merkkijonoksi.
/// Erotinmerkkinä toimii pilkku + välilyönti (", ").
/// </summary>
/// <param name="taulukko">Jonoksi muutettava taulukko.</param>
/// <returns>Taulukko merkkijonona.</returns>
/// <example>
/// <pre name="test">
/// double[] taulukko1 = new double[] {15.4, 20, 1.0, 5.9, -2.4};
/// Arvosanat.TaulukkoJonoksi(taulukko1) === "15.4, 20, 1, 5.9, -2.4";
/// </pre>
/// </example>
public static String TaulukkoJonoksi(double[] taulukko)
{
    String erotin = "";
    StringBuilder jono = new StringBuilder();
    foreach (double luku in taulukko)
    {
        jono.Append(erotin);
        jono.Append(luku.ToString(CultureInfo.InvariantCulture));
        erotin = ", ";
    }
    return jono.ToString();
}

/// <summary>
/// Laskee opiskelijan tenttiarvosanan asteikoilla 0-5.
/// </summary>
/// <param name="pisterajat">Arvosanojen rajat taulukossa.
/// Arvosanan 1 raja taulukon indeksissä 1 jne. </param>
/// <param name="tenttiPisteet">Saadut tenttipisteet.</param>
/// <returns>Tenttiarvosana välillä [0..5].</returns>
/// <example>
/// <pre name="test">
/// double[] rajat = {0, 12, 14, 16, 18, 20};
/// Arvosanat.LaskeArvosana(rajat, 11) === 0;
/// Arvosanat.LaskeArvosana(rajat, 12) === 1;
/// Arvosanat.LaskeArvosana(rajat, 14) === 2;
/// Arvosanat.LaskeArvosana(rajat, 22) === 5;
/// Arvosanat.LaskeArvosana(rajat, 28) === 5;
/// </pre>
/// </example>
public static int LaskeArvosana(double[] pisterajat, int tenttiPisteet)
{
    for (int i = pisterajat.Length - 1; 1 <= i; i--)
    {
        if (pisterajat[i] <= tenttiPisteet) return i;
    }
    return 0;
}

/// <summary>
/// Pääohjelmassa pari esimerkkiä.
/// </summary>
/// <param name="args">Ei käytössä.</param>
public static void Main(String[] args)
{
```

```
        double[] pisterajat = LaskeRajat(24, 12);
        Console.WriteLine(LaskeArvosana(pisterajat, 12)); // tulostaa 1
        Console.WriteLine(LaskeArvosana(pisterajat, 20)); // tulostaa 4
        Console.WriteLine(LaskeArvosana(pisterajat, 11)); // tulostaa 0
        Console.ReadKey();
    }
}
```

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.

Ohjelmassa on testejä varten tehty yksi apufunktio, `TaulukkoJonoksi`, jonka tehtävä on palauttaa taulukon alkiot yhtenä merkkijonona perättäin lueteltuna pilkulla erotettuna.

16.5 foreach-silmukka

Taulukoita käsiteltäessä voidaan käyttää myös `foreach`-silmukkaa. Se on eräänlainen paranneltu versio `for`-silmukasta. 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.

```
foreach (taulukonAlkionTyyppi alkio in taulukko)
{
   lauseet;
}
```

Nyt `foreach`-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 `foreach`-silmukalle pitää antaa sen taulukon nimi, mitä halutaan käsitellä. Huomaa, että tiedot erotetaan `foreach`-silmukassa kaksoispisteellä. Tämä erottaa `foreach`-silmukan `for`-silmukasta. Esimerkiksi `kuukausienPaivienLkm`-taulukon alkioita voisi nyt tulostaa seuraavasti.

```
foreach (int kuukaudenPituus in kuukausienPaivienLkm)
{
   Console.Write(kuukaudenPituus + " ");
}
```

Yllä oleva `foreach`-silmukka voitaisiin lukea seuraavasti: "For each kuukausi in kuukausienPaivienLkm...". Vapaasti suomennettuna

"Jokaiselle kuukaudelle kuukausienPaivienLkm-taulukossa (tee)...".

16.5.1 Esimerkki: taulukon pallot keltaisiksi

Esimerkissä 16.4.2 värjäsimme lumiukon pallot keltaisiksi. Koska halusimme muuttaa *kaikkien* taulukossa olevien olioiden värin, on luontevampaa käyttää tehtävään `foreach`-silmukkaa. `LuoPallo`-aliohjelma on sama kuin esimerkissä 16.4.2 ja jätetty listauksesta pois.

```
using Jypeli;
using System;

/// <summary>
```

```
/// Lumiukko, jonka pallot ovat taulukossa.
/// </summary>
public class Lumiukko : PhysicsGame
{
    /// <summary>
    /// Piirretään oliot ja zoomataan kamera niin että kenttä näkyy kokonaan.
    /// </summary>
    public override void Begin()
    {
        Camera.ZoomToLevel();
        Level.BackgroundColor = Color.Black;

        PhysicsObject[] pallot = new PhysicsObject[3];
        pallot[0] = LuoPallo(0, Level.Bottom + 200, Color.White, 100);
        pallot[1] = LuoPallo(0, pallot[0].Y + 100 + 50, Color.White, 50);
        pallot[2] = LuoPallo(0, pallot[1].Y + 50 + 30, Color.White, 30);
        Add(pallot[0]); Add(pallot[1]); Add(pallot[2]);

        foreach (PhysicsObject pallo in pallot)
        {
            pallo.Color = Color.Yellow;
        }
    }

    /// <summary> ...
    public static PhysicsObject LuoPallo(double x, double y, Color vari, double sade) ...
}
```

16.6 Sisäkkäiset silmukat

Kaikkia silmukoita voi kirjoittaa myös toisten silmukoiden sisälle. Sisäkkäisiä silmukoita tarvitaan ainakin silloin, kun halutaan tehdä jotain moniulotteisille taulukoille. Luvussa 15.5 määrittelimme kaksiulotteisen taulukon elokuvien tallentamista varten. Tulostetaan nyt sen sisältö käyttämällä kahta for-silmukkaa.

```
String[,] elokuvat = new String[3,3]
{
   {"Pulp Fiction", "Toiminta", "Tarantino"},
   {"2001: Avaruusseikkailu", "Scifi", "Kubrick"},
   {"Casablanca", "Draama", "Curtiz"}
};

for (int r = 0; r < 3; r++) // rivit
{
   for (int s = 0; s < 3; s++) // sarakkeet
   {
      Console.Write(elokuvat[r, s] + " | ");
   }
   Console.WriteLine();
}
```

Ulommassa `for`-silmukassa käydään läpi taulukon jokainen rivi, eli eri elokuvat. Kun elokuva on ”valittu”, käydään elokuvan tiedot läpi. Sisemmässä `for`-silmukassa käydään läpi aina kaikki yhden elokuvan tiedot. Tietyn elokuvan eri tiedot tai kentät on tässä päätetty erottaa "|"-merkillä. Sisemmän `for`-silmukan jälkeen tulostetaan vielä rivinvaihto `Console.WriteLine()`-metodilla. Näin eri elokuvat saadaan eri riveille.

Tässä täytyy ottaa huomioon, että ulomman taulukon indeksejä käydään läpi eri muuttujalla kuin sisempää taulukkoa. Usein on tapana kirjoittaa ensimmäisen (ulomman) indeksimuuttujan nimeksi `i`, ja seuraavan nimeksi `j`. Samannimisiä muuttujia ei voi käyttää, sillä ne ovat nyt samalla näkyvyysalueella. Tässä kyseisessä esimerkissä on loogisempaa käyttää riveihin ja sarakkeisiin viittaavia indeksien nimiä – siis `r` ja `s`.

16.7 Esimerkki: rivi, jolla eniten vapaata tilaa

Palataan vielä aikaisempaan laivanupotusesimerkkiin. Tehdään aliohjelma, joka etsii 2-ulotteisen taulukon riveistä sen, jolla on eniten tyhjää (eli missä on eniten tilaa laittaa uusi laiva).

```
/// <summary>
/// Aliohjelma palauttaa sen rivin (indeksin),
/// millä on eniten vapaata (eli rivi, jolla
/// eniten 0-alkioita). Jos näitä rivejä on useita, niin
/// palautetaan niistä ensimmäinen.
/// </summary>
/// <param name="ruudut">taulukko ruuduista, kussakin
/// ruudussa 0 = vapaa, muu numero = laivan numero.</param>
/// <returns>Rivi, jolla eniten vapaata.</returns>
/// <example>
/// <pre name="test">
/// int[,] ruudut1 = {{1, 0, 2}, {0, 0, 2}, {3, 3, 3}};
/// Laivanupotus.RiviJollaEnitenVapaata(ruudut1) === 1;
/// int[,] ruudut2 = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};
/// Laivanupotus.RiviJollaEnitenVapaata(ruudut2) === 0;
/// int[,] ruudut3 = {{1, 1, 0}, {0, 0, 2}, {0, 3, 0}};
/// Laivanupotus.RiviJollaEnitenVapaata(ruudut3) === 1;
/// </pre>
/// </example>
public static int RiviJollaEnitenVapaata(int[,] ruudut)
{
  int riviJollaEnitenVapaata = 0;
  int enitenVapaitaMaara = 0;

  for (int r = 0; r < ruudut.GetLength(0); r++)
  {
    int vapaata = 0;
    for (int s = 0; s < ruudut.GetLength(1); s++)
    {
      if (ruudut[r, s] == 0) vapaata++;
    }
    if (vapaata > enitenVapaitaMaara)
    {
      riviJollaEnitenVapaata = r;
      enitenVapaitaMaara = vapaata;
    }
  }
  return riviJollaEnitenVapaata;
}
```

16.8 Silmukan suorituksen kontrollointi break- ja continue-lauseilla

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

16.8.1 break

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

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

`break`-lauseen käyttö voi kuitenkin olla järkevää, jos kesken silmukan todetaan, että silmukan jatkaminen on syytä lopettaa.

16.8.2 continue

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

```
// Tulostetaan luku vain jos se on pariton
for (int i = 0; i < 100; i++)
{
   if (i % 2 == 0) continue;
   Console.WriteLine(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`-rakennetta voi ja kannattaa pyrkiä välttämään, samoin kuin turhia `if`-rakenteita. Yllä olevan ohjelmanpätkän voisi kirjoittaa vaikka seuraavasti.

```
for (int i = 0; i < 100; i++)
{
   if (i % 2 != 0) Console.WriteLine(i);
}
```

Tai vielä yksinkertaisemmin seuraavasti.

```
for (int i = 1; i < 100; i += 2)
{
   Console.WriteLine(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.

16.9 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 C#-kielestä löydy valmiina. Tämän rakenteen voi kuitenkin tehdä itse, jolloin on perusteltua käyttää hallittua ikuista silmukkaa, joka lopetetaan `break`-lauseella. Rakenne voisi olla suunnilleen seuraavanlainen:

```
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.

16.10 Yhteenveto

Silmukan valinta:

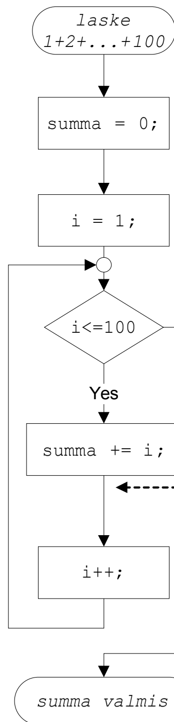
- `for`: Jos silmukan kierrosten määrä on ennalta tiedossa
- `foreach`: Jos haluamme tehdä jotain jonkun `Collection`-tietorakenteen tai taulukon kaikille alkioille.
- `while`: Jos silmukan kierrosten määrä ei ole tiedossa (erikoistapauksena hallittu ikuinen silmukka, josta poistutaan `break`-lauseella), emmekä välttämättä halua suorittaa silmukkaa kertaakaan.
- `do-while`: Jos silmukan kierrosten määrä ei ole tiedossa, mutta haluamme suorittaa silmukan vähintään yhden kerran.

