Ohjelmointi 1, C#

Martti Hyvönen, Vesa Lappalainen ja Antti-Jussi Lakanen

Versio: 22.9.2020

22. syyskuuta 2020

Jyväskylän yliopisto

Sisällys

Esipuhe **1**

Esipuheen jälkipuhe **3**

0 Johdanto **4**
0.1 Kurssin sisällöstä ja tavoitteista . . . 4
0.2 Kurssin osaamistavoitteet . . . 5
0.3 TIM-käyttöohjeita . . . 6

1 Mitä ohjelmointi on? **9**
1.1 Algoritmit eli ohjeet . . . 9
1.2 Ohjelmointikielistä . . . 11
1.2.1 Prosessori ja konekieli . . . 11
1.2.2 C-kieli ja robotti . . . 12
1.2.3 Tavukielet . . . 14
1.2.4 C# ja Jypeli . . . 15
1.2.5 Muita kieliä . . . 16

2 Ensimmäinen C#-ohjelma **17**
2.1 Ohjelman kirjoittaminen . . . 17
2.2 Ohjelman kääntäminen ja ajaminen . . . 19
2.3 Ohjelman rakenne . . . 20
2.3.1 Virhetyypit . . . 24
2.3.2 Kääntäjän virheilmoitusten tulkinta . . . 25
2.3.3 Tyhjät merkit (White spaces) . . . 26
2.4 Kommentointi . . . 27
2.4.1 Dokumentointi . . . 28

3 Algoritmit **31**
3.1 Mikä on algoritmi? . . . 31
3.2 Tarkentaminen . . . 31
3.3 Yleistäminen . . . 32
3.4 Harjoitus . . . 32
3.5 Peräkkäisyys . . . 33

4 Yksinkertainen graafinen C#-ohjelma **35**
4.1 Mikä on kirjasto? . . . 35
4.2 Jypeli-kirjasto . . . 36
4.3 Esimerkki: Lumiukko . . . 36

4.3.1 Ohjelman suoritus . . . 37
4.4 Harjoitus . . . 43
4.5 Kääntäminen ja luokkakirjastoihin viittaaminen . . . 44

5 Lähdekoodista prosessorille **45**
5.1 Kääntäminen . . . 45
5.2 Suorittaminen . . . 45

6 Aliohjelmat **47**
6.1 Aliohjelman kutsuminen . . . 51
6.1.1 Aliohjelmakutsun kirjoittaminen . . . 52
6.2 Aliohjelman kirjoittaminen . . . 52
6.2.1 Valmis kokonaisuus . . . 58
6.3 Aliohjelmien dokumentointi . . . 60
6.3.1 Huomautus . . . 62
6.4 Aliohjelmat, metodit ja funktiot . . . 62
6.4.1 Aliohjelmien kirjoittaminen . . . 63
6.4.2 Tehtävä: Termistöä . . . 65
6.5 Aliohjelman kuormittaminen . . . 65
6.5.1 Yksinkertaisin esimerkki . . . 66
6.5.2 Vakiokokoinen lumiukko vs ukon koko parametrina . . . 67

7 Muuttujat **71**
7.1 Muuttujan määrittely . . . 72
7.2 Alkeistietotyypit . . . 74
7.3 Arvon asettaminen muuttujaan . . . 76
7.3.1 Sijoituksen kohde on aina vasemmalla . . . 78
7.3.2 Tehtävä 7.4 a:n arvon sijoitus b:lle . . . 79
7.3.3 Muuttujan arvo muuttuu vain kun siihen sijoitetaan . . . 79
7.3.3.1 Tehtävä 7.5 i:n kasvatus, mitä ohjelma tulostaa . . . 79
7.4 Muuttujan nimeäminen . . . 80
7.4.1 C#:n avainsanat . . . 81
7.5 Muuttujien näkyvyys . . . 82
7.6 Vakiot . . . 87
7.7 Operaattorit . . . 88
7.7.1 Aritmeettiset operaatiot . . . 89
7.7.2 Vertailuoperaattorit . . . 90
7.7.3 Arvonmuunto-operaattorit . . . 90
7.7.4 Aritmeettisten operaatioiden suoritusjärjestys . . . 91
7.8 Huomautuksia . . . 91
7.8.1 Kokonaisluvun tallentaminen liukulukumuuttujaan . . . 91
7.8.1.1 Tehtävä 7.8 . . . 92
7.8.2 Lisäys- ja vähennysoperaattoreista . . . 93
7.8.3 Varo nollalla jakamista . . . 93
7.8.4 Numeeristen tietotyyppien arvo-alueet . . . 94
7.9 Esimerkki: Painoindeksi . . . 94

8 Oliotietotyypit **97**
8.1 Mitä oliot ovat? . . . 97

8.2 Olion luominen . . . 98
8.3 Arvopohjaiset tietotyypit ja viitepohjaiset tietotyypit . . . 98
8.4 Metodin kutsuminen . . . 100
8.5 Metodin ja aliohjelman ero . . . 101
8.6 Olion tuhoaminen ja roskienkeruu . . . 101
8.7 Olioluokkien dokumentaatio . . . 102
8.7.1 Konstruktorit . . . 102
8.7.2 Harjoitus . . . 104
8.7.3 Metodit . . . 104
8.7.4 Huomautus: Luokkien dokumentaatioiden googlettaminen . . . 105
8.8 Tyyppimuunnokset . . . 105

9 Aliohjelman paluuarvo **107**
9.1 Keskiarvon laskeva funktio . . . 107
9.2 Funktion kutsuminen . . . 108
9.3 Funktion kirjoittaminen toisella tavalla . . . 109
9.4 Useita return-lauseita . . . 110
9.5 Funktio palauttaa yhden arvon . . . 110
9.6 Funktion kutsu maksaa . . . 111
9.7 YmpyranAla, esimerkki yhden parametrin funktiosta . . . 111
9.8 Tehtäviä funktioista . . . 112
9.8.1 Harjoituksia aliohjelmista . . . 112
9.8.2 Harjoituksia funktioista . . . 115

10 Ohjelmoijan työkaluja: Git, IDE **119**
10.1 Git . . . 119
10.2 Integroitu kehitysympäristö, IDE . . . 119
10.3 IDEn käyttö . . . 120
10.3.1 Käyttöönotto, solutionit ja projektit . . . 120
10.3.2 Ohjelman kirjoittaminen . . . 120
10.3.3 Ohjelman kääntäminen ja ajaminen . . . 120
10.4 Debuggaus . . . 121
10.5 Hyödyllisiä ominaisuuksia . . . 122
10.5.1 Syntaksivirheiden etsintä . . . 122
10.5.2 Kooditäydennys, IntelliSense . . . 122
10.5.3 Uudelleenmuotoilu . . . 122
10.5.4 TODO-tehtävien luettelo . . . 123
10.6 Lisätietoja . . . 123

11 Testaaminen **124**
11.1 Comtest . . . 124
11.2 Käyttö . . . 125
11.2.1 Kirjoita tynkä-toteutus . . . 126
11.2.2 Kirjoita dokumentaatio ja testit . . . 126
11.2.3 Toteuta aliohjelma ja aja testit . . . 127
11.2.4 Yleistä testeistä . . . 128
11.3 Liukulukujen testaaminen . . . 130

12 Merkkijonot **132**

12.1 Alustaminen . . . 132
12.1.1 Merkkijono on kuin taulukko . . . 134
12.1.2 Null-viittaus ja tyhjä merkkijono . . . 134
12.2 Hyödyllisiä metodeja ja ominaisuuksia . . . 135
12.2.1 Metodit palauttavat uuden jonon . . . 135
12.2.2 Merkkijonometodeja . . . 136
12.2.2.1 Equals . . . 136
12.2.2.2 Compare . . . 137
12.2.2.3 Contains . . . 137
12.2.2.4 IndexOf . . . 137
12.2.2.5 Substring . . . 138
12.2.2.6 ToLower . . . 138
12.2.2.7 ToUpper . . . 139
12.2.2.8 Replace . . . 139
12.2.2.9 Lisäksi . . . 139
12.2.3 Harjoitus 12.2 . . . 141
12.3 Huomautus . . . 141
12.4 Muokattavat merkkijonot: StringBuilder . . . 141
12.4.1 Muita StringBuilder-luokan hyödyllisiä metodeja . . . 143
12.4.2 StringBuildereiden testaaminen . . . 143
12.4.3 Kutsujen ketjuttaminen . . . 145
12.5 string-viite vs. StringBuilder-viite . . . 146
12.6 Huomautus: aritmeettinen + vs. merkkijonoja yhdistelevä + . . . 147
12.7 Vinkki: näppärä tyyppimuunnos string-tyypiksi . . . 147
12.8 Reaalilukujen muotoilu String.Format-metodilla . . . 148
12.8.1 Yhteenveto erilaista tavoista tulostaa . . . 151
12.9 Char-luokka . . . 151

13 Ehtolauseet (Valintalauseet) 153
13.1 Mihin ehtolauseita tarvitaan? . . . 153
13.2 if-rakenne: “Jos aurinko paistaa, mene ulos.” . . . 153
13.3 Vertailuoperaattorit . . . 155
13.3.1 Huomautus: sijoitusoperaattori (=) ja vertailuoperaattori (==) . . . 155
13.4 Esimerkki: yksinkertaisia if-lauseita . . . 155
13.5 if-else -rakenne . . . 156
13.5.1 Esimerkki: Pariton vai parillinen . . . 157
13.6 Loogiset operaattorit . . . 158
13.6.1 Operaattoreiden totuustaulut . . . 159
13.6.2 Operaattoreiden käyttö . . . 159
13.6.3 De Morganin lait . . . 161
13.6.4 Osittelulaki . . . 162
13.7 else if-rakenne . . . 163
13.7.1 Esimerkki: Tenttiarvosanan laskeminen . . . 164
13.7.2 Harjoitus . . . 165
13.7.3 Harjoitus . . . 165
13.8 switch-rakenne . . . 166
13.8.1 Esimerkki: Arvosana kirjalliseksi . . . 167

14 Olioiden ja alkeistietotyyppien erot 170

15 Taulukot **176**
15.1 Taulukon luominen . . . 177
15.2 Taulukon alkioon viittaaminen . . . 178
15.2.1 Funktioita jotka käsittelevät taulukkoa ja niiden testaaminen . . . 179
15.2.2 Muita taulukkoesimerkkejä . . . 183
15.3 Esimerkki: lumiukon pallot taulukkoon . . . 185
15.4 Esimerkki: arvosana kirjalliseksi . . . 186
15.5 Moniulotteiset taulukot . . . 188
15.5.1 Harjoitus . . . 192
15.6 Taulukon kopioiminen . . . 192
15.7 Esimerkki: Moniulotteiset taulukot käytännössä . . . 193
15.8 Taulukoiden täydennykset lisätietosivuilla . . . 199