Seuraava kuva kertaa vielä kaikki C#:n valmiit silmukat:

C#:n silmukkarakenteet

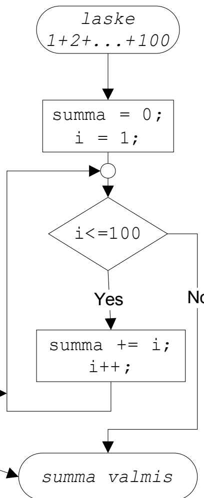
for

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



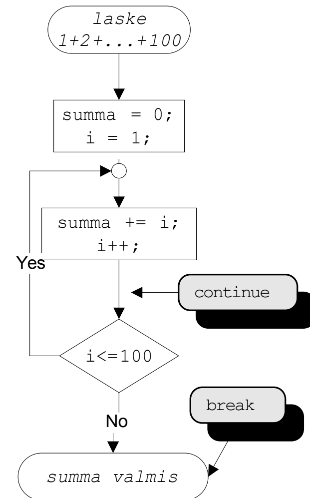
while

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



do-while

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



continue

break

```
/* 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. taulukoiden tapauksessa
- **while**-silmukka valitaan jos for ei ole ilmeinen ja runkoa mahdollisesti ei suoriteta
- **do-while** -valitaan, mikäli runko on suoritettava vähintään kerran
- joskus siistein rakenne saadaan ikuisella silmukalla (ehto esim. true)

Kuva 27: C#:n silmukat

17. Merkkijonojen pilkkominen ja muokkaaminen

17.1 String.Split()

Merkkijonoja voidaan pilkkoa `String`-olion `Split`-metodilla. Metodi palauttaa palaset merkkijonotyyppisessä taulukossa `String[]`. Nyt `Split()`-metodille annetaan parametrina taulukko niistä merkeistä (`Char`), joiden halutaan toimivan erotinmerkkeinä. Pyydetään käyttäjältä syöte, ja oletetaan, että haluamme, että välilyönti, puolipiste ja pilkku toimivat erotinmerkkeinä.

```
char[] erottimet = new char[] { ' ', ';', ',' };
Console.Write("Kirjoita > ");
String jono1 = Console.ReadLine();
String[] pilkottu = jono1.Split(erottimet);
```

Merkkijono katkaistaan, jos merkki on välilyönti, pilkku tai puolipiste. Jos käyttäjä antaa useamman erotinmerkin peräkkäin (vaikkapa kaksi välilyöntiä), niin voi olla toivottavaa, ettei kuitenkaan taulukkoon luoda tyhjää alkiota. Tämä voidaan hoitaa antamalla `Split()`-metodille lisäparametri `StringSplitOptions.RemoveEmptyEntries`.

Jättämällä tyhjät alkiot huomiotta esimerkiksi merkkijono ”`kissa,,,; koira`” palauttaisi vain kaksialkioisen taulukon:

```
String jono2 = "kissa,,,; koira";
char[] erottimet = new char[] { ' ', ';', ',' };
String[] pilkottu = jono2.Split(erottimet,
  StringSplitOptions.RemoveEmptyEntries);
// Palat olisivat nyt ["kissa", "koira"]
```

Huomaa, että erotinmerkit eivät tule mukaan taulukkoon, vaan ne "häviävät".

17.2 String.Trim()

String-olion Trim()-metodi palauttaa merkkijonon, josta on poistettu välilyönnit parametrina annetun merkkijonon alusta ja lopusta. Esimerkiksi merkkijono

```
String jono3 = "  kalle   ja kille    ";
```

muuttuisi muotoon

```
Console.WriteLine(jono3 .Trim()); // "kalle   ja kille".
```

Huomaa, että merkkijonon keskellä olevia "ylimääräisiä" välilyöntejä `Trim`-metodi ei kuitenkaan poista.

17.3 Esimerkki: Merkkijonon pilkkominen ja muuttaminen kokonaisluvuiksi

Tehdään ohjelma, joka kysyy käyttäjältä positiivisia kokonaislukuja, laskee ne yhteen ja tulostaa tuloksen näytölle. Käyttäjä antaa luvut siten, että välilyönti ja pilkku toimivat erotinmerkkeinä. Mikäli käyttäjä antaa jotain muita merkkejä kuin positiivisia kokonaislukuja (ja erotinmerkkejä), ohjelma antaa virheilmoituksen ja suoritus päättyy. Ohjelmassa tehdään seuraavat aliohjelmat.

`int[] MerkkijonoLuvuiksi(String, char[])`

Aliohjelma muuttaa annetun merkkijonon kokonaislukutaulukoksi siten, etta luvut erotellaan annetun merkkitaulukon (erotinmerkkien) perusteella. Syötteen tulee sisältää

vain lukuja ja erotinmerkkejä.

`int LaskeYhteen(int[])`
Palauttaa annetun kokonaislukutaulukon alkioiden summan.

`bool OnkoVainLukuja(String, char[])`
Tutkii sisältääkö annettu merkkijono vain lukuja (positiivisia kokonaislukuja) ja erotinmerkkejä. Jos annettu merkkijono on tyhjä (pituus on 0), palautetaan `false`.

`void TulostaTaulukko(int[])`
Tulostaa annetun kokonaislukutaulukon kaikki alkiot.

```
using System;

/// @author  Antti-Jussi Lakanen
/// @version 22.12.2011

/// <summary>
/// Harjoitellaan merkkijonojen pilkkomista.
/// </summary>
public class MjLuvuiksi
{
    /// <summary>
    /// Kysellaan kayttajalta merkkijonoja ja
    /// tehdaan niista taulukkoja, lasketaan lukuja yhteen ja tulostellaan.
    /// </summary>
    /// <param name="args"></param>
    public static void Main(string[] args)
    {
        char[] erottimet = new char[] { ' ', ',' };
        Console.Write("Anna positiivisia kokonaislukuja > ");
        String lukusyote = Console.ReadLine();

        // Jos kayttaja antanut jotain muuta kuin positiivisia
        // kokonaislukuja, ei yritetakaan laskea lukuja yhteen
        if (OnkoVainLukuja(lukusyote, erottimet))
        {
            int[] luvut = MerkkijonoLuvuiksi(lukusyote, erottimet);
            Console.WriteLine("Tulkittiin luvut:");
            TulostaTaulukko(luvut);
            Console.WriteLine("Antamiesi lukujen summa on : " + LaskeYhteen(luvut));
        }
        else
            Console.WriteLine("Annoit muuta kuin lukuja, tai tyhjan jonon");
        Console.ReadKey();
    }

    /// <summary>
    /// Aliohjelma muuttaa annetun merkkijonon
    /// kokonaislukutaulukoksi siten, etta luvut
    /// erotellaan annetun merkkitaulukon (erotinmerkkien)
    /// perusteella. Syötteen tulee sisältää vain
    /// lukuja ja erotinmerkkejä.
    /// </summary>
    /// <param name="lukusyote">Muunnettava merkkijono</param>
    /// <param name="erottimet">Sallitut erotinmerkit merkkitaulukossa</param>
    /// <returns>Merkkijonosta selvitetty kokonaislukutaulukko.</returns>
    /// <example>
    /// <pre name="test">
    /// int[] luvut1 = MjLuvuiksi.MerkkijonoLuvuiksi("1 2 3", new char[] {' '});
    /// String.Join(",", luvut1) === "1,2,3";
    /// int[] luvut2 = MjLuvuiksi.MerkkijonoLuvuiksi("  ,,1,, 2 ,  3", new char[] {' ',
',');
    /// String.Join(",", luvut2) === "1,2,3";
    /// int[] luvut3 = MjLuvuiksi.MerkkijonoLuvuiksi("", new char[] {' '});
    /// String.Join(",", luvut3) === "";
    /// </pre>
```

```
    /// </example>
    public static int[] MerkkijonoLuvuiksi(string lukusyote, char[] erottimet)
    {
        // Tyhjat pois edesta ja lopusta (Trim)
        // Jos on annettu ylimaaraisia valilyonteja, ei lisata niita taulukkoon.
        String[] pilkottu = lukusyote.Trim().Split(erottimet,
StringSplitOptions.RemoveEmptyEntries);
        int[] luvut = new int[pilkottu.Length]; // luvut[] saa kookseen saman kuin
pilkottu[]
        for (int i = 0; i < pilkottu.Length; i++)
            luvut[i] = int.Parse(pilkottu[i]);
        return luvut;
    }

    /// <summary>
    /// Laskee kokonaislukutaulukon alkiot yhteen ja palauttaa alkioiden summan.
    /// </summary>
    /// <param name="luvut">Tutkittava kokonaislukutaulukko</param>
    /// <returns>Taulukon alkioiden summa</returns>
    /// <example>
    /// <pre name="test">
    /// int[] luvut1 = {5, 7, 9, 10};
    /// MjLuvuiksi.LaskeYhteen(luvut1) === 31;
    /// int[] luvut2 = {-5, 5, -10, 10};
    /// MjLuvuiksi.LaskeYhteen(luvut2) === 0;
    /// int[] luvut3 = {};
    /// MjLuvuiksi.LaskeYhteen(luvut3) === 0;
    /// </pre>
    /// </example>
    public static int LaskeYhteen(int[] luvut)
    {
        int summa = 0;
        for (int i = 0; i < luvut.Length; i++)
            summa += luvut[i];
        return summa;
    }

    /// <summary>
    /// Aliohjelmassa tutkitaan sisaltaako merkkijono muitakin
    /// merkkeja kuin positiivisia kokonaislukuja ja erotinmerkkeja.
    /// </summary>
    /// <param name="lukusyote">Tutkittava merkkinojo,
    /// josta etsitaan vieraita merkkeja</param>
    /// <param name="erottimet">Sallitut erotinmerkit
    /// merkkitaulukossa</param>
    /// <returns>Onko pelkkiä lukuja</returns>
    /// <example>
    /// <pre name="test">
    /// MjLuvuiksi.OnkoVainLukuja("1,2,3", new char[]{','}) === true;
    /// MjLuvuiksi.OnkoVainLukuja("1, 2, 3", new char[]{','}) === false;
    /// MjLuvuiksi.OnkoVainLukuja("1, 2, 3", new char[]{',', ' '}) === true;
    /// MjLuvuiksi.OnkoVainLukuja("", new char[]{' '}) === false;
    /// </pre>
    /// </example>
    public static bool OnkoVainLukuja(string lukusyote, char[] erottimet)
    {
        // Jos yhtaan merkkia ei ole annettu,
        // palautetaan automaattisesti kielteinen vastaus.
        if (lukusyote.Length == 0) return false;
        for (int i = 0; i < erottimet.Length; i++)
            // Korvataan erotinmerkit tyhjalla merkkijonolla,
            // silla olemme kiinnostuneita vain "varsinaisesta sisallosta"
            lukusyote = lukusyote.Replace(erottimet[i].ToString(), "");

        foreach (char merkki in lukusyote)
            // Jos yksikin merkki on jokin muu kuin numero,
            // palautetaan kielteinen vastaus.
            if (!Char.IsDigit(merkki)) return false;
        return true;
```

```
    }

    /// <summary>
    /// Tulostetaan kokonaislukutaulukon osat foreach-silmukassa
    /// </summary>
    /// <param name="t">Tulostettava taulukko</param>
    public static void TulostaTaulukko(int[] t)
    {
        foreach (int pala in t)
            Console.WriteLine(pala);
        Console.WriteLine("-------------------------------");
    }
}
```

18. 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 todellisuudessa kymmeniä, ellei satoja.

Järjestämisalgoritmeja käsitellään enemmän muilla kursseilla (esim. ITKA201 Algoritmit 1 ja TIEP111 Ohjelmointi 2). Tässä vaiheessa meille riittää, että osaamme käyttää C#:sta 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 `Array`-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};
Array.Sort(taulukko);

// Tulostetaan alkiot, että nähdää onnistuiko järjestäminen.
foreach (int alkio in taulukko)
{
   Console.WriteLine(alkio);
}
```

Alkioiden pitäisi nyt tulostua numerojärjestyksessä. Taulukko voitaisiin myös järjestää vain osittain antamalla `Sort`-aliohjelmalle lisäksi parametreina aloitusindeksi, sekä järjestettävien alkioiden määrä.