16 Toistorakenteet (silmukat) **200**
16.1 "Syö niin kauan kuin puuroa on lautasella" . . . 200
16.2 while-silmukka . . . 201
16.2.1 Kohti silmukkaa . . . 201
16.2.2 Esimerkkejä while-silmukoista . . . 203
16.2.3 Huomautus: ikuinen silmukka . . . 204
16.2.4 while-silmukka vuokaaviona . . . 205
16.2.5 Esimerkki: Taulukon tulostaminen . . . 205
16.2.6 Esimerkki: Monta palloa . . . 207
16.3 do-while-silmukka . . . 210
16.3.1 Esimerkki: nimen kysyminen käyttäjältä . . . 211
16.4 for-silmukka . . . 211
16.4.1 Huomautus: while- ja for-silmukoiden yhtäläisyydet ja erot . . . 215
16.4.2 Esimerkki: lumiukon pallot keltaisiksi . . . 216
16.4.3 Harjoitus . . . 217
16.4.4 Esimerkki: Keskiarvo-aliohjelma . . . 217
16.4.5 Harjoitus . . . 218
16.4.6 Esimerkki: Taulukon kääntäminen käänteiseen järjestykseen . . . 218
16.4.7 Harjoitus . . . 220
16.4.8 Esimerkki: arvosanan laskeminen taulukoilla . . . 220
16.4.9 Harjoitus 16.4 . . . 224
16.5 foreach-silmukka . . . 224
16.5.1 Esimerkki: taulukon pallot keltaisiksi . . . 226
16.6 Sisäkkäiset silmukat . . . 226
16.7 Esimerkki: rivi, jolla eniten vapaata tilaa . . . 227
16.8 Silmukan suorituksen kontrollointi break- ja continue-lauseilla . . . 228
16.8.1 break . . . 228
16.8.2 continue . . . 229
16.8.3 return . . . 230
16.9 Poistuminen ennen silmukkaa . . . 230
16.10 Älä tee silmukassa testejä, jotka voi tehdä sen ulkopuolella. . . . 231
16.11 Ohjelmointikielistä puuttuva silmukkarakenne . . . 232
16.12 Yhteenveto . . . 232

17 Merkkijonojen pilkkominen ja muokkaaminen **235**
17.1 String.Split() . . . 235

17.1.1 Tyhjien alkioiden huomiotta jättäminen . . . 236
17.2 String.Trim() . . . 237
17.3 Esimerkki: Merkkijonon pilkkominen ja muuttaminen kokonaisluvuiksi . . . 238
17.4 Komentoriviparametrit . . . 241

18 Järjestäminen **242**

19 Olion ulkonäön muuttaminen (Jypeli) **244**
19.1 Väri . . . 244
19.2 Koko . . . 244
19.3 Tekstuuri . . . 245
19.4 Olion muoto . . . 245

20 Ohjainten lisääminen peliin (Jypeli) **246**
20.1 Näppäimistö . . . 247
20.2 Lopetuspainike ja näppäinohjepainike . . . 248
20.3 Peliohjain . . . 248
20.3.1 Analoginen “tatti” . . . 248
20.4 Hiiri . . . 249
20.4.1 Näppäimet . . . 249
20.4.2 Hiiren liike . . . 250
20.4.3 Hiiren kuunteleminen vain tietyille peliolioille . . . 251

21 Piirtoalusta (Jypeli) **252**
21.1 Esimerkki: Punainen rasti . . . 253
21.2 Esimerkki: Pyörivä jana . . . 253

22 Rekursio **254**
22.1 Sierpinskin kolmio . . . 256
22.2 Harjoitus . . . 261
22.3 Huomautus . . . 261
22.4 Rekursio muilla ohjelmointikielillä . . . 261

23 Dynaamiset tietorakenteet **263**
23.1 Rajapinnat . . . 263
23.2 Listat (List<T>) . . . 264
23.2.1 Tietorakenteen määrittäminen . . . 264
23.2.2 Alkioiden lisääminen ja poistaminen . . . 265
23.2.3 Listan sisäinen rakenne . . . 266
23.2.4 Esimerkki listaa käsittelevästä funktiosta ja sen testaamisesta . . . 267
23.3 Anonyymit funktiot (lambda-lausekkeet) . . . 268
23.3.1 `Find` . . . 269
23.3.2 Delegaatit ja Lambda-lausekkeet . . . 270
23.3.3 `FindIndex` . . . 271
23.3.4 `Exists` . . . 271
23.3.5 `FindAll` . . . 272
23.3.6 Usean lauseen sisältävät anonyymit funktiot . . . 273
23.3.7 Lambda-lausekkeen ulkopuolisten muuttujien käyttö . . . 274
23.3.8 Muita yleisiä valmiita metodeja . . . 274

24 Poikkeukset **276**
24.1 try-catch . . . 276
24.2 finally-lohko . . . 278
24.3 Yleistä . . . 278

25 Tietojen lukeminen ulkoisesta lähteestä **279**
25.1 Tekstin lukeminen tiedostosta . . . 279
25.2 Tekstin lukeminen netistä . . . 281
25.3 Satunnaisluvut . . . 282

26 Lukujen esitys tietokoneessa **283**
26.1 Lukujärjestelmät . . . 283
26.2 Paikkajärjestelmät . . . 284
26.3 Binääriluvut . . . 284
26.3.1 Binääriluku 10-järjestelmän luvuksi . . . 285
26.3.2 10-järjestelmän luku binääriluvuksi . . . 286
26.4 Negatiiviset binääriluvut . . . 290
26.4.1 Suora tulkinta . . . 290
26.4.2 1-komplementti . . . 290
26.4.3 2-komplementti . . . 290
26.4.4 Bittien yhteenlasku . . . 292
26.4.5 2-komplementin yhteenlasku . . . 294
26.5 Lukujärjestelmien suhde toisiinsa . . . 295
26.6 Liukuluku (floating-point) . . . 297
26.6.1 Liukuluvun binääriesityksen muuttaminen 10-järjestelmään . . . 298
26.6.2 10-järjestelmän luku liukuluvun binääriesitykseksi . . . 299
26.6.3 Huomio: doublen lukualue . . . 300
26.6.4 Liukulukujen tarkkuus . . . 300
26.6.5 Intelin prosessorikaan ei ole aina osannut laskea liukulukuja oikein . . . 301

27 ASCII-koodi **302**
27.1 Muut merkistöt . . . 305

28 Syntaksin kuvaaminen **306**
28.1 BNF . . . 306
28.2 Laajennettu BNF (EBNF) . . . 307

29 Jälkisanat **310**

Liite: Sanasto **311**

Liite: Yleisimmät virheilmoitukset ja niiden syyt **313**
Tyyppiä tai nimiavaruutta ei löydy . . . 313
Peli.Aliohjelma(): not all code paths return a value . . . 313
Muuttujaa ei ole olemassa nykyisessä kontekstissa . . . 314
Lähdeluettelo . . . 315

Esipuhe

Arvaa mikä olisi oikea järjestys, jotta alla oleva olisi toimiva ohjelma (vinkki: koita päätellä sulkujen parillisuudesta ja sisennyksistä):

```
1 public class HelloWorld
2 {
3     public static void Main()
4     {
5         System.Console.WriteLine("Tervetuloa opiskelemaan ohjelmointia!");
6     }
7 }
```

Tämä oppimateriaali on niin kutsuttu luentomoniste kurssille Ohjelmointi 1. Luentomoniste tarkoittaa sellaista kirjallista materiaalia, jossa esitetään asiat suurin piirtein samassa järjestyksessä ja samassa valossa kuin ne esitetään luennolla. Jotta moniste ei paisuisi kohtuuttomasti, ei asioita käsitellä missään nimessä kaikenkattavasti. Siksi opiskelun tueksi tarvitaan jokin hyvä aihetta käsittelevä kirja, sekä rutkasti ennakkoluulotonta asennetta ottaa asioista itse selvää. Tuorein tieto löytyy tietenkin internetistä - kunhan muistaa lähdekritiikin. On myös huomattava, että useimmat saatavilla olevat kirjat lähestyvät ohjelmointia tietyn ohjelmointikielen näkökulmasta — 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 järjestelmällisesti alusta loppuun. Aloittaessaan puhumaan lapsi ei kuitenkaan ole kykeneväinen omaksumaan kaikkea tietyn lauserakenteen kieliopista yhdellä kertaa. Vastaavasti ohjelmoinnin alkeita kahlattaessa vastaanottokyky ei vielä riitä kaikkien rakenteiden ja mahdollisuuksien käsittämiseen. Tässä luentomonisteessa ja samoin luennolla asioiden käsittelyjärjestys on sellainen, että asioista annetaan ensin esimerkkejä tai johdatellaan näiden esimerkkien tarpeellisuuteen, ja sitten kerrotaan niin teoreettisesti kuin käytännöllisesti mistä oli kyse. Näin ollen tästä monisteesta saa yhden näkemyksen mukaisen pintaraapaisun ohjelmoinnin alkutaipaleelle. Kirjoista ja nettilähteistä asiaa on kuitenkin syvennettävä.

Tässä monisteessa käytetään esimerkkikielenä *C#*-kieltä. Kuitenkin nimenomaan esimerkkinä, koska monisteen rakenne ja esimerkit voisivat olla aivan samanlaisia mille tahansa muullekin ohjelmointikielelle. Tärkeintä ohjelmoinnin johdantokurssilla on ohjelmoinnin ajattelutavan oppiminen. Kielen vaihtaminen toiseen samansukuiseen kieleen on ennemmin verrattavissa Savon murteen vaihtamiseen Turun murteeseen, kuin suomen kielen vaihtamiseen 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, eikä kannata tyytyä yhteen kieleen, vaan kannattaa opetella useita. Tätäkin vastaavaa kurssia

on pidetty Jyväskylän yliopistossa seuraavilla kielillä: Fortran, Pascal, C, C++, Java ja nyt C#. Joissakin yliopistoissa aloituskielenä on Python, toisissa Scala. Nämä kaikki ovat tietyssä mielessä samansukuisia kieliä ja noudattavat monilta osin samanlaisia periaatteita, vaikka yksityiskohdat vaihtelevat joskus paljonkin.

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

https://tim.jyu.fi/view/kurssit/tie/ohj1/koti

Tämä moniste perustuu Martti Hyvösen ja Vesa Lappalaisen syksyllä 2009 kirjoittamaan *Ohjelmointi 1* -monisteeseen, joka osaltaan sai muotonsa monen eri kirjoittajan työn tuloksena aina 80-luvulta alkaen. Suurimman panoksen monisteeseen ovat antaneet Timo Männikkö ja Vesa Lappalainen.