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

// järjestetään nyt vain kolme ensimmäistä alkiota
Array.Sort(taulukko2, 0, 3);

foreach (int alkio in taulukko)
{
   Console.WriteLine(alkio);
}

// Tulostuu -4 -2 4 5 5 12 9
```

Kaikkia alkeistietotyyppisiä taulukoita voidaan järjestää `Sort`-aliohjelmalla. Lisäksi voidaan järjestää taulukoita, joiden alkioiden tietotyyppi *toteuttaa* (**implements**) `IComparable`-rajapinnan. Esimerkiksi `String`-luokka toteuttaa tuon rajapinnan. Rajapinnoista puhutaan lisää kohdassa 23.1.

19. Olion ulkonäön muuttaminen (Jypeli)

Olemme tähän mennessä käyttäneet jo monia Jypeli-kirjastoon kirjoitettuja luokkia ja aliohjelmia. Tässä luvussa esitellään muutamia yksittäisiä tärkeitä luokkia, aliohjelmia ja ominaisuuksia.

Luodaan ensin olio, jonka ulkonäköä esimerkeissä muutetaan.

```
PhysicsObject palikka = new PhysicsObject(100, 50);
```

Olion on suorakulmio, jonka leveys on 100 ja korkeus 50. Jos haluat olion näkyviin pelikentälle, muista aina lisätä se seuraavalla lauseella.

```
Add(palikka);
```

19.1 Väri

Vaihdetaan seuraavaksi luomamme olion väri. Värin voi vaihtaa seuraavalla tavalla:

```
palikka.Color = Color.Gray;
```

Esimerkissä siis oliosta tehtiin harmaa. Värejä on valmiina paljon ja niistä voi valita haluamansa. Voit esikatsella valmiita värejä osoitteesta

https://trac.cc.jyu.fi/projects/npo/wiki/OlionUlkonako#a2.Väri.

Omia värejä voi myös tehdä seuraavasti:

```
palikka.Color = new Color( 0, 0, 0 );
```

Oliosta tuli musta. Ensimmäinen arvo kertoo punaisen värin määrän, toinen arvo vihreän värin määrän ja kolmas sinisen värin määrän. "Värimaailman" lyhenne RGB (**R**ed, **G**reen, **B**lue) tulee tästä. Lyhenteestä on helppo muistaa missä järjestyksessä värit tulevat. Määrät ovat kokonaislukuja välillä 0-255 (`byte`). Muitakin tapoja värien asettamiseen on olemassa, mutta näillä kahdella pärjää jo hyvin.

19.2 Koko

Kokoa voi vaihtaa seuraavasti.

```
palikka.Width = leveys;
palikka.Height = korkeus;
```

Leveys ja korkeus annetaan `double`-tyyppisinä lukuina. Saman asian voi tehdä myös vektorin avulla yhdellä rivillä.

```
palikka.Size = new Vector(leveys, korkeus);
```

19.3 Tekstuuri

Tekstuurikuvat kannattaa tallentaa `png`-muodossa, jolloin kuvaan voidaan tallentaa myös alpha-kanavan tieto (läpinäkyvyys). Tallenna `png`-kuva projektin `Content`-kansioon. Klikkaa sitten Visual Studion Solution Explorerissa projektin nimen päällä hiiren oikealla napilla ja Add -> Existing item. Hae kansiorakenteesta juuri tallentamasi kuva.

Tämän jälkeen tekstuuri asetetaan kuvalle seuraavasti.

```
Image olionKuva = LoadImage("kuvanNimi");
olio.Image = olionKuva;
```

Huomaa, että `png`-tunnistetta ei tarvitse laittaa kuvan nimen perään.

Saman voi tehdä myös lyhyemmin:

```
palikka.Image = LoadImage("kuvanNimi");
```

Kohta `kuvanNimi` on `Contentiin` siirretyn kuvan nimi esimerkiksi, jos kuva on `kissa.png`, niin kuvan nimi on silloin pelkkä `kissa`.

19.4 Olion muoto

Joskus olion muodon voi antaa jo oliota luotaessa. Muotoa voi kuitenkin myös jälkikäteen muuttaa. Esimerkiksi:

```
olio.Shape = Shape.Circle;
```

Tämä tekee oliostamme ympyrän muotoisen. Muita mahdollisia muotoja on esimerkiksi nelikulmio, `Shape.Rectangle`.

20. Ohjainten lisääminen peliin (Jypeli)

Peli voi ottaa vastaan näppäimistön, Xbox 360 -ohjaimen, hiiren ja Windows Phone 7 -puhelimen ohjausta. Ohjainten liikettä "kuunnellaan" ja jokaiselle ohjaimelle voidaan määrittää erikseen, mitä mistäkin tapahtuu. Kullekin ohjaimelle (näppäimistö, hiiri, Xbox-ohjain, WP7-kosketusnäyttö, WP7-kiihtyvyysanturi) on tehty oma `Listen`-aliohjelma jolla kuuntelun asettaminen onnistuu.

Jokainen Listen-kutsu on muodoltaan samanlainen riippumatta siitä mitä ohjainta kuunnellaan. Ensimmäinen parametri kertoo mitä näppäintä kuunnellaan, esimerkiksi:

```
Näppäimistö: Key.Up
Xbox360-ohjain: Button.DPadLeft
Hiiri: MouseButton.Left
```

Visual Studion kirjoitusapu auttaa löytämään mitä erilaisia näppäinvaihtoehtoja kullakin ohjaimella on.

Toinen parametri määrittää minkälaisia näppäinten tapahtumia halutaan kuunnella ja sillä on neljä mahdollista arvoa:

- ButtonState.Released: Näppäin on juuri vapautettu
- ButtonState.Pressed: Näppäin on juuri painettu alas
- ButtonState.Up: Näppäin on ylhäällä (vapautettuna)
- ButtonState.Down: Näppäin on alaspainettuna

Kolmas parametri kertoo mitä tehdään, kun näppäin sitten on painettuna. Tähän tulee tapahtuman käsittelijä, eli sen aliohjelman nimi, jonka suoritukseen haluamme siirtyä näppäimen tapahtuman sattuessa.

Neljäs parametri on ohjeteksti, joka voidaan näyttää pelaajalle pelin alussa. Tässä tarvitsee vain kertoa mitä tapahtuu kun näppäintä painetaan. Ohjetekstin tyyppi on `String` eli merkkijono. Merkkijono on jono kirjoitusmerkkejä tietokoneen muistissa. Merkkijonoilla voimme esittää mm. sanoja ja lauseita. Jos ohjetta ei halua tai tarvitse laittaa, neljännen parametrin arvoksi voi antaa null jolloin se jää tyhjäksi.

Parametrejä voi antaa enemmänkin sen mukaan mitä pelissä tarvitsee. Omat (eli valinnaiset) parametrit laitetaan edellä mainittujen pakollisten parametrien jälkeen ja ne viedään automaattisesti Listen-kutsussa annetulle käsittelijälle. Tästä esimerkki hetken kuluttua.

Esimerkki näppäimistön kuuntelusta:

```
Keyboard.Listen(Key.Left, ButtonState.Down,
  LiikutaPelaajaaVasemmalle, "Liikuta pelaajaa vasemmalle");
```

Kun vasen (`Key.Left`) näppäin on alhaalla (`ButtonState.Down`), niin liikutetaan pelaajaa suorittamalla metodi `LiikutaPelaajaaVasemmalle`. Viimeisenä parametrina on pelissä näkyvä näppäinohjeteksti.

Vastaava esimerkki Xbox 360 -ohjaimen kuuntelusta:

```
ControllerOne.Listen(Button.DPadLeft, ButtonState.Down, LiikutaPelaajaaVasemmalle,
"Liikuta pelaajaa vasemmalle");
```

Yhtäaikaisesti voidaan kuunnella jopa neljää XBox-ohjainta. Tässä kuunnellaan ohjaimista

ensimmäistä (`ControllerOne`). Muut ohjaimet ovat `ControllerTwo` ja niin edelleen. Kunkin ohjaimen järjestysluku näkyy ohjaimen keskellä olevassa Xbox-kuvakkeessa, missä erityinen valo indikoi, mikä ohjain on kysymyksessä.

20.1 Näppäimistö

Tässä esimerkissä asetetaan näppäimistön nuolinäppäimet liikuttamaan pelaajaa.

```
using System;
using Jypeli;

/// <summary>
/// Peli, jossa liikutellaan palloa.
/// </summary>
public class Peli : PhysicsGame
{
    /// <summary>
    /// Luodaan pelaaja ja asetetaan näppäintenkuuntelijat
    /// </summary>
    public override void Begin()
    {
        PhysicsObject pelaaja = new PhysicsObject(50, 50, Shape.Circle);
        Add(pelaaja);
        Keyboard.Listen(Key.Left, ButtonState.Down,
            LiikutaPelaajaa, "Liikuta vasemmalle",
            pelaaja, new Vector(-1000, 0));
        Keyboard.Listen(Key.Right, ButtonState.Down,
            LiikutaPelaajaa, "Liikuta oikealle", pelaaja, new Vector(1000, 0));
        Keyboard.Listen(Key.Up, ButtonState.Down,
            LiikutaPelaajaa, "Liikuta ylös", pelaaja, new Vector(0, 1000));
        Keyboard.Listen(Key.Down, ButtonState.Down,
            LiikutaPelaajaa, "Liikuta alas", pelaaja, new Vector(0, -1000));
    }

    /// <summary>
    /// Aliohjelmassa liikutetaan
    /// oliota "työntämällä".
    /// </summary>
    /// <param name="suunta">Mihin suuntaan</param>
    private void LiikutaPelaajaa(PhysicsObject olio, Vector suunta)
    {
        olio.Push(suunta);
    }
}
```

Tapahtumankäsittelijän `LiikutaPelaajaa` parametreista `Physicsobject olio` ja `Vector suunta` saadaan tiedot, *mitä oliota* halutaan liikuttaa ja *mihin suuntaan*. Huomaa, että nämä tiedot annetaan kutsuvaiheessa ”ylimääräisinä parametreina”, eli `Keyboard.Listen`-riveillä.

20.2 Lopetuspainike ja näppäinohjepainike

Pelin lopettamiselle ja näppäinohjeen näyttämiselle ruudulla on Jypelissä olemassa valmiit aliohjelmat. Ne voidaan asettaa näppäimiin seuraavasti:

```
Keyboard.Listen(Key.Escape, ButtonState.Pressed, Exit, "Poistu");
Keyboard.Listen(Key.F1, ButtonState.Pressed, ShowControlHelp, "Näytä ohjeet");
```

Tässä näppäimistön **Esc**-painike lopettaa pelin ja **F1**-painike näyttää ohjeet.

`ShowControlHelp` näyttää peliruudulla pelissä käytetyt näppäimet ja niille asetetut ohjetekstit. Ohjeteksti on `Listen`-kutsun neljäntenä parametrina annettu merkkijono.

20.3 Peliohjain

Sama esimerkki XBox-peliohjainta käyttäen voidaan tehdä korvaamalla rivit

```
Keyboard.Listen(...);
```

riveillä

```
ControllerOne.Listen(...);
```

esimerkiksi näin

```
ControllerOne.Listen(Button.DPadLeft, ButtonState.Down, LiikutaPelaajaa, "Liikuta
vasemmalle", pelaaja, new Vector(-1000, 0));
```

`LiikutaPelaajaa`-aliohjelmaan sen sijaan ei tarvitse tehdä muutoksia, joten sama aliohjelma kelpaa sekä näppäimen että Xbox-ohjaimen "digipad"-napin kuunteluun.

20.3.1 Analoginen ”tatti”

Jos halutaan kuunnella ohjaimen tattien liikettä, käytetään `ListenAnalog`-kutsua.