Jyväskylässä 2.1.2013

Martti Hyvönen, Vesa Lappalainen, Antti-Jussi Lakanen

Esipuheen jälkipuhe

Monisteen uusin versio on kirjoitettu TIM-järjestelmään (The Interactive Material). TIM-järjestelmän ideana on, että asioita, esimerkiksi ohjelmointia, pääsee kokeilemaan ilman mitään ohjelmien asentamista. Tämä toivottavasti helpottaa hieman ohjelmoinnin aloituskynnystä. Valitettavasti käyttämämme tekniikka (kurssille valittu kieli ja aliohjelmakirjastot) eivät anna mahdollisuutta interaktiivisten pelien tekemiseen, joten vakavampaa ohjelmointia varten joudumme kuitenkin asentamaan ohjelmointityökaluja, tässä tapauksessa Visual Studion ja Jypelin. Näistä myöhemmin tässä monisteessa ja muussa kurssin materiaalissa.

Materiaalissa olevista algoritmivisualisaatioista kiitos Aalto-yliopiston ACOS Content Server -projektille.

Jyväskylässä 29.8.2014 *Vesa Lappalainen, Antti-Jussi Lakanen*

Monisteen 2023 versiossa muutetaan Visual Studio viittauksia yleisemmäksi, koska JY:n kursseilla on työkalua vaihdettu Rider-työkaluksi.

Luku 0

Johdanto

Vaikka kurssi onkin tehty "peliohjelmointi"-kurssiksi, on 90% sen sisällöstä täysin samaa asiaa minkä ohjelmointikurssin kanssa tahansa. Jos joku ei halua tehdä kurssin harjoitustyönä peliä, voi toki tehdä myös minkä tahansa muun pienen ohjelman.

0.1 Kurssin sisällöstä ja tavoitteista

Pikaisen idean (*englanniksi*) tämän kurssin sisältöön saat katsomalla videon siitä, miten tehdään alle 5 minuutissa *Galaksit räjähtää* - peli. Jos katsot alla olevia videoita, älä pelkää ettet osaa (vielä), vaan katso mitä sinun pitää kurssin aikana oppia ja opitkin.

Video 1: GalaxyTrip less than 5 minutes, Demonstrated in SIGCSE11 symposium. Antti-Jussi Lakanen/Vesa Lappalainen

Katso video osoitteessa: https://www.youtube.com/embed/cHJ73xVOFD4

Jos haluat samasta aiheesta pidemmän version (suomeksi), niin katso video:

Video 2: Galaksit räjähtää: Pelin tekeminen 45 minuutissa, Antti-Jussi Lakanen, Levels-tapahtuma 9.4.2011

Katso video osoitteessa: https://www.youtube.com/embed/R3KgCkuMEs4

Seuraavista videoista näet millaisia pelejä kursseilla on tehty:

Video 3: Ohjelmointi 1, kevät 2014 -kurssin harjoitustöitä

Katso video osoitteessa: https://www.youtube.com/embed/zF46zbTxdPM

Video 4: Nuorten peliohjelmointikurssi (1 viikko), kesä 2013

Katso video osoitteessa: https://www.youtube.com/embed/sXCCd2NqSoQ

0.2 Kurssin osaamistavoitteet

Kurssin aluksi sinun oletetaan osaavan tietokoneen käyttöä. Tuttuja asioita pitäisi olla muun muassa erilaisten editorien käyttö, näppäinoikotiet sekä mielellään komentorivi. Toki nykypäivänä komentorivi ei valitettavasti ole kovin hyvin tunnettu asia ja voitkin tutustua komentoriviin esimerkiksi kurssin lisätietosivuilta tai Paavon selviytymisoppaasta.

Tarkista tietosi

Mitä seuraavista osaat tehdä komentoriviltä? Vastaamisen jälkeen näytetään hyvin yksinkertainen komentolista, jossa kauttaviivalla erotetaan Windows / Linux ja macOS -ohjeet. Paremmat ohjeet ylläolevissa linkeissä.

	True	False
Avata komentorivin	☐	☐
Liikkua hakemistosta toiseen	☐	☐
Katsoa nykyhakemiston sisältöä	☐	☐
Tehdä uuden hakemiston	☐	☐
Katsoa tekstitiedoston sisältöä	☐	☐
Tehdä uuden tekstitiedoston	☐	☐
Tehdä uuden C#-kielisen tiedoston	☐	☐

Aikaisempaa ohjelmointikokemusta sinulla ei tarvitse olla.

Kurssin aikana sinun on tarkoitus oppia seuraavia asioita (osaamisen taso sovelletulla Bloomin asteikolla: 1=muistaa, 2=ymmärtää, 3=osaa soveltaa, 4=osaa analysoida, 5=osaa arvioida, 6=osaa luoda)

Osattava asia	1	2	3	4	5	6
Rakenteisen ohjelmoinnin perusajatus			o			
Algoritminen ajattelu			o			
C#-kielen perusteet			o			
Peräkkäisyys				o		
Muuttujat						o
Aliohjelmat ja funktiot						o
Parametrin välitys				o		
Ehtolauseet				o		
Silmukat				o		
Taulukot			o			
Tiedostot ohjelmasta käytettynä		o				
Olioiden käyttö			o			
Yksikkötestit (TDD)		o				
Debuggerin käyttö			o			
Lukujärjestelmät, ASCII-koodi		o				
Rekursio	o					
Dokumentointi ja sen lukeminen			o			

0.3 TIM-käyttöohjeita

Alla olevat ohjeet koskevat tämän monisteen interaktiivista nettiversiota, joka on saatavilla osoitteessa https://tim.jyu.fi/view/1. Suosittelemme nettiversion käyttöä printatun monisteen rinnalla. Esimerkiksi malliohjelmien koko koodit saa näkyville vain nettiversiossa.

Yleiset TIM-ohjeet ◘ Luento 1 (2m46s)

Tämä TIM-pohjainen moniste koostuu erilaisista interaktiivista osista. Videoihin jo varmaan edellä tutustuitkin. Laitteesi kapasiteetin säästämiseksi videot kannattaa sulkea katsomisen jälkeen.

Monisteessa voi olla linkkejä muuhun materiaaliin. Nämä linkit on tarkoitettu lisälukemiseksi ja niitä ei kannata seurata kun monistetta käy ensimmäisen kerran lävitse. Linkkiviidakkoon vaan eksyy turhan helposti.

TIM-monisteessa kannattaa aina olla kirjautuneena (Login), niin voit seurata *omaa* edistymistäsi. Kirjautuneille monisteen oikeassa reunassa näkyy punaisia palkkeja niissä kohti, mitä et ole vielä lukenut. Kun olet lukenut (ja ymmärtänyt :-)) jonkin tekstinpätkän, klikkaa punaista palkkia palkin poistamiseksi. Näin näet helposti mitä kohtia sinulla on vielä käymättä. Erityisesti tästä on hyötyä, jos hyppelet monistetta eri järjestyksessä kuin missä se on kirjoitettu. Palkki voi olla myös keltainen silloin, kun olet lukenut kappaleen, mutta sen sisältö on muuttunut viimeisen lukemisesi jälkeen. Klikkaa tämäkin pois jos sisäistät muutetun tekstin. Jos et tykkää toiminnosta, voit rattaan kuvan takaa klikata kaikki punaiset kerralla pois.

Vasemmassa yläkulmassa on kirjan kuva tai näytön koosta riippuen menun kuva jonka takaa löytyy kirjan kuva. Kirjan kuvasta aukeaa sisällysluettelo. Samasta paikasta voi sisällysluettelon sulkea.

Muokkausmenun saa joko klikkaamalla lohkoa, jolloin tulee kynä oikealle tai kun vie hiiren kappaleen vasempaan reunaan tulee sinertävä palkki. Tuo riippuu kummanko tavan on itsel-

leen valinnut. Kynää tai palkkia painamalla aukeaa menu. Menusta saa mm. `Comment/Note` -painikkeen, mistä voit lisätä itsellesi muistiinpanoja kuhunkin kappaleeseen liittyen. Käytä tätä ominaisuutta ahkerasti. Voit laittaa huomioita itsellesi tai huomauttaa, jos jonkin kappaleen sisältö on epäselvä tai virheellinen. Anna mielellään tällaisessa tapauksessa myös korjausehdotus. Muuten käytä harkiten "Everyone" valintaa ja laita omat kommentit "Just me".

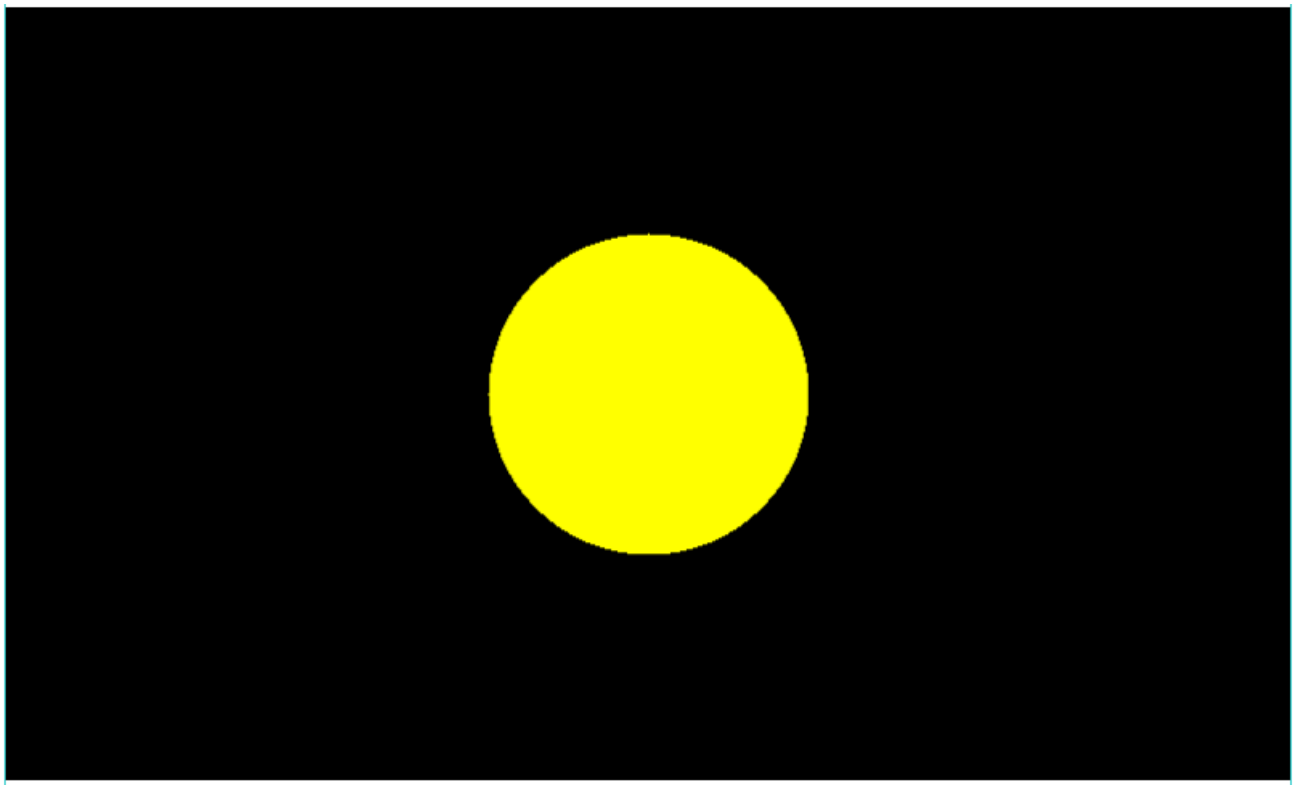
Jos haluat etsiä jotakin, käytä selaimen etsi toimintoa (`Ctrl-F` useimmissa selaimissa).