```
ControllerOne.ListenAnalog(AnalogControl.LeftStick, 0.1,
LiikutaPelaajaa, "Liikuta pelaajaa tattia pyörittämällä.");
```

Kuunnellaan vasenta tattia (`AnalogControl.LeftStick`). Luku 0.1 kuvaa sitä, miten herkästä liikkeestä tattia kuunteleva aliohjelma suoritetaan. Kuuntelua käsittelee aliohjelma `LiikutaPelaajaa`.

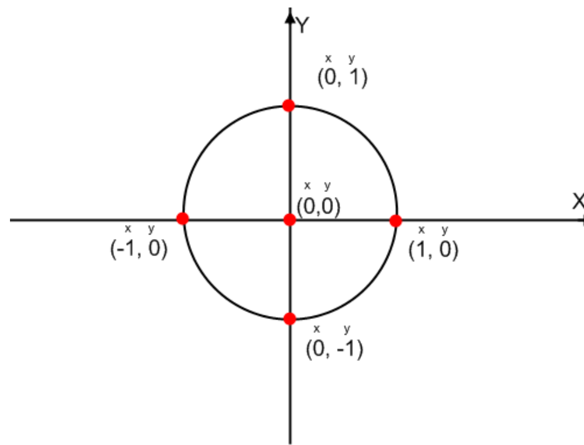
`LiikutaPelaajaa`-aliohjelman tulee ottaa vastaan seuraavanlainen parametri:

```
private void LiikutaPelaajaa(AnalogState tatinTila)
{
  // Liikutellaan
}
```

Tatin asento saadaan selville parametrina vastaan otettavasta `AnalogState`-tyyppisestä muuttujasta:

```
private void LiikutaPelaajaa(AnalogState tatinTila)
{
  Vector tatinAsento = tatinTila.StateVector;
  // Tehdään jotain tatin asennolla, esim liikutetaan pelaajaa...
}
```

`StateVector` antaa siis vektorin, joka kertoo mihin suuntaan tatti osoittaa. Vektorin X ja Y -koordinaattien arvot ovat molemmat väliltä miinus yhdestä yhteen (-1 - 1) tatin suunnasta riippuen. Tämän vektorin avulla voidaan esimerkiksi kertoa pelaajalle mihin suuntaan sen kuuluu liikkua.



Kuva 28: Yksikköympyrä.

Tatin asennon tietyllä hetkellä saa selville myös ilman jatkuvaa tatin kuuntelua kirjoittamalla:

```
Vector tatinAsento = ControllerOne.LeftThumbDirection;
```

Tämä palauttaa samoin vektorin tatin sen hetkisestä asennosta (X ja Y väliltä -1, 1).

Myös Xbox-ohjaimen liipaisimia voidaan kuunnella. Lue lisää ohjewikistä https://trac.cc.jyu.fi/projects/npo/wiki/OhjaintenLisays.

20.4 Hiiri

20.4.1 Näppäimet

Hiiren näppäimiä voi kuunnella aivan samaan tapaan kuin näppäimistön ja Xbox-ohjaimenkin.

```
Mouse.Listen(MouseButton.Left, ButtonState.Pressed,
  Ammu, "Ammu aseella.");
```

Tässä esimerkissä painettaessa hiiren vasenta näppäintä kutsutaan `Ammu`-nimistä aliohjelmaa. Tuo aliohjelma pitää tietenkin erikseen tehdä:

```
private void Ammu()
{
  /// Kirjoita tähän Ammu()-aliohjelman koodi.
}
```

20.4.2 Hiiren liike

Hiirellä ohjauksessa on kuitenkin usein oleellista tietää jotain kursorin sijainnista. Hiiren kursori ei ole oletuksena näkyvä peliruudulla, mutta sen saa halutessaan helposti näkyviin, kun kirjoittaa koodiin seuraavan rivin vaikkapa kentän luomisen yhteydessä:

```
Mouse.IsCursorVisible = true;
```

Hiiren paikan ruudulla saadaan vektorina kirjoittamalla:

```
Vector paikkaRuudulla = Mouse.PositionOnScreen;
```

Tämä kertoo kursorin paikan näyttökoordinaateissa, ts. origo keskellä. Y-akseli kasvaa ylöspäin.

Hiiren paikan pelimaailmassa (peli- ja fysiikkaolioiden koordinaatistossa) voi saada kirjoittamalla

```
Vector paikkaKentalla = Mouse.PositionOnWorld;
```

Tämä kertoo kursorin paikan maailmankoordinaateissa. Origo on keskellä ja Y-akseli kasvaa ylöspäin.

Hiiren liikettä voidaan kuunnella aliohjelmalla `Mouse.ListenMovement`. Sille annetaan parametrina kuuntelun herkkyyttä kuvaava `double`, käsittelijä sekä ohjeteksti. Näiden lisäksi voidaan antaa myös omia parametreja. Käsittelijällä on yksi pakollinen parametri. Esimerkki hiiren kuuntelusta:

```
private PhysicsObject pallo;

public override void Begin()
{
    pallo = new PhysicsObject(30.0, 30.0, Shape.Circle);
    Add(pallo);
    Mouse.IsCursorVisible = true;
    Mouse.ListenMovement(0.1, KuunteleLiiketta, null);
}

private void KuunteleLiiketta(AnalogState hiirenTila)
{
    pallo.X = Mouse.PositionOnWorld.X;
    pallo.Y = Mouse.PositionOnWorld.Y;

    Vector hiirenLiike = hiirenTila.MouseMovement;
}
```

Tässä esimerkissä luomamme fysiikkaolio nimeltä `pallo` seuraa hiiren kursoria. Käsittelijää kutsutaan aina kun hiirtä liikuttaa.

`ListenMovement`:in parametreissa herkkyys (tässä 0.1) tarkoittaa sitä, miten pieni hiiren liike aiheuttaa tapahtuman.

Tapahtumankäsittelijällä on pakollinen `AnalogState`-luokan olio parametrina. Siitä saa myös irti tietoa hiiren liikkeistä. Tässä esimerkissä `hiirenTila.MouseMovement` antaa hiiren liikevektorin, joka kertoo mihin suuntaan ja miten voimakkaasti kursori on liikkunut (hiiren ollessa paikoillaan se on nollavektori).

20.4.3 Hiiren kuunteleminen vain tietyille peliolioille

Jos hiiren painalluksia halutaan kuunnella vain tietyn peliolion (tai fysiikkaolion) kohdalla, voidaan käyttää apuna `Mouse.ListenOn`-aliohjelmaa:

```
Mouse.ListenOn(pallo, MouseButton.Left,
  ButtonState.Down, PoimiPallo, null);
```

Parametrina annetaan se olio, jonka päällä hiiren painalluksia halutaan kuunnella. Muut parametrit ovat kuin normaalissa `Listen`-kutsussa. Käsittelijää `PoimiPallo` kutsutaan tässä esimerkissä silloin, kun hiiren kursori on pallo-nimisen olion päällä ja hiiren vasen nappi on painettuna pohjaan.

Hiirellä on olemassa myös esimerkiksi seuraavanlainen metodi:

```
PhysicsObject kappale = new PhysicsObject(50.0, 50.0);
bool onkoPaalla = Mouse.IsCursorOn(kappale);
```

`Mouse.IsCursorOn` palauttaa totuusarvon `true` tai `false` riippuen siitä, onko kursori sille annetun olion (peli-, fysiikka- tai näyttöolion) päällä.

21. Piirtoalusta (Jypeli)

Piirtoalustalla voidaan peliin piirtää kuvioita. Nämä kuviot ovat siis pelissä näkyviä elementtejä, jotka eivät ole `PhysicsObject`- tai `GameObject`-olioita, vaan ne piirretään ”erillään” peliolioista. Ne eivät noudata fysiikan lakeja. Tällä hetkellä piirtoalustalle voi piirtää janoja.

Piirtämistä varten peliluokkaan lisätään `Paint`-aliohjelma, joka ylikirjoittaa (*override*) kantaluokan vastaavan aliohjelman.

```
protected override void Paint(Canvas canvas)
{
  // Tässä välissä piirretään kuviot
  base.Paint(canvas);
}
```

Jypeli-kirjasto kutsuu `Paint`-aliohjelmaa tasaisin väliajoin (kymmeniä kertoja sekunnissa) pelin ollessa käynnissä. Siinä voi siis toteuttaa animaatioita muuttamalla koordinaatteja sen mukaan millä ajanhetkellä piirretään.

Itse piirtäminen tapahtuu parametrina saatavan `Canvas`-olion metodeilla. Nykyisellään niitä on yksi:

- `DrawLine`: Piirtää janan. Parametrina alku- ja loppupisteen koordinaatit joko vektoreina tai luettelemalla molempien pisteiden x- ja y-koordinaatit.

Väri voidaan asettaa `BrushColor`-ominaisuuden kautta.

Piirtoalueen reunojen koordinaatteja voi lukea samaan tapaan kuin kentänkin reunoja:

```
canvas.Left        Vasemman reunan x-koordinaatti
canvas.Right       Oikean reunan x-koordinaatti
canvas.Bottom      Alareunan y-koordinaatti
canvas.Top         Yläreunan y-koordinaatti
canvas.TopLeft     Vasen ylänurkka
canvas.TopRight    Oikea ylänurkka
canvas.BottomLeft  Vasen alanurkka
canvas.BottomRight Oikea alanurkka
```

Seuraavaksi esimerkkejä.

21.1 Esimerkki: Punainen rasti

Alla oleva esimerkki piirtää punaisen rastin `Canvas`-olion vasempaan ylänurkkaan ja mustan rastin oikeaan ylänurkkaan.

```
protected override void Paint(Canvas canvas)
{
  canvas.BrushColor = Color.Red;
  double x = canvas.Left + 100, y = canvas.Top - 100;
  canvas.DrawLine(new Vector(x - 50, y + 50), new Vector(x + 50, y - 50));
  canvas.DrawLine(new Vector(x + 50, y + 50), new Vector(x - 50, y - 50));

  canvas.BrushColor = Color.Black;
  x = canvas.Right - 100;
  y = canvas.Top - 100;
  canvas.DrawLine(new Vector(x - 50, y + 50), new Vector(x + 50, y - 50));
  canvas.DrawLine(new Vector(x + 50, y + 50), new Vector(x - 50, y - 50));

  base.Paint(canvas);
}
```

Alla kuva lopputuloksesta.

Kuva 29: Punainen ja musta rasti Paint-aliohjelmalla ja Canvas-oliolla piirrettynä.

21.2 Esimerkki: Pyörivä jana

Seuraavassa esimerkissä tehdään satunnaisesti väriään vaihtava jana, joka pyörii alkupisteensä ympäri.

```
protected override void Paint(Canvas canvas)
{
  canvas.BrushColor = RandomGen.NextColor();
  double ajanhetki = Game.Time.SinceStartOfGame.TotalSeconds;
  Vector keskipiste = new Vector(0, 0);
  Vector reunapiste = new Vector(100 * Math.Cos(ajanhetki), 100 * Math.Sin(ajanhetki));
  canvas.DrawLine(keskipiste, keskipiste + reunapiste);
  base.Paint(canvas);
}
```

22. 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 (lopetusehto) return;
   // toimenpiteitä ...
   Rekursio(uudet parametrit);  // Itsensä kutsuminen
}
```

Oleellista on, että rekursiivisessa aliohjelmassa on joku *lopetusehto*. Muutoin aliohjelma kutsuu itseään loputtomasti. Toinen oleellinen seikka on, että seuraavan kutsun, tässä `Rekursio(uudet parametrit)`, 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. Luonnollisesti laitamme mukaan myös ComTest-testit.

```
/// <summary>
/// Lasketaan luvun kertoma kaavasta
/// <code>
/// 0! = 1
/// 1! = 1
/// n! = n*(n-1)!
/// </code>
/// </summary>
/// <param name="n">Minkä luvun kertoma lasketaan</param>
/// <returns>n!</returns>
/// <example>
/// <pre name="test">
/// Rekursio.Kertoma(0) === 1;
/// Rekursio.Kertoma(1) === 1;
/// Rekursio.Kertoma(5) === 120;
/// </pre>
/// </example>
public static long Kertoma(int n)
{
    if (n <= 1) return 1;
    return n * Kertoma(n - 1);
}

/// <summary>
/// Pääohjelma
/// </summary>
/// <param name="args">Ei käytössä</param>
public static void Main(String[] args)
{
    long k = Kertoma(5);
    Console.WriteLine(k);
    Console.ReadLine();
}
```

Aliohjelma `Kertoma` saa parametrikseen luvun, jonka kertoma halutaan laskea. Tutustutaan aliohjelmaan tarkemmin.

```
if (n <= 1) return 1;
```