Jos haluat helposti löytää jonkin sivun uudelleen, niin tee siitä TIMin kirjanmerkki. Kirjanmerkin voit tehdä vasemmassa yläkulmassa "klemmarin" kohdalta. Toki voit tehdä kirjanmerkin selaimeesikin normaalisti, mutta TIMin kirjanmerkin hyvä puoli on siinä, että se toimii missä selaimessa tahansa. Aloita tekemällä tästä sivusta kirjanmerkki itsellesi. Eli paina "klemmarin" kuvaa ja lisää sivu vaikkapa otsikon `Ohj1` alle nimellä `Moniste`.

Edellisissä videoissa ohjelmia kirjoitettiin *Visual Studio* -nimisessä ohjelmointi-ympäristössä (IDE = *Integrated Development Environment*). TIMissä on itsessään pieni sisäänrakennettu ympäristö, jolla voi tehdä yksinkertaisia tehtäviä, esimerkiksi:

Aja ensin alla oleva koodi painamalla Aja-painiketta. Muuta sitten koodia niin, että pallo on punainen. Aja uudelleen. Muuta vielä tausta siniseksi.10pt

```
1        Level.Background.Color = Color.Black;
2        PhysicsObject pallo = new PhysicsObject(200,200,Shape.Circle);
3        pallo.Color = Color.Yellow;
4        Add(pallo);
```



Kun ohjelma on ajettu

Katso tehtävään ohjeita videolta Luento 1 (2m21s)

Tehtävälaatikon alla on *Näytä koko koodi*-linkki, jota painamalla näet kaiken sen koodin, mitä ohjelman takia tarvitaan. Voit edelleen muutella ohjelmaa, mutta et voi kirjoittaa "väärään"

paikkaan. Samasta linkistä voit piilottaa “ylimääräisen koodin”.

Highlight-linkistä voit vaihtaa editorin tyypin sellaiseksi, että se värittää koodia käytettävän kielen syntaksin mukaan sekä osaa täydentää editorille tuttuja sanoja.

Alusta-linkistä voit “nollata” oman vastauksesi ja aloittaa uudelleen mallista. Kokeile kumpaakin linkkiä.

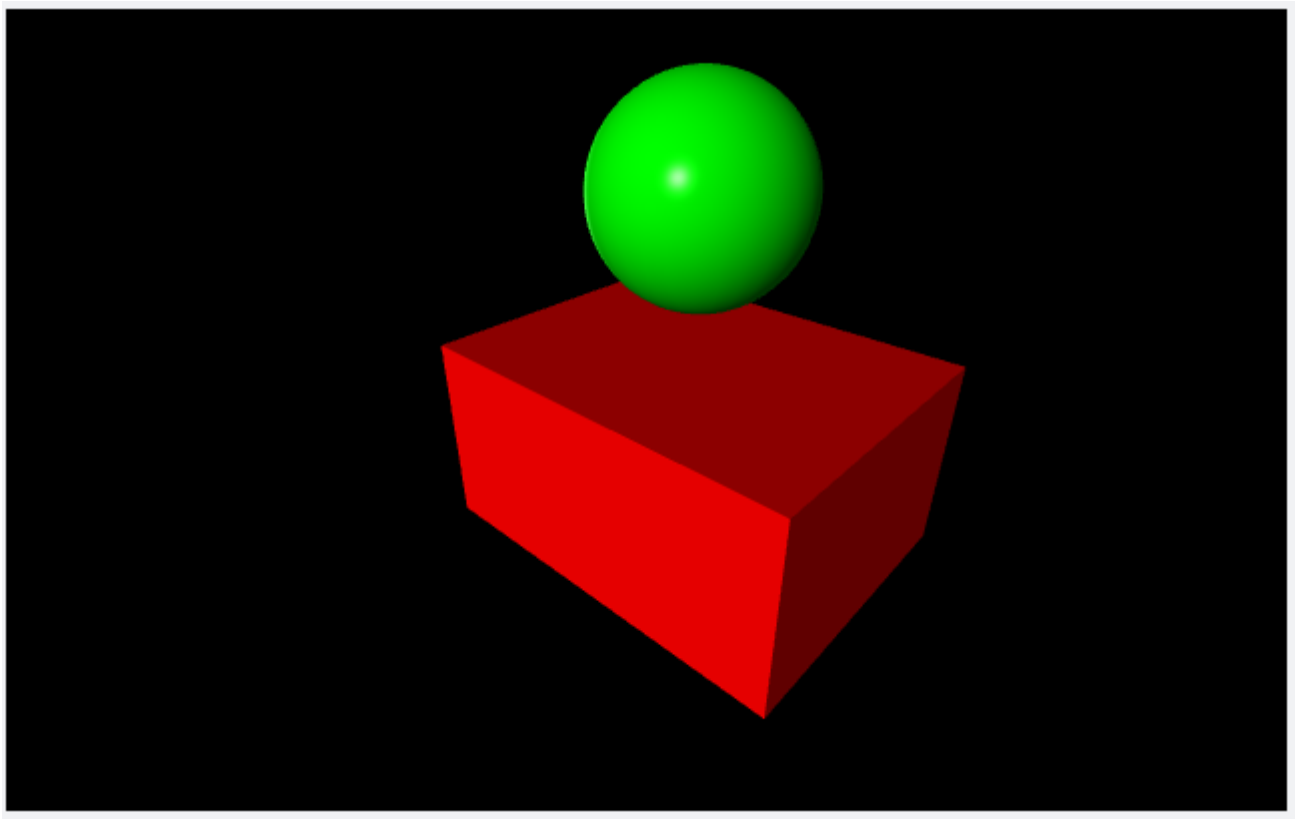
Tehtävä voisi jatkua vielä niin, että: Lisää ennen `Add(pallo)`-riviä rivi

```
pallo.Position = new Vector(150, 100);
```

Kokeile tätäkin, eli copy/paste yllä oleva koodi siihen isompaan koodiin `Add(pallo)` rivin yläpuolelle. Kokeile myös mitä tapahtuu, jos kirjoitat värissä `Red` pienillä kirjaimilla. Korjaa takaisin `Red` ja kokeile mitä vaikuttaa, kun vaihtaa `Vector`issa olevia arvoja.

Voit kokeilla myös toisella kielellä (VPython) tehtyä esimerkkiä. Tätä voit myös pyöritellä hiiren oikealla painikkeella.

```
1 ball=sphere(pos=vector(4,7,3),radius=2,color=color.green)
2 redbox=box(pos=vector(4,2,3),size=vector(8,4,6),color=color.red)
```



Pallo ja kuutio VPython-ympäristössä tehtynä.

Luku 1

Mitä ohjelmointi on?

Sana ohjelmointi sisältää sanan **ohje**.

1.1 Algoritmit eli ohjeet

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. Nykyisin navigaattori lukee ohjeista aina seuraavan kun sitä tarvitaan. Vastaavalla tavalla ohjelmassakin tulee olemaan kohta missä suoritus on menossa. Alkeellista ohjelmointia on tavallaan myös mikroaaltouunin käyttäminen, sillä tällöin uunille annetaan ohjeet siitä, kuinka kauan ja kuinka suurella teholla sen tulee toimia.

Ohjelmointi jakautuu hyvin monelle tasolle. Nykyisin on esimerkiksi traktoreita, joissa maanviljelijä ohjelmoi, miten peltoja kuljetaan. Varotoimenpiteenä ja tiukkoja käännöksiä varten tosin viljelijän pitää vielä itse olla mukana traktorissa varmistamassa, että kaikki sujuu hyvin. Eli tietyssä mielessä viljelijänkin pitää osata ohjelmoida. Mutta ennen kuin traktori on saatu tähän vaiheeseen, on tarvittu valtavasti insinöörityötä ja ohjelmointia. GPS-satelliitit, virheenkorjaus, traktorin varsinaisen tietokoneen ohjelmointi sille tasolle, että se tekee viljelijän ohjelmoinnin helpoksi jne.

Suonenjoella mansikoita keräävällä poimijalla on kaulassaan lähilukukortti (NFC-siru) ja aina kun hän saa tuokkosen täyteen ja vie sen keruupaikalle, rekisteröityy tieto siitä, kuka on kerännyt, mistä on kerännyt ja paljonko on tullut kiloja. Viljelijä on ohjelmoinut taustalle tiedot peltojen sijainneista ja toimenpiteistä ja voi seurata aikaisempaa tarkemmin, milloin joltakin saralta tuotto pienenee ja se kannattaa "alustaa" kokonaan.

Eli itse asiassa tietokoneet ja ohjelmointi tulevat joka paikkaan arkipäivän elämään. Tosin useinkaan käyttäjä ei välttämättä ymmärrä (ja toivottavasti ei tarvitsekaan ymmärtää), että hän käyttää tietokonetta ja ehkä jopa ohjelmoi sitä.

Näissä tapauksissa puhutaan sulautetuista järjestelmistä ja/tai IoT (*Internet of Things*) -laitteista, jos laite on yhteydessä verkkoon, kuten esimerkiksi traktorin ja maanviljelijän tapauksessa.

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 digiboxin ajastaminen kaukosäätimellä, 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.

Vaikka ohjelmointia käytännössä tehdään suurelta osin tietokoneella, on silti kynä ja paperia syytä aina olla esillä. Ohjelmoinnin suurin vaikeus aloittelijalle onkin siinä, että ei malteta istua **kynän ja paperin** kanssa ja miettiä mitä ollaan tekemässä. Jos esimerkiksi pitää tehdä laivanupotuspeli, pitää ensin pelata useita kertoja peliä, jotta hahmottuu, mitä kaikkia asioita tulee aikanaan vastaan.

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.

Oleellista tässä ohjeessa on se, että sitä suoritetaan “käsky” (esimerkissä rivi) kerrallaan. Seuraavaa käskyä ei voida suorittaa ennen kuin edellinen on valmis. Tällöin puhutaan peräkkäisestä ohjelmoinnista. Jotta pysytään selvillä mitä käskyä ollaan tekemässä, pitää jossakin pitää mielessä käsky numero. Tästä paikasta puhutaan jatkossa nimellä käskyosoitin, IP (*instruction pointer*) tai ohjelmalaskurista, PC (*program counter*).

Rinnakkaisessa ohjelmoinnissa voisi olla kaksi kokkia, joista toinen tekisi käskyn 1 sillä aikaa kun toinen tekee käskyn 2. Käskyjä 3 ja 4 ei voi kuitenkaan rinnakkaistaa. Eli välttämättä kaksi kokkia ei saa kakkua valmiiksi puolta nopeammassa ajassa.

Koneelle kirjoitettavat ohjeet poikkeavat merkittävästi ihmisille kirjoitetuista ohjeista. Kone ei osaa automaattisesti kysyä neuvoa törmätessään uuteen ja ennalta arvaamattomaan tilan-

teeseen. 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]

1.2 Ohjelmointikielistä

Tässä aliluvussa kerrotaan mutkia oikoen hieman tietokoneen ideasta ja ohjelmointikielistä. Asiasta tulee tarkemmin ja lisää Tietokoneen rakenne ja arkkitehtuurikurssilla sekä Käyttöjärjestelmät. Asiaa sivutaan myös luvussa Lukujen esitys tietokoneessa.

1.2.1 Prosessori ja konekieli

Tietokoneen tärkeimmät osat ovat prosessori ja muisti. Prosessorin oleellinen ominaisuus on se, että sillä on tiedossa suoritettava käsky. Yleensä tämä tieto on IP-rekisterissä (*Instruction Pointer*, myös PC = *Program Counter* on yleisesti käytetty termi tälle). IP-rekisteri osoittaa koneessa muistipaikkaan, josta löytyy suoritettava käsky. Prosessorin toiminta on periaatteessa hyvin yksinkertaista:

1. hae käsky IP-rekisterin osoittamasta paikasta
2. kasvata IP-rekisterin sisältöä niin, että se osoittaa seuraavan käskyyn
3. suorita haettu käsky (voi muuttaa IP:tä JUMP-käskyillä)
4. jatka kohdasta 1.

Rekisterit ovat prosessorin sisäisiä nopeita muistipaikkoja. Käskyt ovat usein hyvin alkeellisia tyyliin:

- hae luku muistipaikasta 7F34 rekisteriin AX
- lisää rekisteriin AX rekisterin BX arvo

Jokaisella käskyllä on oma numeerinen arvo, joka tietokoneessa tietysti esitetään bitteinä. Esimerkiksi käsky

- laita luku 62 (heksaluku) rekisteriin BL

olisi Intel x86 -sarjan prosessorissa

```
B3 62
```

ja muistissa siis binäärisenä

```
10110011 01100010
```

Eli periaatteessa ohjelmointi olisi saada koneen muistiin noita oikeita binäärilukuja. Koska binäärilukuja on aika vaikea ihmisen hahmottaa, käytetään niille usein edellä olevaa heksalukuesitystä. Tuokaan ei ole ihan helppoa muistaa, että `B3` tarkoittaisi, että "laita BL rekisteriin". Siksi käytetään yleensä assembly-kieltä, jossa on suurin piirtein 1:1 vastaavuus konekielisen binääriluvun ja ihmisen luettavan mnemonicin (muistikas) välillä. Eli eräällä (niitä on monia variantteja) assembly-kielellä edellinen komento olisi

```
mov bl,$62
```

Aluksi tietokoneita ohjelmoitiinkin syöttämällä suoraan käskyjen numeroarvoja. Sitten assembly-kielten myötä ihminen kirjoitti assembly-kieltä ja se käännettiin noiksi numeroarvoiksi ja näin saatiin syntymään koneen muistiin tarvittava ohjelma.

Koska prosessorin käskyt ovat varsin "alkeellisia", tarvitaan niitä paljon yksinkertaisenkin ohjelman tekemiseksi. Erityisesti tiedon lukemiseksi ihmissyötteestä tai tiedostosta. Siksi tarvitaan käyttöjärjestelmä, joka tarjoaa usein tarvittavat ominaisuudet valmiina. Mutta siltikin assembly-kielillä joutuisi kirjoittamaan pieneenkin ohjelmaan paljon koodia.

1950-luvulta lähtien alettiin kehittämään ohjelmointikieliä, joilla ohjelmien kirjoittaminen olisi helpompaa ja selkeämpää kuin assemblerilla. Näin syntyi monia vieläkin käytössä olevia ohjelmointikieliä, kuten Fortran (1957), Lisp (1958), Cobol (1959) ja Pascal (1970). 70-luvulle tultaessa kieliä oli jo kymmeniä ellei jopa satoja, kun pienet kielet lasketaan mukaan.

1.2.2 C-kieli ja robotti

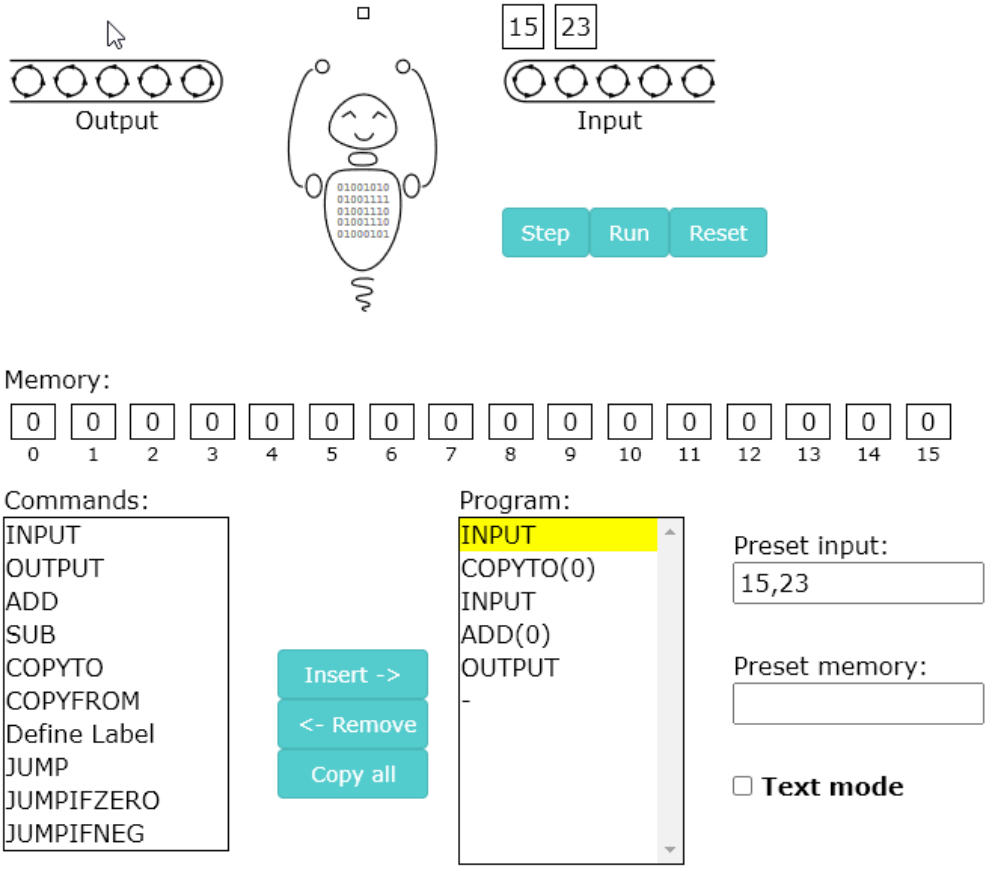
Kielen kääntäjä on ohjelma, joka lukee syötteenään ihmisen kirjoittaman selkokielisen (esim C tai C++ -kieli) ohjelmatiedoston (tekstitiedosto) ja tuottaa siitä binäärimuotoisen suoritettavan (*executable*) konekielisen tiedoston, joka voidaan sitten ajaa. Tämän takia esimerkiksi Windows-järjestelmässä ajettavan tiedoston nimen tarkentimena on usein `.exe`. Kun ohjelma käynnistetään, on käyttöjärjestelmän tehtävä laittaa ohjelmakoodi koneen muistiin ja siirtää ohjelmalaskuri ohjelman ensimmäiseen käskyyn.

Jälkeenpäin tunnetuin 70-lukulainen käännettävä korkeamman tason kieli on C-kieli (1972). Ideana (kuten sen edeltäjissäkin) on nostaa abstraktiota ylemmäksi, eli voidaan suoraan sanoa esimerkiksi:

```
int a = 15;
int b = 23;
int c = a + b;
```

Jos vastaava kirjoitettaisiin konekielellä, joutuisi ohjelmoija itse miettimään mitä kohtaa muistista käyttää muuttujille `a`, `b` ja `c`. C-ohjelmassa (ja kurssin käyttämässä C#) kääntäjä pitää kirjaa tarvittavista muistipaikoista ja aina kun puhutaan muuttujasta `a`, kääntäjä kääntää konekieliseen koodiin viittauksen a:lle varattuun muistipaikkaan.

Kurssin demotehtävissä on esimerkkinä pieni robotti, joka osaa vain muutamia käskyjä. Tämä robotti toimii hyvin vastaavalla tavalla kuin prosessori. Esimerkiksi edellinen C-ohjelman osa (joka itse asiassa tuolta osin on täsmälleen samanlainen C#-kielellä) olisi robotilla:



Robotti

Voit kokeilla robotin toimintaa painamalla `Step`-painiketta. Harjoitustehtävänä voit muuttaa sen laskemaan yhteen kaikki Input-hihnalla olevat luvut (tosin tämä vaatii sopimuksen että esim hihnalla oleva 0 lopettaa laskemisen). Input hihnalle saat uusia lukuja laittamalla ne `Preset input`-kohtaan ja painamalla `Reset`. `Run`-painikkeesta robotti suorittaa kerralla koko ohjelman.

Robotissa `Program`-kohdassa oleva keltainen rivi vastaa prosessorin IP-rekisteriä, eli osoittaa suoritettavaa käskyä.

Käytetty kieli on nyt tavallaan robotin assembly-kieltä.