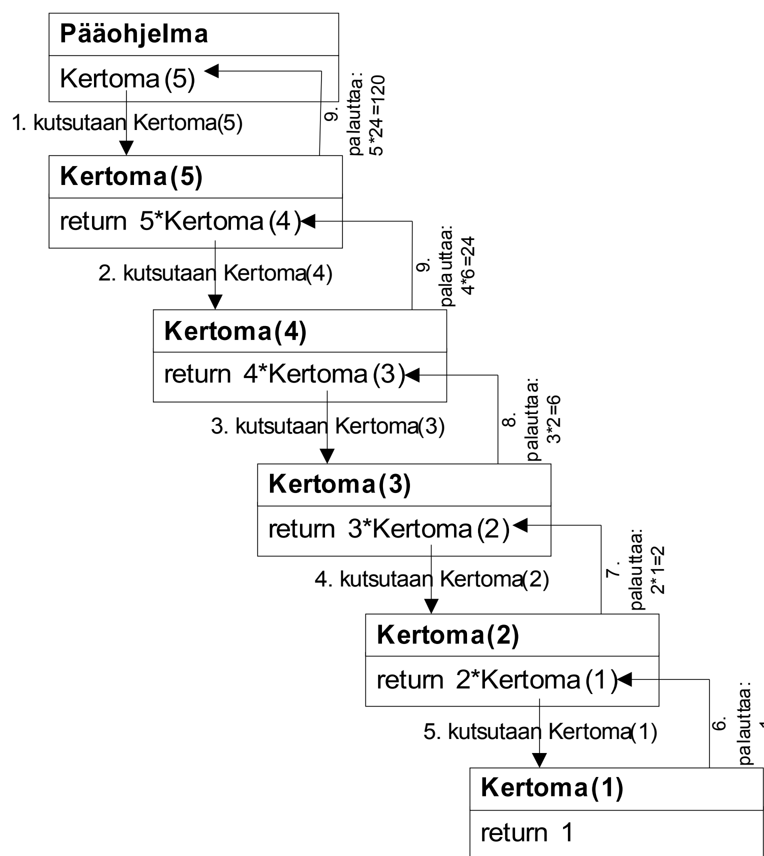
Yllä oleva rivi on ikään kuin rekursion lopetusehto. Jos `n` on pienempää tai yhtä suurta kuin 1, niin palautetaan luku 1. Oleellista on, että lopetusehto on ennen uutta rekursiivista aliohjelmakutsua.

```
return n * Kertoma(n-1);
```

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.



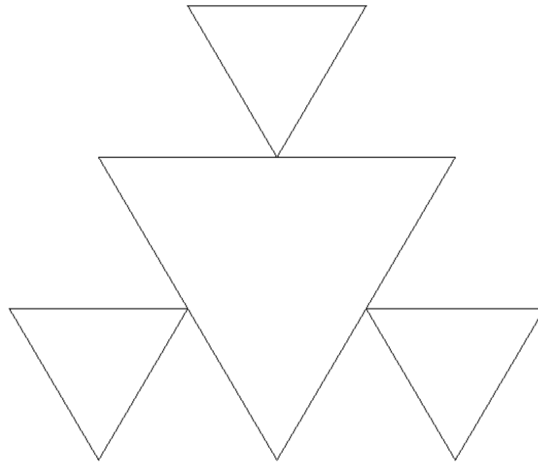
Kuva 30: Kertoman laskeminen rekursiivisesti. Vaiheet numeroitu.

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]

22.1 Sierpinskin kolmio

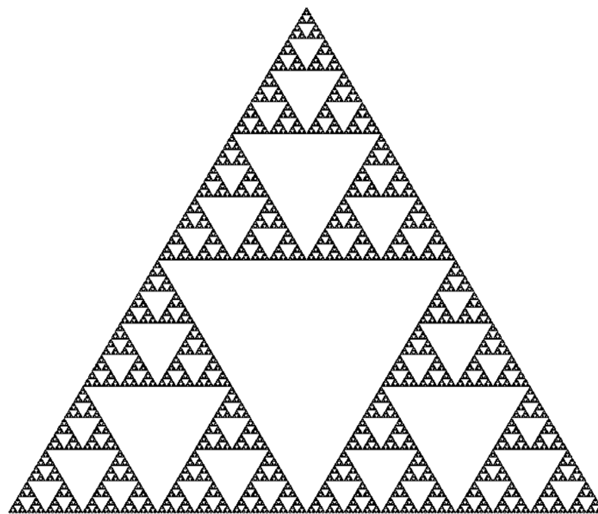
Sierpinskin kolmio on puolalaisen matemaatikko *Wacław Sierpińskin* vuonna 1915 esittelemä *fraktaali*. Se on tasasivuinen kolmio, jonka ympärille piirretään kolme uutta tasasivuista kolmiota niin, että kunkin uuden kolmion jokin kärki on edellisen (suuremman) kolmion sivun keskipisteessä. Kunkin uuden kolmion koko on puolet suuremmasta kolmiosta. Uudet kolmiot

muodostuvat siis ”ison” kolmion yläosaan, vasempaan alakulmaan ja oikeaan alakulmaan. Tilanne selviää paremmin kuvasta. Sierpinskin kolmion toinen vaihe on alla. Kolmion viivojen piirtämiseen käytämme `Canvas`-oliota (ks. luku 21).



Kuva 31: Sierpinskin kolmion toisessa vaiheessa ensimmäisen kolmion ympärille on piirretty kolme uutta kolmiota.

sekä ”lopputulos”, missä pienimpiä kolmioita on jo hyvin vaikea erottaa toisistaan.



Kuva 32: Valmis Sierpinskin kolmio.

Sierpinskin kolmion piirtäminen onnistuu loistavasti rekursiolla, mutta ilman rekursiota kolmion piirtäminen olisi melko työlästä. Sierpinskin kolmiosta voi lukea lisää esim. Wikipediasta: `http://en.wikipedia.org/wiki/Sierpinski_triangle`.

Kirjoitetaan algoritmi *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, x, y) // x ja y tarkoittavat kärjellään
                                       // seisovan kolmion alakulman koordinaatteja
{
   jos (korkeus < PIENIN_SALLITTU_KORKEUS) poistu

   sivunPituus2 = korkeus / sqrt(3) // Sivun pituus jaettuna kahdella
   alakulma = (x, y) // Pistepari
   vasenYlakulma = (x – sivunPituus2, y + korkeus)
   oikeaYlakulma = (x + sivunpituus2, y + korkeus)

   PiirraViiva(alakulma, vasenYlakulma) // Viiva alakulmasta vasempaan yläkulmaan
   PiirraViiva(vasenYlakulma, oikeaYlakulma)  // Vastaavasti ...
   PiirraViiva(oikeaYlakulma, alakulma)

   PiirraSierpinskinKolmio(korkeus / 2, x – sivunPituus2, y)
   PiirraSierpinskinKolmio(korkeus / 2, x + sivunPituus2, y)
   PiirraSierpinskinKolmio(korkeus / 2, x, y + korkeus)
}
```

Tämä muistuttaa jo paljon oikeaa koodia. Käytetään seuraavaksi oikeaa koodia.

```
using System;
using Jypeli;

/// <summary>
/// Sierpinskin kolmio
/// </summary>
public class Sierpinski : Game
{
    private static double pieninKorkeus = 10.0;

    public override void Begin()
    {
        Level.BackgroundColor = Color.White;
    }

    protected override void Paint(Canvas canvas)
    {
        base.Paint(canvas);
        double korkeus = 300;
        SierpinskinKolmio(canvas, 0, -korkeus, korkeus);
    }

    /// <summary>
    /// Piirtää Sierpinskin kolmion.
    /// </summary>
    /// <param name="canvas">Piirtoalusta</param>
    /// <param name="x">Alareunan x</param>
    /// <param name="y">Alareunan y</param>
    /// <param name="h">Korkeus</param>
    public static void SierpinskinKolmio(Canvas canvas, double x, double y, double h)
    {
        if (h < pieninKorkeus) return;

        double s2 = h / (Math.Sqrt(3)); // sivun pituus s/2
        Vector p1 = new Vector(x, y);
```

```
        Vector p2 = new Vector(x - s2, y + h);
        Vector p3 = new Vector(x + s2, y + h);
        canvas.DrawLine(p1, p2);
        canvas.DrawLine(p2, p3);
        canvas.DrawLine(p3, p1);

        SierpinskinKolmio(canvas, x - s2, y, h / 2);
        SierpinskinKolmio(canvas, x + s2, y, h / 2);
        SierpinskinKolmio(canvas, x, y + h, h / 2);
    }
}
```

Tarkastellaan ohjelman tiettyjä osia hieman tarkemmin.

```
private static double pieninKorkeus = 10.0;
```

Attribuuttina määritellään muuttuja, jolla kontrolloidaan kuinka kauan rekursiota jatketaan. Muuttuja `pieninKorkeus` näkyy siis kaikkialla luokassa `Sierpinski`. `pieninKorkeus` on määritelty ”globaaliksi” sillä perusteella, ettei muuttujan alustus toistuisi loputtomasti. Tässä ohjelmassa voidaan nimittäin suorittaa aliohjelma `SierpinskinKolmio` todella monta kertaa, riippuen muuttujan `pieninKorkeus` arvosta.

Yllä oleva muuttuja voisi olla myös vakio. Tämän kyseisen ohjelman tapauksessa se voisi olla jopa perusteltua. Kuitenkin, on myöskin perusteltua olettaa, että ohjelmamme kehittyessä pienimmän kolmion korkeutta olisi mahdollista muuttaa vaikkapa käyttäjän toimesta, ja silloin `pieninKorkeus` ei olisikaan enää vakio, vaan ohjelman ajon aikana muuttuva luku.

```
protected override void Paint(Canvas canvas)
{
  base.Paint(canvas);
  double korkeus = 300;
  SierpinskinKolmio(canvas, 0, -korkeus, korkeus);
}
```

`Paint`-aliohjelmassa määrittelemme ensimmäisenä piirrettävän, eli suurimman, kolmion korkeuden. Sen jälkeen kutsumme `SierpinskinKolmio`-aliohjelmaa, jolle välitämme parametrina `canvas`-olion, johon kolmioita piirretään, ja kolmion paikan (0, -korkeus) sekä tietenkin korkeuden.

```
public static void SierpinskinKolmio(Canvas canvas, double x, double y, double h)
```

Aliohjelma `SierpinskinKolmio` on staattinen, sillä sen suorittamiseksi riittää parametrina tulevat tiedot. Se on myös `void`-tyyppinen, koska emme odota sen palauttavan mitään. Aliohjelma saa neljä parametria: piirtoalusta, johon kolmio piirretään, kolmion alimman pisteen x- ja y-koordinaatit, sekä kolmion korkeuden. Nämä parametrit riittävät tasasivuisen kolmion piirtämiseen `Canvas`-olion avulla.

Sivuutetaan hetkeksi `if`-rakenne ja tarkastellaan `if`-lauseen jälkeen tulevia lauseita.

```
double s2 = h / (Math.Sqrt(3)); // sivun pituus s/2
```

Ennen kuin voimme piirtää kolmiot, meidän on selvitettävä, mitkä ovat kolmion sivujen pituudet. Tasasivuisen kolmion kaikki sivut ovat yhtä pitkiä, joten yhden sivun pituuden laskeminen riittää meille! Käytämme vanhaa kunnon Pythagoraan lausetta. Olkoon h kolmion korkeus ja s sivun pituus.

$$
\begin{aligned}
s^2 &= h^2 + (\frac{s}{2})^2 \\
s^2 - \frac{s^2}{4} &= h^2 \\
\frac{3s^2}{4} &= h^2 \\
s^2 &= \frac{4}{3}h^2 \\
s &= \sqrt{\frac{4h^2}{3}} = \frac{2h}{\sqrt{3}}, s \geq 0, h \geq 0
\end{aligned}
$$

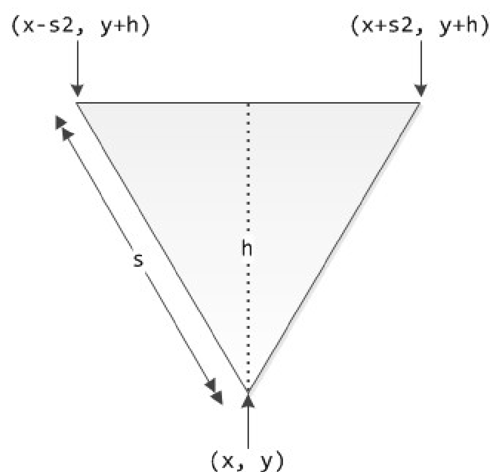
Koska x-akselilla siirrymme kolmion alimmasta kärjestä puolen sivun mitan verran joko vasemmalle tai oikealle, on mielekästä jakaa sivun pituus s vielä kahdella, jotta laskut hieman helpottuvat jatkossa.

$$
\frac{s}{2} = \frac{h}{\sqrt{3}}
$$

Tämä tulos on tallennetaan `s2`-muuttujaan.