Jos käskyille annettaisiin numeeriset arvot (joita niillä sisäisesti onkin), esimerkiksi:

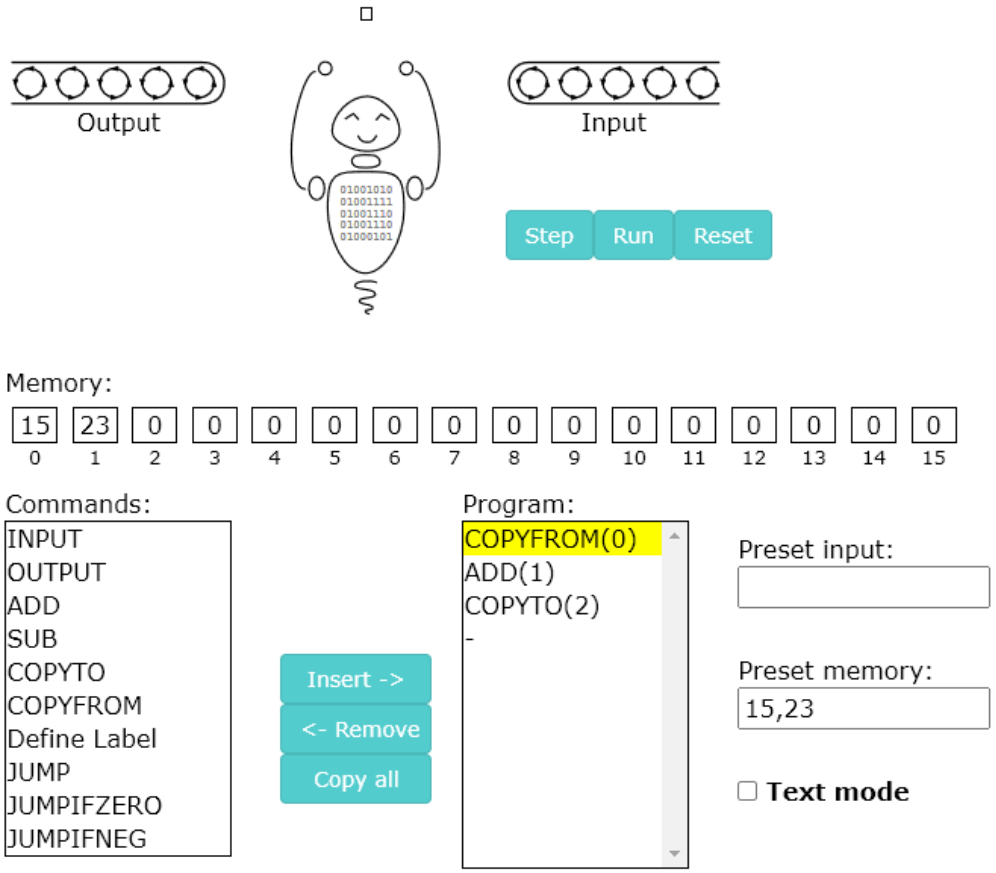
```
00 = INPUT
01 = OUTPUT
02 = ADD
03 = SUB
04 = COPYTO
...
09 = JUMPIFNEG
```

olisi tämä ohjelma robotin “konekielellä”:

```
00 04 00 00 02 00 01
```

jossa siis osa käskyistä vaatii kaksi tavua (tavu on 8 bittiä, esitetään kahden numeron pareina), kuten esim `COPYTO` jossa on käskyn vastaava lukuarvo ja sitten käskyn kohteen osoite (nyt muistipaikka 00).

Sitten meillä voisi olla C-kääntäjä, joka kääntäisi aikaisemmin kuvatun ohjelman osan tuoksi lukujonoksi. Paitsi että muistipaikat `a` ja `b` tuossa tapauksessa kääntyisivät `Input`-hihnalla oleviksi paikoiksi. Toki sama ohjelma voitaisiin tehdä myös muistipaikkoja käyttäen:



Robotti

Tämä vastaisi jo melko tarkoin kirjoitettua C-ohjelmaa. Kääntäjän yksi tehtävä on silloin päättää, että vaikkapa muuttujasta `a` puhuttaessa tarkoitetaan muistipaikkaa `00` ja `b`:stä muistipaikkaa `01`.

Tehtävä: Robotin konekieli

Millainen olisi tämä ohjelma robotin ”konekielellä”? Erota tavut yhdellä välilyönnillä toisistaan.10pt

Robotin käsittelyä luennolla: Luento 2 (8m2s)

1.2.3 Tavukielet

C-kieli oli valtakieli 70-luvun lopulta 80-luvun lopulle. 80-luvun alussa C-kielestä tehtiin alaspäin yhteensopiva oliolla laajennettu kieli `C++` (1982). Myös tämä oli käännettävä kieli. 90-luvulla kehitettiin Java-kieli (1995) alun perin erilaisten sulautettujen järjestelmien kieleksi. Samalla Java paikkasi C++:n tunnettuja ongelmia. Javassa oli C++:aan nähden muutamia merkittäviä eroja:

1. Javaa ei käännetä suoraan konekieleksi, vaan välikieleksi. Välikielistä tiedostoa ajetaan erikseen kullekin prosessorille tehdyllä Java-nimisellä ohjelmalla. Java-ohjelma (Java-virtuaalikone) lukee välikielen tavukoodia (vrt em robotin kielen lukuarvoinen esitys) ja suorittaa sitä askel kerrallaan. Java ei suinkaan ollut ensimmäinen tavukoodiin perustuva kieli, mutta se on tunnetuin tämän hetken virtuaalikoneeseen pohjautuvista kielistä.
2. Javassa on automaattinen muistinhallinta, eli ohjelmoijan ei itse tarvitse muistaa vapauttaa varaamiaan muistialueita. Toki automaattinen muistinhallinta oli jo "tuttua" tekniikkaa vanhemmista kielistä.
3. Javassa ei voi vahingossa osoittaa muistiin, jota ei ole varannut käyttötarkoitukseen (sanotaan ettei Javassa ole osoittimia)

Tavukoodin ideana on, että kääntäjää ei tarvitse tehdä erikseen joka prosessoriarkkitehtuurille ja käyttöjärjestelmälle. Riittää olla yksi kääntäjä, joka tuottaa välikooditiedoston (Javassa yleensä `.class`). Toisaalta ohjelman suorittaminen vaatii sitten välikielen tulkitsemista todellisen prosessorin konekielelle ja aluksi Java-ohjelmat olivatkin hitaampia kuin C-ohjelmat. Nykyisin Java-kääntäjien kehitykseen on panostettu paljon ja lisäksi tavukoodia suoritettaessa sitä käännetään samalla konekielelle (JIT = Just In Time compiling) ja näin jos samaan koodin kohtaan tullaan uudelleen, se onkin valmiiksi käännetty ja suoritusnopeus ei eroa oleellisesti C-koodin suoritusnopeudesta.

Javan suosio ponnahti raketin lailla 90-luvun puolivälin jälkeen. VL:n mielipide syistä:

> "*Syynä oli automaattinen muistinhallinta ja sitä kautta helpommin vähemmän virheitä sisältävän ohjelmakoodin tuottaminen. Lisäksi Javassa oli toimivat merkkijonot, jotka puuttuivat esimerkiksi C++ standardista tuohon aikaan. Asiaa auttoi myös hyvin paljon C:tä muistuttava syntaksi, joka loivensi kielen vaihtoa.*"

Microsoft oli panostanut paljon C++ -kieleen, mutta huomasi Javan suosion nousun ja otti sen myös käyttöönsä, kuitenkin lisäten siihen omia ominaisuuksiaan. Tämä aiheutti lisenssiriitoja Javan kehittäneen Sun-yhtiön kanssa. Tästä syystä Microsoft lähti kehittämään omaa kieltä, jossa olisi kaikki Javan hyvät ominaisuudet. Tuloksena oli C#-kieli (C sharp, 2000). Monilta ominaisuuksiltaan kielet ovat hyvin samankaltaisia ja niiden välillä on aika helppoa ohjelmoijan siirtyä.

1.2.4 C# ja Jypeli

Jyväskylän yliopiston IT-tiedekunnassa ruvettiin miettimään nuorille sopivaa ohjelmointikurssia vuoden 2008-2009 tienoilla. Tällöin oli melko selkeää, että kurssilla pitäisi tehdä pelejä. Microsoftilla oli tällöin hyvät ympäristöt (Visual Studio) ja kirjastot (XNA) tehdä pelejä C#-kielellä ja saada ne toimimaan niin tietokoneissa kuin puhelimissakin (Windows Phone). Suoraan XNA:lla pelien ohjelmointi oli kuitenkin liian haastavaa ja siksi kehitettiin Jypeli-kirjasto, joka peittää alleen "turhia" yksityiskohtia, jotka jarruttaisivat aloittelevan ohjelmoijan ideointia. Tämä Nuorten pelikurssi osoittautui menestykseksi. Samaan aikaan takuttiin Java-pohjaisilla yliopiston ohjelmointikursseilla motivaation kanssa. Monia yliopistotason opiskelijoitakin pelit kiinnostavat ja siksi ensimmäiselle ohjelmointikurssille vaihdettiin teemaksi peliohjelmointi ja siinä samalla oli sujuvaa ottaa käyttöön Jypeli ja kieleksi C#. Tämä nostikin Ohjelmointi 1 -kurssin läpimenoa merkittävästi, kun voitiin tehdä "mielekkäämpiä" ohjelmia. Pelkkä Hello Worldin tulostaminen ei enää herättänyt intohimoa 2010-luvulla.

1.2.5 Muita kieliä

Edellä lueteltiin vain muutamia tunnettuja kieliä, `C`, `C++`, `Java` ja `C#`. Näillä on pitkälle samat sukujuuret. Puhuttiin myös välikielen tulkkaamisesta. Yksi hyvin tunnettu kokonaan alun perin tulkattavasi tehty kieli oli `Basic` (1964). Ideana on silloin että käännösvaihe puuttuu ja ihmisen kirjoittamaa ohjelmakoodia ruvetaan suorittamaan suoraan rivi riviltä. Nykyisin `Python` (1990) on noussut suosituksi tulkattavaksi kieleksi. Erilainen lähestymistapa ohjelmointiin on funktio-ohjelmointi, johon sopivia kieliä ovat esimerkiksi `Haskell` (1990), `Scala` (2004) ja `F#` (2005).

Vastaavasti Javascript on selainten käyttämä kieli, jonka avulla alunperin staattiset HTML-sivut saadaan "elämään". Esimerkiksi tämä luentomoniste pyörii TIM-nimisessä sovelluksessa, jossa Pythonilla ja Haskelilla kirjoitettu palvelinohjelma lähettää selaimella Javascriptiä (1995) ja HTML:ää (1993), joiden avulla selain muodostaa interaktiivisen tekstin. Lisäksi TIMIä kirjoitettaessa käytetään nykyisin Javascriptin tilalla TypeScript-nimistä kieltä (2012), joka käännetään selainta varten Javascriptiksi. 3D-grafiikassa käytetään varjostinkieliä kuten GLSL ja HLSL riippumatta siitä, millä kielillä muut osiot grafiikkaa käyttävästä sovelluksesta kirjoitetaan. Näiden lisäksi tulevat erilaisiin sovelluskohteisiin kehitetyt kielet (DSL, *domain specific language*), joiden lukumäärää kukaan ei voi tietää. Eli käytännön elämässä yhden ohjelman kirjoittamisessa voidaan vaatia useiden eri ohjelmointikielten osaamista.

- Kokeile eri ohjelmointikieliä TIMissä
- HelloWorld eri kielillä, esimerkissä 603 ohjelmointikieltä

Eri kielten suosiosta ja historiasta voi katsoa lisää alla olevista linkeistä. Tosin kielten suosiota voidaan mitata hyvin eri tavoin, joten erilaisiin indekseihin kannattaa suhtautua kriittisesti.

- Tiobe-index
- Animaatio YouTubessa: ▣ Most Popular Programming Languages 1965 - 2019

Tällä kurssilla keskitytään kuitenkin käyttämään esimerkkinä C#-kieltä.

Luku 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*, *Visual Studio Code*, *Sublime Text ja NotePad++*, mutta monet muutkin ovat varmasti hyviä. Monisteen alun esimerkkien kirjoittamiseen soveltuu hyvin mikä tahansa tekstieditori.