```
Vector p1 = new Vector(x, y);
Vector p2 = new Vector(x - s2, y + h);
Vector p3 = new Vector(x + s2, y + h);
```

Yllä lasketaan kolmion kärkipisteiden paikat edellä laskettua sivun pituutta hyväksi käyttäen. Allaoleva kuva selventää vielä pisteiden laskemista.



Kuva 33: Kolmion pisteiden laskeminen.

Piirretään sitten yksi kolmio.

```
canvas.DrawLine(p1, p2);
canvas.DrawLine(p2, p3);
canvas.DrawLine(p3, p1);
```

Yllä olevat rivit piirtävät yhden kolmion hyödyntäen laskettuja kärkipisteiden koordinaatteja.

```
SierpinskinKolmio(canvas, x - s2, y, h / 2); // Vasen alakolmio
SierpinskinKolmio(canvas, x + s2, y, h / 2); // Oikea alakolmio
SierpinskinKolmio(canvas, x, y + h, h / 2); // Yläkolmio
```

Kutsutaan tehtyä aliohjelmaa kolmesti, jolloin alkuperäisen kolmion koordinaattien ja koon perusteella piirretään kolme pienempää kolmiota: alkuperäisen kolmion vasemmalle, oikealle ja yläpuolelle.

Otetaan hetkeksi askel taaksepäin, ja tarkastellaan, milloin rekursiosta poistutaan.

```
if (h < pieninKorkeus) return;
```

Aliohjelmaan tultaessa saatiin parametrina korkeus, `h`-muuttuja. Mikäli `h`:n arvo alittaa annetun pienimmän korkeuden, poistutaan välittömästi `return`-lauseella. Tällöin h:ta pienempiä kolmioita ei enää piirretä. Toisaalta, mikäli kolmion korkeus `h` *ei alita* annettua minimiä, niin silloin piirrellään, kuten aiemmin käytiin läpi.

Olennaista tässä on huomata, että niin kauan, kuin korkeus `h` on *enemmän* kuin annettu kolmion minimikorkeus, emme pääse ensimmäistä `SierpinskinKolmio`-aliohjelmakutsua ”pidemmälle”. Kullakin kutsukerralla näet korkeus `h` puolittuu, joten vasta `h`:n ollessa riittävän pieni lopetusehto toteutuu. Rekursion idean mukaisesti vasta sitten etenemme seuraaviin `SierpinskinKolmio`-kutsuihin (kaksi jälkimmäistä).

22.2 Harjoitus

Montako kertaa tässä esimerkissä lopulta suoritetaan aliohjelma `SierpinskinKolmio`?

23. 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 C#:ssä on dynaamisia tietorakenteita, eli kokoelmia. 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.

23.1 Rajapinnat

C#:ssa on olemassa *rajapintoja* (**interface**), joissa määritellään tietyt metodit, ja kaikkien luokkien, jotka toteuttavat (**implement**) tämän rajapinnan, täytyy sisältää samat metodit. Rajapintojen hienous on siinä, että voimme käyttää samoja metodeja kaikkiin niihin olioihin, jotka toteuttavat saman rajapinnan. 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 värittää silmukassa kerralla taulukollisen ympyröitä, kolmioita ja suorakulmioita samalla metodilla.

Kokoelmat ovat olio-ohjelmoinnin taulukoita. `Generics`-kokoelmaluokat (`System.Collections.Generics`-nimiavaruus) ovat tyyppiturvallisia, toisin sanoen kokoelman jäsenten (ja mahdollisen avaimen) tyyppi voidaan määritellä. `System.Collections.ObjectModel`-nimiavaruudessa on geneerisiä kantaluokkia omien kokoelmien toteuttamiseen sekä ”wrappereitä” (ns. kääreluokkia), joilla voidaan esimerkiksi tehdä read-only-kokoelmia.

Valmiita tietorakenteita on C#:ssa melko paljon, joten ennen oman tietorakenteen tekemistä kannattaa tutustua niihin. Tässä luvussa tutustumme lähinnä geneeriseen listaan (`List<T>`). Oman tietorakenteen tekeminen onkin jo sitten Ohjelmointi 2-kurssin asiaa.

23.2 Listat (`List<T>`)

Tutustutaan seuraavaksi yhteen C#:n dynaamisista tietorakenteista, `List<T>`-luokkaan, joka on geneerinen tietorakenne. `List<T>` muistuttaa jonkin verran taulukkoa; taulukoilla ja listoilla on paljon yhteistä:

- Niissä voi olla vain yhden tyyppisiä alkioita
- Yksittäiseen alkioon päästään käsiksi laittamalla alkion paikkaindeksi hakasulkujen sisään, esimerkiksi `luvut[15, 14]`, tai `pallot[4]`.
- Molemmilla on metodeja (funktioita, aliohjelmia) sekä ominaisuuksia

`List<T>`-olioon ja muihin dynaamisiin tietorakenteisiin voi tallentaa niin alkeistietotyyppejä kuin oliotietotyyppejäkin. Käsittelemämme *geneerinen lista* vaatii *aina* tiedon siitä, minkä tyyppisiä alkioita tietorakenteeseen laitetaan. Muun tyyppisiä alkioita listaan ei voi laittaa.

Tietotyyppi laitetaan tietorakenneluokan jälkeen kulmasulkujen sisään - tästä esimerkki seuraavaksi.

23.2.1 Tietorakenteen määrittäminen

Dynaamisen tietorakenteen määrittämisen syntaksi poikkeaa hieman tavallisen olion määrittelystä. Ehdit jo varmaan ihmetellä, mikä on kulmasulkeissa oleva `T` `List`-sanan jälkeen. Kyseinen `T` tarkoittaa listaan talletettavien alkioiden tyyppiä. Tyyppi voi olla alkeistietotyyppi tai oliotyyppi. Yleisessä muodossa uuden listan määrittely menee seuraavasti:

```
TietorakenneLuokanNimi<TallettavienOlioidenTyyppi> rakenteenNimi =
  new TietorakenneLuokanNimi<TallettavienOlioidenTyyppi>();
```

Voisimme esimerkiksi tallettaa elokuvien nimiä seuraavaan `List<String>`-rakenteeseen. Määritellään uusi (tyhjä) lista seuraavasti.

```
List<String> elokuvat = new List<String>();
```

23.2.2 Alkioiden lisääminen ja poistaminen

Alkioiden lisääminen `List<T>`-olioon, ja itse asiassa kaikkiin `Collections.Generic`-nimiavaruuden luokkien olioihin, onnistuu `Add`-metodilla. `Add`-metodi lisää alkion aina tietorakenteen ”loppuun”, eli loogisessa mielessä viimeiseksi. 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");
elokuvat.Add("Toy Story");
```

Alkion poistaminen halutusta paikasta (indeksistä) tehdään `RemoveAt`-metodilla. Parametriksi annetaan sen alkion indeksi, joka halutaan poistaa. Alkion "Casablanca" poistaminen onnistuisi seuraavasti.

```
elokuvat.RemoveAt(0);
```

Koska rakenne on dynaaminen, muuttuu listan alkioiden järjestys lennosta. Nyt "Star Wars"-merkkijono löytyisi indeksistä 0. Poistaa voi myös suoraan alkion sisällöllä.

```
elokuvat.Remove("Star Wars");
```

`Remove`-metodi toimii siten, että se poistaa listasta ensimmäisen esiintymän, joka vastaa annettua parametria. Metodi palauttaa `true`, mikäli listasta poistettiin alkio. Vastaavasti palautetaan `false`, mikäli annettua parametria vastaavaa alkiota ei löytynyt, jolloin listasta ei poistettu mitään.

Tietorakenteen koon, tai oikeammin sanottuna tietorakenteen sisältämien alkioiden lukumäärän, tietää olion `Count`-ominaisuus.

```
Console.WriteLine(elokuvat.Count); //tulostaa 1
elokuvat.Add("Full Metal Jacket");
Console.WriteLine(elokuvat.Count); //tulostaa 2
```

Tiettyyn alkioon pääsee käsiksi taulukon tapaan, eli laittamalla haluttu paikkaindeksi hakasulkeiden sisään. Ensimmäisen alkion voisi tulostaa esimerkiksi seuraavaksi:

```
Console.WriteLine(elokuvat[0]); // tulostaa "Toy Story"
```

Näillä metodeilla pärjää jo melko hyvin. Muista metodeista voi lukea `List<T>`-luokan dokumentaatiosta: http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx.

Tehdään toinen esimerkki `int`-tyyppisillä luvuilla. Annetaan listalle sisältö heti listaa alustettaessa.

```
List<int> luvut = new List<int>() { 3, 3, 1, 7, 3, 5, 7 };
```

Huomaa, että edellä olevaan listaan ei voi tallentaa muita kuin `int`-tyyppisiä kokonaislukuja.

```
luvut.Add(5.3); // Kääntäjä ilmoittaa virheestä!
```

Yllä oleva esimerkki osoittaa, että näiden *vahvasti tyypitettyjen* tietorakenteiden käyttö on myös turvallista – tietorakenteeseen ei voi ”vahingossa” laittaa väärän tyyppisiä alkioita, joka saattaisi sitten myöhemmin aiheuttaa vakavia ongelmia.

Tarkistetaan vielä listan sisältämien alkioiden lukumäärä.

```
Console.WriteLine(luvut.Count); // Tulostaa 7
```

Poistetaan sitten kaikki ne alkiot, joiden arvo on 3. Tässä voimme käyttää tehokkaasti hyväksemme `while`-silmukkaa ja `Remove`-funktiota. `Remove`-funktion totuusarvotyyppinen paluuarvo käy hyvin `while`-ehdoksi.

```
While (luvut.Remove(3));
```

Listan alkiot näyttävät tämän jälkeen seuraavalta.

```
1, 7, 5, 7
```

24. 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 `IndexOutOfRangeException`-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.

C#:n poikkeukset ovat olioita. [VES][KOS][DEI]

24.1 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 `DivideByZeroException`-poikkeuksen.

```
int n1 = 7, n2 = 0, n3 = 4;

try
{
   Console.WriteLine("{0}", 10 / n1);
   Console.WriteLine("{0}", 10 / n2);
   Console.WriteLine("{0}", 10 / n3);
}
```

```
catch (DivideByZeroException e)
{
   Console.WriteLine("Nollalla jako: " + e.Message);
}
```

Yllä olevassa esimerkissä keskimmäinen tulostus aiheuttaisi `DivideByZeroException`-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 C#:n poikkeukset olivat olioita, on niillä myös joukko metodeja ja ominaisuuksia. `catch`-lohkossa on kutsuttu `DivideByZeroException`-luokan `Message`-ominaisuutta, joka sisältää poikkeukselle määritellyn virheilmoituksen, jonka siis tulostamme tässä konsoli-ikkunaan.

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 C#:n 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]

24.2 finally-lohko

`finally`-lohkon käyttäminen ei ole pakollista, mutta kun sitä käytetään, kirjoitetaan se `catch`-lohkojen jälkeen. Mikäli `finally`-lohko kirjoitetaan mukaan, suoritetaan se joka tapauksessa riippumatta siitä aiheuttiko `try`-lohko poikkeuksia.