> *Koodi, lähdekoodi* = Ohjelmoijan tuottama tiedosto, josta varsinainen ohjelma muutetaan kääntämällä tai tulkkaamalla 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, joka tulee käytetyn ohjelmointikielen nimestä, joten tälläkin kurssilla käytämme tätä tarkenninta. 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.

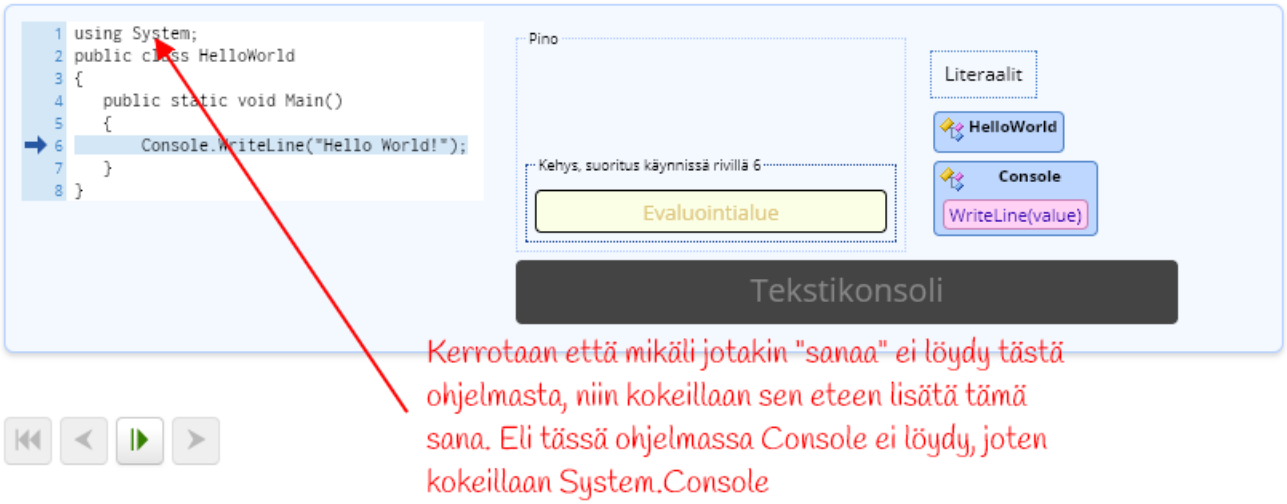
Selvennykseksi vielä video ohjelmakoodin kirjoittamisesta Notepad++:lla Luento 3 (6m40s)

Sekä vastaava Sublime text -editorilla. Luento 1 (2m55s)

```
1 public class HelloWorld
2 {
3     public static void Main()
4     {
5         System.Console.WriteLine("Hello World!");
6     }
7 }
```

Ajamisen jälkeen ohjelma tulostaa seuraavan tekstin komentorivi-ikkunaan.

```
Hello World!
```



Tutki sanojen merkitystä ja ohjelman toimintaa (animaatio verkkoversiossa)

Tämän ohjelman pitäisi tulostaa näytölle teksti

```
Hello World!
```

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

> *Kääntäminen* = Kirjoitetun lähdekoodin muuntaminen suoritettavaksi ohjelmaksi.

Kun painat tässä TIM-monisteessa `Aja`-painiketta, niin aluksi ohjelma käännetään konekieliseen muotoon ja sitten jos kääntäminen onnistuu virheittä, ohjelma ajetaan ja näytetään mitä se tulosti. Näistä vaiheista lisää seuraavassa alaluvuissa. Sitä ennen kuitenkin muutamia tehtäviä joissa voit kokeilla “taitojasi”.

Esimerkkejä muilla ohjelmointikielillä kirjoitetusta HelloWorld -ohjelmasta löydät vaikkapa:

http://www2.latech.edu/~acm/HelloWorld.html.

Tehtävä 2.1

Muuta alla olevaa koodin osaa niin, että se tulostaa oman nimesi yhdelle riville ja kotipaikkakuntasi toiselle riville. Jos haluat tulostaa kaksi riviä, niin laita tulostuslause kaksi kertaa.10pt

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

Muuta tehtävä tulostamaan ISOLLA oma etunimesi. Saat käyttää vain asteriski (*)-merkkiä ja välilyöntiä. 10pt

```
1         System.Console.WriteLine("*******      *       *     *");
2         System.Console.WriteLine("   *         *       * * * *");
3         System.Console.WriteLine("   *         *       *  *  *");
4         System.Console.WriteLine("   *         *       *     *");
```



Edellisen esimerkin voisi tehdä myös seuraavasti10pt

```
1        System.Console.Write("*******      *       *     *\n" +
2                             "   *         *       * * * *\n" +
3                             "   *         *       *  *  *\n" +
4                             "   *         *       *     *\n"  );
```

Kokeile mitä edellä tapahtuu (ja miksi?) jos jättää kirjaimet `\n` pois rivien lopuista.

2.2 Ohjelman kääntäminen ja ajaminen

Jotta ohjelman kääntäminen ja suorittaminen onnistuu, täytyy koneelle olla asennettuna joku C#-sovelluskehitin. Aluksi riittää asentaa Microsoftin .NET-kehitysympäristö, jonka mukana tulee `dotnet`-komento, jonka avulla voidaan kääntäminen ja ajaminen suorittaa.

Esimerkiksi tämä käyttämäsi TIM-ympäristö on toteutettu (Python, Haskell ja Javascript/TypeScript-kielillä) niin, että ruutuun kirjoittamasi teksti annetaan Linux-palvelimelle, joka tallettaa tiedoston tilapäistiedostoon ja kääntää sen edellä mainitulla `dotnet`-komennolla. Jos käännös menee virheittä, syntynyt konekielinen ohjelma ajetaan Linux-palvelimessa ja kaapataan ohjelman tuottama tulostus ja näytetään se selaimen ruudussa. Nämä vaiheet vievät yhteensä muutaman sekunnin.

Lisätietoa .NET-kehitystyökaluista ja asentamisesta löytyy kurssin kotisivuilta kohdasta Työkalut.

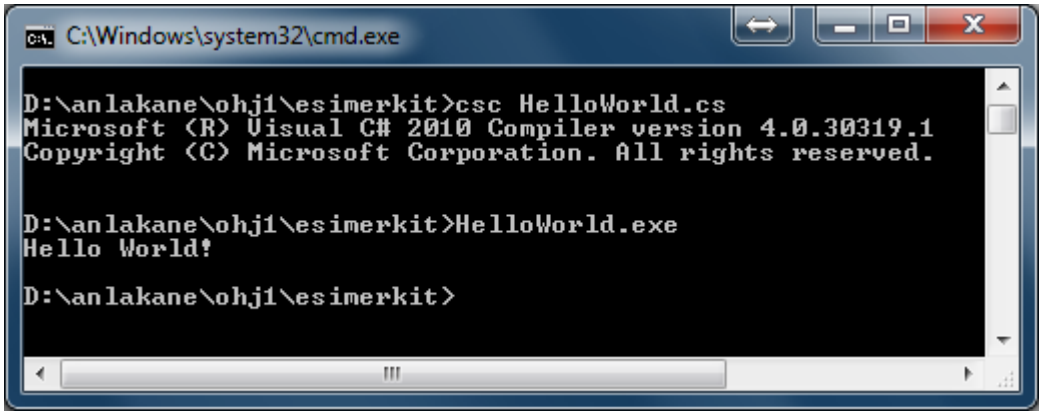
Seuraavaksi opettelemme tekemään nämä vaiheet käsin, jotta ymmärtäisimme paremmin mitä taustalla tapahtuu.

HelloWorld-ohjelman kääntäminen komentorivillä Luento 3 (-1h5m1s)

Kääntäjän versiot vaihtuvat helposti vuosittain, samoin miten niitä käytetään. Ajantasaisimman esimerkin kääntämisestä löydät harjoituksesta:

- Pääteohjaus 1, HelloWorld (syksy 2023)

Jos noudatit yllä olevan linkin ohjeita, ohjelman tulisi nyt tulostaa näyttöön teksti `Hello World!`.



Kuva 1: Ohjelman kääntäminen ja ajaminen Windowsin komentorivillä.

Tehtävä 2.2

Avaa uuteen ikkunaan (ctrl+klikkaa linkkiä) oheinen materiaali ja tee siellä olevat tehtävät. Vastaa sitten alla olevaan testiin.

https://tim.jyu.fi/view/kurssit/tie/ohj1/materiaali/Kaantaminen

Mitkä komennot pitää antaa uudelleen kun lähdekoodia on muokattu?

	True	False
tallennus	☐	☐
kääntäminen	☐	☐
sisällysluettelon katsominen	☐	☐
hakemiston luominen	☐	☐
ajaminen	☐	☐

2.3 Ohjelman rakenne

Vaikka ensimmäisen ohjelmamme “ainoa oleellinen rivi” onkin

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

tarvitaan C#-kielessä tämän ympärillä tietoa siitä, mihin ohjelman osaan lause kuuluu sekä mistä kohti ohjelma pitää käynnistää. Tämä hieman lisää sinänsä yksinkertaisen ohjelma koodirivien määrää. Joissakin kielissä tulostavaan ohjelmaan riittää pelkkä tulostuslause. Rivimäärien ero pienenee ohjelman koon kasvaessa. Yleisesti ottaen rivien vähyys ei ole itseisarvo, joten sen perusteella ei pelkästään voi kieliä laittaa paremmuusjärjestykseen.

Kirjoittamamme ohjelma `HelloWorld.cs` (tai oikeastaan kirjoittamamme tekstitiedosto) on melkein yksinkertaisin mahdollinen C#-ohjelma. Alla yksinkertaisimman ohjelman kaksi ensimmäistä riviä.

```
public class HelloWorld
{
```

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 tarvittaessa luokkaan kirjoitetun koodin avulla. Olioita voidaan myös tuhota. Yhdellä luokalla voidaan siis tehdä monta samanlaista oliota, aivan kuten yhdellä piparkakkumuotilla voidaan tehdä monta samanlaista (melkein 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! C#-kielessä luokka aloitetaan isolla alkukirjaimella. Skandeja (ääö yms) ei kannata käyttää luokan nimessä.

Tässä tulostuslauseen 'System' on kirjoitettuna pienellä. Jos koitat ajaa sitä, se ei käänny vaan antaa virheilmoituksen. Muuta ohjelma toimivaksi. Kokeile muuttaa muitakin merkkejä isoiksi tai pieniksi. 10pt

```
1        system.Console.WriteLine("Tässä kohtaa tulostetaan kirjaimet sellaisenaan.↩
     ");
```

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 C#-kielessä 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 `static`-aliohjelmia), jolloin `public`-sana kirjoitetaan lähes aina **luokan** ja `static`- **aliohjelman** eteen. Koska oletukset saantimääristä vaihtelevat eri kielissä, on selvyyden vuoksi varminta ne aina kirjoittaa.