`finally`-lohko on 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
}
```

24.3 Yleistä

Poikkeukset ovat nimensä mukaan säännöstä poikkeavia tapahtumia. Niitä ei tulisikaan käyttää periaatteella: "En ole varma toimiiko tämä, joten laitan `try-catch`-rakenteen 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.

25. Tietojen lukeminen ulkoisesta lähteestä

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. Tässä luvussa selvitetään yksinkertaisten esimerkkien avulla tiedon lukeminen tiedostosta, sekä tiedon hakeminen WWW:stä.

Tutkitaan seuraavaksi esimerkkiä tekstin lukemisesta tiedostosta Windows-ympäristössä. Muuntyyppisten tiedostojen lukeminen, tiedostoon kirjoittaminen, sekä toiminta Windows Phone 7- ja Xbox-ympäristöissä jätetään tämän kurssin osaamistavoitteiden ulkopuolelle.

25.1 Tekstin lukeminen tiedostosta

Tehdään yksinkertainen ohjelma, joka lukee tekstitiedoston sisällön. Tekstitiedoston sisältö on seuraavanlainen.

```
Kalle, 5
Pekka, 10
Janne, 0
Irmeli, 15
```

Näiden voidaan ajatella olevan vaikka topten-listan pisteitä. Annetaan tiedoston nimeksi `data.txt`. Yksinkertaisuuden vuoksi tulostamme listan vain ruudulle. Oikeassa elämässä todennäköisesti haluaisimme paloitella listan rivit taulukkoon (tai muuhun tietorakenteeseen), josta saisimme eroteltua henkilöiden nimet ja pisteet.

```
using System;
using System.IO;

/// <summary>
/// Yksinkertainen tiedoston-sisältö-näytölle -ohjelma,
/// joka demonstroi StreamReader-olion käyttöä.
/// </summary>
public class TiedostostaTulostaminen
{
    /// <summary>
    /// Luetaan tiedosto ja tulostetaan sen sisältö näytölle.
    /// </summary>
    public static void Main()
    {
        FileStream tiedostovirta;
        const string TIEDOSTO = "data.txt";
        string rivi;

        if (!File.Exists(TIEDOSTO)) return;
        tiedostovirta = new FileStream(TIEDOSTO, FileMode.Open);
        StreamReader virtaLukija = new StreamReader(tiedostovirta);

        // Luetaan, ja tulostetaan, rivi kerrallaan
        while ((rivi = virtaLukija.ReadLine()) != null)
        {
            Console.WriteLine(rivi);
        }

        virtaLukija.Close();
        Console.ReadKey();
    }
```

```
}
```

Tutkitaan tärkeimpiä kohtia hieman tarkemmin

```
if (!File.Exists(TIEDOSTO)) return;
```

Tarkistetaan, onko tiedostoa olemassa. Mikäli ei ole, ohjelman suoritus päättyy siihen.

```
fin = new FileStream(TIEDOSTO, FileMode.Open);
```

Luodaan tietovirtaolio (`FileStream`), jonka avulla tiedostosta luettavaa tietoa hallitaan.

```
StreamReader fstr_in = new StreamReader(fin);
```

Tietovirtaa luetaan `StreamReader`-olion avulla. Käytännössä tämä muuttaa tavuvirran (**byte stream**) luettavaan merkkimuotoon.

```
while ((rivi = fstr_in.ReadLine()) != null)
```

Luetaan tietovirtaa `ReadLine`-metodin avulla, ja kullakin kierroksella rivi sijoitetaan `rivi`-muuttujaan. Tietovirtaa luetaan niin kauan, kunnes tullaan `null`-viitteeseen, joka tarkoittaa `StreamReader`-oliolla sitä, että tiedosto päättyy, eli tultiin loppuun.

```
fstr_in.Close();
```

Suljetaan lopuksi tietovirran lukija. Tämä on hyvä tehdä aina lopuksi, jottei lukija jää viemään turhia resursseja.

Huomaa, että Visual Studio hakee tiedostoja oletuksena samasta kansiosta, kuin mihin ajettava `exe`-tiedosto syntyy, joten VS:ssä `data.txt`-tiedostolle pitää antaa ominaisuus Copy to Output Directory → Copy if newer (tai Copy always). Muutoin tiedostoa ei löydy, ja `File.Exists`-metodi palauttaa `false` ja ohjelman ajaminen päättyy siihen.

25.2 Tekstin lukeminen netistä

Luetaan seuraavaksi tietoja netistä. Tässä koko HTML-sivun data tallennetaan rivi kerrallaan `List<String>`-tietorakenteeseen ilman sen kummempaa jatkokäsittelyä. Lopuksi listan sisältö tulostetaan ruudulle rivi kerrallaan.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Text;

/*
 * Author: Antti-Jussi Lakanen.
 * Jyväskylän yliopisto, tietotekniikan laitos, 2011.
 */

/// <summary>
/// Tulostetaan kayttajan antaman www-osoitteen
/// sisältö (GET-pyynnön palauttama HTML-koodi).
/// </summary>
public class TiedotNetista
{
```

```
    /// <summary>
    /// Paaohjelmassa haetaan tiedot netistä listaan ja tulostetaan listan sisältö.
    /// </summary>
    /// <param name="args"></param>
    public static void Main(string[] args)
    {
        List<String> lista = new List<string>();
        LueNetistaListaan("https://www.jyu.fi", lista);

        foreach (String rivi in lista)
        {
            Console.WriteLine(rivi);
        }
        Console.ReadKey();
    }

    /// <summary>
    /// Luetaan annetun url-osoitteen koko html-koodi
    /// ja laitetaan rivi kerrallaan String-listaan.
    /// </summary>
    /// <param name="url">URL-osoite, mika halutaan lukea.</param>
    /// <param name="lista">Lista, johon riveja kirjoitetaan.</param>
    public static void LueNetistaListaan(String url, List<String> lista)
    {
        HttpWebResponse response = null;
        StreamReader reader = null;
        try
        {
            // Lahetetaan HTTP-pyynto palvelimelle
            HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
            request.Method = "GET"; // Maaritellaan pyynnon tyypiksi GET
            response = (HttpWebResponse)request.GetResponse();
            reader = new StreamReader(response.GetResponseStream(), Encoding.UTF8);
            while (!reader.EndOfStream)
            {
                String rivi = reader.ReadLine();
                lista.Add(rivi);
            }
        }
        catch (WebException we) // Ellei saada luotua verkkoyhteytta sopivalla
protokollalla
        {
            Console.WriteLine(we.Message);
        }
        finally
        {
            if (reader != null)
                reader.Close();
            if (response != null)
            // Tietovirta taytyy sulkea, jottei sovelluksessa
            // jouduttaisi tilaan, missa yhteydet "loppuvat kesken".
                response.Close();
        }
    }
}
```

Mukana on yksi `try-catch-finally`-rakenne. Verkkoyhteydet ovat alttiita kaikenlaisille virheille, joten poikkeusten ”kiinniottaminen” on perusteltua ja suorastaan välttämätöntä.

25.3 Satunnaisluvut

`Random`-luokka sijaitsee System-nimiavaruudessa. `Random`-luokasta löytyy metodeja joilla voimme arpoa erityyppisiä satunnaislukuja. Arpomista varten meidän täytyy luoda `Random`-olio, jotta metodeja voitaisiin kutsua. `Random`-oliolla on metodi `Next`, 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 muuttaa

parametria vastaavasti, sillä 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.Next(11);
```

Jos haluaisimme arpoa luvun esimerkiksi suljetulta väliltä [50, 99], sanoisimme

```
Random rand = new Random();
int satunnaisluku = rand.Next(50, 100);
```

Liukuluku arvotaan `NextDouble`-metodilla.

Jypelissä olevasta `RandomGen`-luokasta löytyy useita staattisia metodeja, joilla satunnaislukujen (ja -värien, totuusarvojen, jne.) luominen on helpompaa. Lue `RandomGen`-luokan dokumentaatio osoitteesta http://kurssit.it.jyu.fi/npo/material/latest/documentation/html/ .

26. Lukujen esitys tietokoneessa

26.1 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..9` ja (jatkuen) `A..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]

26.2 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

$$25.36 = 2*10^1 + 5*10^0 + 3*10^{-1} + 6*10^{-2} = 2*10 + 5*1 + 3*0.1 + 6*0.01$$

26.3 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.

26.3.1 Binääriluku 10-järjestelmän luvuksi

Esimerkiksi binääriluku 10110 voidaan muuttaa 10-järjestelmän luvuksi seuraavasti.

$$10110_2 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 0 + 2 + 4 + 0 + 16 = 22_{10}$$

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.

26.3.2 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.

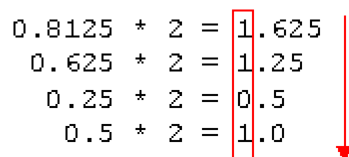
```
126 = 2*63+0
 63 = 2*31+1
 31 = 2*15+1
 15 =  2*7+1
  7 =  2*3+1
  3 =  2*1+1
  1 =  2*0+1
```

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.



Kuva 34: 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 $23.375_{10} = 10111.011_2$.

26.4 Negatiiviset binääriluvut

Negatiivinen luku voidaan esittää joko suorana, 1-komplementtina tai 2-komplementtina.

26.4.1 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.

26.4.2 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.

26.4.3 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 totutuun 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!

26.4.4 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

```
 111
  0110
+ 0111
  ----
  1001   => 0110 + 1  => -7 (siis neg. luku kahden pos. luvun yhteenlaskusta)
```

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 alkeistietotyyppien lukualueisiin, jotka esiteltiin kohdassa 7.2. 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.

26.5 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.

```
11101101₂ = 1110 1101₂
1110₂ = E₁₆
1101₂ = D₁₆
11101101₂ = 1110 1101₂ = ED₁₆
```

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.

Taulukko 8: Lukujen vastaavuus eri lukujärjestelmissä.

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

26.6 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:

$$x = (-1)^s m k^c$$

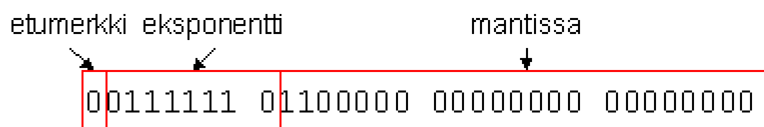
Tietotekniikassa yleisimmin käytetyssä standardissa IEE 754 kantaluku on 2, jolloin kaava saadaan muotoon:

$$x = (-1)^s m 2^c$$

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.

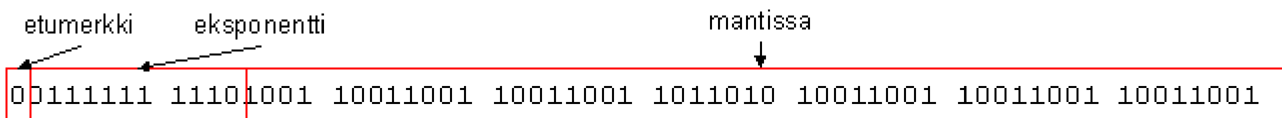
Tutustutaan seuraavaksi kuinka float ja double esitetään bittimuodossa.

`float` on kooltaan 32 bittiä. Siinä ensimmäinen bitti siis tarkoittaa etumerkkiä, seuraavat 8 bittiä eksponenttia ja jäljelle jäävät 23 bittiä mantissaa.



Kuva 35: 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 36: 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.

26.6.1 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:

$$1.75 * 2^{-1} = 0.875$$

26.6.2 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*2⁰ = 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.

26.6.3 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.

26.6.4 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 C#:n Decimal-luokkaa (Javan `BigDecimal`). 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

nopeampia, 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**).

26.6.5 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

27. 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.

Taulukko 9: ASCII-merkistö.

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

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
```

28. Syntaksin kuvaaminen

28.1 BNF

Tässä luvussa kuvataan Java-kielen syntaksia. Syntaksia eli kielioppia voidaan kuvata ns. 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.

28.2 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.

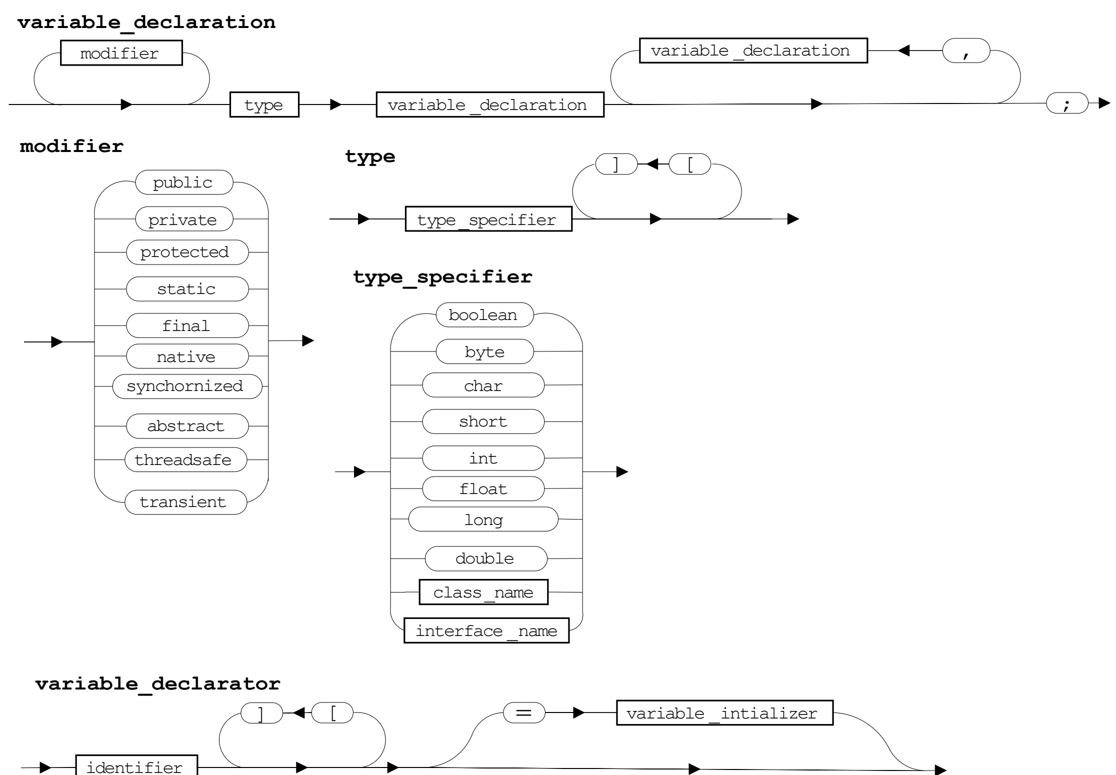
{}	Aaltosulkeiden sisään kirjoitetut osat voidaan jättää joko kokonaan pois tai toistaa yhden tai useamman kerran.
[]	Hakasulkeiden sisään kirjoitetut osat voidaan suorittaa joko kerran tai ei ollenkaan.

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://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html



Kuva 37: Muuttujan määrittelyn syntaksia "junaradoilla" esitettynä

Vastaavasti syntaksia voidaan kuvata ”junaradoilla”. Tämä on eräs graafinen tapa kuvata syntaksia.

Kuvataan seuraavaksi muuttujaan määrittelyä "junaratojen" avulla. 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://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html
- Wikipedia: http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form
- Googlella löytyy lisää

C#:n syntaksia on kuvattu MSDN-dokumentaatiossa muun muassa seuraavilla sivuilla.

http://msdn.microsoft.com/en-us/library/Aa664812.

29. Jälkisanat

Joskus ohjelmoidessa tulee vaan tämmöinen olo:

http://www.youtube.com/watch?v=K21fuhDo5Bo

Totu siihen ja keitä lisää kahvia.

Liite: Sanasto

Internetistä löytyy ohjelmoinnista paremmin tietoa englanniksi. Tässä tiedonhakua auttava sanasto ohjelmoinnin perustermeistä.

aliohjelma	**subprogram, subroutine, procedure**	konstruktori	**constructor**	rajapinta	**interface**
alirajapinta	**subinterface**	koodauskäytänteet	**code conventions**	roskienkeruu	**garbage collection**
alivuoto	**underflow**	kääntäjä	**compiler**	roskienkerääjä	**garbage collector**
alkeistietotyyppi	**primitive types**	kääriä	**wrap**	sijoituslause	**assignment statement**
alkio	**element**	lause	**statement**	sijoitusoperaattori	**assignment operator**
alustaa	**intialize**	lippu	**flag**	silmukka	**loop**
aritmeettinen operaatio	**arithmetic operation**	lohko	**block**	sovelluskehitin	**Integrated Development Environment**
aritmeettinen lauseke	**arithmetic expression**	luokka	**class**	staattinen	**static**
bugi	**bug**	metodi	**method**	standardi syöttövirta	**standard input stream**
destruktori	**destructor**	muuttuja	**variable**	standardi tulostusvirta	**standard output stream**
dokumentaatio	**documentation**	määritellä	**declare**	standardi virhetulostusvirta	**standard error output stream**
funktio	**function**	olio	**object**	syntaksi	**syntax**
globaali vakio	**global constant**	ottaa kiinni	**catch**	taulukko	**array**
globaali muuttuja	**global variable**	paketti	**package**	testaus	**testing**
indeksi	**index**	parametri	**parameter**	toteuttaa	**implement**
julkinen	**public**	periytyminen	**inheritance**	tuoda	**import**
keskeytyskohta	**breakpoint**	poikkeus	**exception**	vakio	**constant**
komentorivi	**Command Prompt**	poikkeustenhallinta	**exception handling**	yksikkötestaus-rajapinta	**unit testing framework**
				ylivuoto	**overflow**

Liite: Yleisimmät virheilmoitukset ja niiden syyt

Aloittavan ohjelmoijan voi joskus olla vaikeaa saada selvää kääntäjän virheilmoituksista. Kootaan tänne muutamia yleisimpiä C#-kääntäjän virheilmoituksia. Osa virheilmoituksista on Jypeli-spesifisiä.

Tyyppiä tai nimiavaruutta ei löydy

```
The type or namespace name 'PhysisObject' could not be found (are you missing a using
directive or an assembly reference?)
```

Syitä

- Oletko kirjoittanut esim. jonkun aliohjelman tai tyypin nimen väärin? Katso sanoja, jotka on väritetty punakynällä. Äskeisessä virheviestissä `PhysisObject` pitäisi kirjoittaa `PhysicsObject`. Käytä Visual Studion koodin täydennystä kirjoitusvirheiden välttämiseksi.
- Jokin kirjasto puuttuu (kts Kirjastojen liittäminen projektiin: wiki, video)
- Jokin `using`-lause puuttuu. Jypeli-pelien projektimalleissa ovat vakiona seuraavat `using`-lauseet:

```
using System;
using Jypeli;
using Jypeli.Widgets;
using Jypeli.Assets;
```

Peli.Aliohjelma(): not all code paths return a value

Aliohjelmalle on määritelty paluuarvo, mutta se ei palauta mitään (eli `return`-lause puuttuu).

Seuraavassa aliohjelmassa paluuarvoksi on määritelty `PhysicsObject`, mutta aliohjelma ei palauta mitään arvoa.

```
PhysicsObject LuoPallo()
{
    PhysicsObject pallo = new PhysicsObject(50.0, 50.0, Shape.Circle);
}
```

Tällöin Visual Studio antaa virheilmoituksen.

Jos pallo halutaan palauttaa aliohjelmasta, niin aliohjelmaa pitää korjata seuraavasti.

```
PhysicsObject LuoPallo()
{
    PhysicsObject pallo = new PhysicsObject(50.0, 50.0, Shape.Circle);
    return pallo;
}
```

Muuttujaa ei ole olemassa nykyisessä kontekstissa

```
The name 'massa' does not exist in the current context
```

Seuraavassa koodinpätkässä käytetään muuttujaa nimeltä `massa`, mutta tuota muuttujaa ei ole esitelty missään. Jokainen muuttuja, jota ohjelmassa käytetään, tulee esitellä jossakin. Esittely tarkoittaa, että jollakin rivillä kirjoitetaan muuttujan tyyppi sekä nimi seuraavasti:

```
double massa;
```

Samalla rivillä esittelyn kanssa voi myös sijoittaa muuttujalle alkuarvon:

```
double massa = 100.0;
```

Niinpä äskeisen koodin virhe voidaan korjata kertomalla muuttujan `massa` tyyppi (tyyppi on `double`) siinä missä tuo muuttuja ensimmäisen kerran otetaan käyttöön:

Jos muuttuja esitellään aliohjelmassa, se pitää esitellä ennen sen käyttöä. Jos muuttujaa tarvitaan useammassa aliohjelmassa, pitää se esitellä luokan sisällä:

```
public class Peli : PhysicsGame
{
    double massa;

    public override void Begin()
    {
        massa = 100.0;
        PhysicsObject pallo = new PhysicsObject(50.0, 50.0, Shape.Circle);
        pallo.Mass = massa;
        TulostaMassa();
    }

    void TulostaMassa()
    {
        MessageDisplay.Add("Massa on " + massa);
    }
}
```

Lähdeluettelo

DOC: Sun, , , http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
HYV: Hyvönen Martti, Lappalainen Vesa, Ohjelmointi 1, 2009
KOSK: Jussi Koskinen, Ohjelmistotuotanto-kurssin luentokalvot(Osa: Ohjelmistojen ylläpito),
KOS: Kosonen, Pekka; Peltomäki, Juha; Silander, Simo, Java 2 Ohjelmoinnin peruskirja, 2005
VES: Vesterholm, Mika; Kyppö, Jorma, Java-ohjelmointi, 2003
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
DEI: Deitel, H.M; Deitel, P.J, Java How to Program, 2003

Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 17

University of Jyväskylä
Department of Mathematical Information Technology

Lecture Notes 17

1. MÄKINEN, RAINO A. E., Numeeriset menetelmät. 1999 (107 s.)
2. LAPPALAINEN, VESA ja RISTO LAHDELMA, Olio-ohjelmointi ja C++. 1999 (107 s.)
3. LAPPALAINEN, VESA, Windows-ohjelmointi C-kielellä. 1999 (150 s.)
4. ORPONEN, PEKKA, Tietorakenteet ja algoritmit 2. 2.p., 2000 (50 s.)
5. LAPPALAINEN, VESA, Ohjelmointi++. 1999 (315 s.)
6. MÄNNIKKÖ, TIMO, Johdatus ohjelmointiin. 2000 (155 s.)
7. KOIKKALAINEN, PASI ja PEKKA ORPONEN, Tietotekniikan perusteet. 2001 (150 s.)
8. ARNĂUTU, VIOREL, Numerical methods for variational problems. 2001 (100 s.)
9. KRAVCHUK, ALEXANDER, Mathematical modelling of the biomedical tomography: The 12th Jyväskylä Summer School. 2003 (83 s.)
10. MIETTINEN, KAISA, Epälineaarinen optimointi. 2003 (146 s.)
11. LAPPALAINEN VESA & VIITANEN SANTTU, Ohjelmointi 2. 2012 (240 s.)
12. KAIJANAHO, ANTTI-JUHANI & KÄRKKÄINEN TOMMI, Formaalit menetelmät. 2005 (171 s.)
13. HOPPE, RONALD H. W., Numerical solution of optimization problems with PDE constraints: Lecture notes of a course given in the 14th Jyväskylä Summer School, August 9-27, 2004. 2006 (65 s.)
14. JYRKI JOUTSENSALO, TIMO HÄMÄLÄINEN & ALEXANDER SAYENKO, QoS Supported Networks, Scheduling, and Pricing; Theory and Applications (214 s.)
15. MARTTI HYVÖNEN, VESA LAPPALAINEN, Ohjelmointi 1. 2009. (132 s.)
16. ANTTI-JUHANI KAIJANAHO, Ohjelmointikielten periaatteet. 2010. (153 s.)
17. MARTTI HYVÖNEN, VESA LAPPALAINEN, ANTTI-JUSSI LAKANEN, Ohjelmointi 1 C#. Korjattu painos, 2012. (160 s.)

ISBN 978-951-39-4434-6 (korjattu painos)
ISSN 1456-9787