> Huomaa kuitenkin, että kun jatkossa tulee puhetta olion omista muuttujista (eli *attribuuteista*), niin niiden eteen kirjoitetaan lähes poikkeuksetta `private`. Jatkossa puhutaan myös metodeista (olion omista "tempuista") ja niiden suojausmääre pitää miettiä tilannekohtaisesti sen mukaan, halutaanko kenen tahansa voivan käskeä olion tekemään jotakin.

Luokat ja `static`-aliohjelmat esitellään yleensä saantimääreellä `public`. Attribuutit esitellään vastaavasti `private`-määreellä.

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()
{
```

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. `static` tarkoittaa, että aliohjelma on **luokkakohtainen** (vastakohtana **oliokohtainen**, jolloin `static`-sanaa ei kirjoiteta). Vastaavasti `void` merkitsee, ettei aliohjelma palauta mitään tietoa. Paneudumme näihin määreisiin tarkemmin myöhemmin. `Main` voisi myös palauttaa arvon ja silloin `void` tilalla olisi `int`, mutta tätä ominaisuutta emme käytä tällä kurssilla.

Samoin kuin luokan sisältö, niin myös pääohjelman sisältö kirjoitetaan aaltosulkeiden sisään. C#:ssa ohjelmoijan kirjoittaman koodin suorittaminen alkaa aina käynnistettävän luokan pääohjelmasta (`Main`). 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 `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 (mutta ei siis ohjelmakoodissa). 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. Lauseiden väliin kirjoitetuilla tyhjillä merkeillä (engl. *white space*), kuten välilyönneillä, tabulointimerkeillä tai rivinvaihdoilla ei C#:ssa ole merkitystä ohjelman toiminnan kannalta. Ohjelmakoodin luettavuuden kannalta tyhjillä merkeillä on kuitenkin suuri merkitys. Siksi koodiin ei esimerkiksi kannata turhaan kirjoittaa ylimääräisiä rivinvaihtoja.

Huomaa myös, että puolipisteen unohtaminen on yksi yleisimmistä ohjelmointivirheistä ja tarkemmin sanottuna *syntaksivirheistä.*

Syntaksi = Tietyn ohjelmointikielen (esimerkiksi C#:n) kielioppisäännöstö. Katso myös luku Syntaksin kuvaaminen.

Tehtävä 2.3

Alla on vasta suunnitelma siitä, millainen ohjelma haluttaisiin tehdä.

```
//
julkinen luokka LuokanNimi{

  julkinen luokkakohtainen ei-palauta-mitään Pääohjelma(){

     Tulosta("Nimi");
  }

}
```

Huomaa että alla olevassa esimerkissä muuttujan `a` arvo saadaan tulostettua muodostamalla uusi merkkijono, joka yhdistää plus-operaattorilla toisen jonon ja a:n arvon. Näin `WriteLine`-aliohjelmalle saadaan vietyä parametrina vain yksi merkkijono kuten kuuluukin. WriteLine-aliohjelmalle ei perusmuodossa viedä pilkulla eroteltua listaa kuten joissakin kielissä.

Tehtävä 2.4

Kokeile mihin kaikkiin kohtiin voit koodissa laittaa ylimääräisen välilyönnin tai jopa rivinvaihdon niin, että ohjelma toimii vielä oikein.10pt

```
public class Tyhjia
{
    public static void Main()
    {
       int a = 3;
       System.Console.WriteLine("a:n arvo on " + a);
       a++;  // Kasvattaa a:ta yhdellä
       System.Console.WriteLine("ja nyt se on yhtä isompi: " + a);
    }
}
```

Tarkista tietosi

Mihin kohti saa laittaa välilyönnin tai rivinvaihdon C#-kielessä?

	True	False
rivin alkuun	☐	☐
ennen rivin ensimmäistä kirjainta	☐	☐
keskelle sanaa	☐	☐
aaltosulun jommallekummalle puolelle	☐	☐
Ennen tai jälkeen välimerkin	☐	☐
public sanan eteen	☐	☐
++ operaattorin + merkkien väliin	☐	☐
++ operaattorin etu- tai takapuolelle	☐	☐

Tarkista tietosi

Mitkä väittämät pitävät paikkaansa koskien tehtävän 2.4 ohjelmaa.

	True	False
Ohjelmassa on neljä lausetta jotka loppuvat puolipisteeseen.	☐	☐
Luokan nimi voisi olla tyhjä (=puuttua kokonaan)	☐	☐
Ohjelmassa on yksi pääohjelma ja kaksi aliohjelmaa	☐	☐
Luokan nimen saa valita itse	☐	☐
Pääohjelman nimen saa valita itse	☐	☐
Pääohjelmassa on yksi lohko	☐	☐

2.3.1 Virhetyypit

Ohjelmointivirheet voidaan jakaa karkeasti *syntaksivirheisiin* ja *loogisiin virheisiin*.

Edellä tutkittiin mihin välilyönnin tai rivinvaihdon voi laittaa. Silloin kun ohjelma ei kääntynyt, oli kyseessä syntaksivirhe. Silloin kun ohjelma toimi, mutta tekstinä näytti erilaiselta, on kyseessä oikeastaan kirjoitustyylin virhe (tai mielipide-ero).

Syntaksivirhe estää ohjelman kääntymisen vaikka merkitys eli *semantiikka* olisikin periaatteessa oikein. Siksi ne huomataankin aina viimeistään ohjelmaa käännettäessä. Syntaksivirhe voi olla esimerkiksi joku kirjoitusvirhe tai puolipisteen unohtaminen lauseen lopusta. Katso myös luku Syntaksin kuvaaminen. Nykyään voi olla myös muitakin virheitä, jotka estävät kääntymisen, kuten esimerkiksi tyyppivirheet (vaikkapa yritetään sijoittaa double-tyyppinen arvo

kokonaislukuun).

Loogisissa virheissä semantiikka, eli merkitys, on väärin. Ne ovat vaikeampia 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, jonka ajoissa havaitseminen ja korjaaminen kuitenkin esti isot tuhot:

https://fi.wikipedia.org/wiki/Y2K.

Esimerkki ajonaikaisesta virheestä. Ohjelma tulostaa mitä 10 jaettuna 2:lla on. Kokeile ajaa ohjelma. Jos jakajaksi (2) laitetaankin 0, tulee ajonaikainen virhe, koska nollalla ei voi jakaa. Kokeile. 10pt

```
1         int jakaja = 2;
2         System.Console.WriteLine("10/" + jakaja + "=" + 10/jakaja );
```

```
10/2=5
```

2.3.2 Kääntäjän virheilmoitusten tulkinta

Alla on esimerkki syntaksivirheestä `HelloWorld`-ohjelmassa.

```
1 public class HelloWorld
2 {
3     public static void Main()
4     {
5         System.Console.Writeline("Hello World!");
6     }
7 }
```

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 puolipiste väärään paikkaan. Koeta ensin itse löytää mihin, ennen kuin jatkat tai kokeilet.

```
1 public class HelloWorld
2 {
3     public static void Main();
4     {
5         System.Console.WriteLine("Hello World!");
6     }
7 }
```

Virheilmoitus, tai oikeastaan virheilmoitukset, näyttävät kääntäjästä riippuen esimerkiksi 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.

Mikäli virhe ei löydy ilmoitetulta riviltä, kannattaa sitä usein lähteä etsimään edellisiltä riveiltä.

Tehtävä 2.5

Kokeile edellä olevia ja muita mahdollisia virhetyyppejä alla olevaan ohjelmaan. Muista että Alusta-linkistä saat ohjelman taas toimivaksi.10pt

```
1 public class Virheita
2 {
3     public static void Main()
4     {
5        int a = 5;  // Vaihda tähän kokeeksi iso A
6        System.Console.WriteLine("a:n arvo on " + a);
7     }
8 }
```

```
a: n arvo on 5
```

Lisää virheilmoitusten tulkintaesimerkkejä on kurssin lisämateriaalissa.

2.3.3 Tyhjät merkit (White spaces)

Kuten aikaisemmassa tehtävässä kokeilimme, esimerkkinämme ollut HelloWorld-ohjelma voitaisiin, ilman että sen toiminta muuttuisi, vaihtoehtoisesti kirjoittaa myös seuraavassa muodossa.

```
1    public class HelloWorld
2                               {
3
4
5          public static void Main()
6      {
7 System.Console.WriteLine("Hello World!");
8        }
9
10
11   }
```

Edelleen, koodi voitaisiin kirjoittaa myös seuraavasti.

Tehtävä 2.6

Korjaa rivitykset ja sisennykset.10pt

```
1 public class HelloWorld { public static void Main() {
2 System.Console.WriteLine("Hello World!"); } }
```

Tai jopa niin, että koko koodi on yhdellä rivillä, kokeile.

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 sovitut 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 lisätietosivulta osoitteesta

> https://tim.jyu.fi/view/kurssit/tie/ohj1/materiaali/koodauskaytanteet

Merkkijono kirjoitetaan lainausmerkkien " väliin. Merkkijonoja käsiteltäessä välilyönneillä, tabulaattoreilla ja rivinvaihdoilla on kuitenkin merkitystä. Vertaa alla olevia tulostuksia.

```
1         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 !
```

```
1         System.Console.WriteLine("H e l l o    W o r l d !");
```

Lukemisen helpottamiseksi tyhjiä merkkejä käytetään rivien alussa sisentämään lohkoja. Tapana on, että jokaisen aloittavan aaltosulun jälkeen sisennetään koodia 4 yksikköä ja vastaavasti saman verran tullaan takaisin lopettavan aaltosulun jälkeen. Parina olevat aaltosulut pyritään (C#-tyylissä) laittamaan samaan sarakkeeseen. Yleensä IDEt osaavat muotoilla koodin ja tätä ominaisuutta kannattaa käyttää, jos ei itse osaa muotoilla koodia kauniisti.

2.4 Kommentointi

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

C# -kielessä on kolme erilaista kommenttityyppiä ja sitä kautta neljä erilaista merkintää näiden käyttämiseen:

merkintä	tarkoitus
`//`	yhden rivin kommentti
`///`	dokumentaatiokomentti
`/*`	monirivisen kommentin alku
`*/`	monirivisen kommentin loppu

Kommentointiin ja dokumentointiin kuuluu myös ohjelman kirjoittamisen käytänteiden noudattaminen (*code conventions*), mm. oikeanlainen sisentäminen ja muuttujien yms. hyvä nimeäminen. Pitää ajatella ohjelmakoodia sellaisena, että toinen kielen tunteva osaa sitä lukea.

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 ähkäillä, että miksihän tuonkin tuohon kirjoitin.

Kääntäjä jättää kommentit huomioimatta, joten ne eivät vaikuta ohjelman toimintaan.

```
// 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 jatkuu kunnes vastaan tulee asteriski ja vinoviiva (*/). Huomaa, ettei asteriskin ja vinoviivan väliin tule välilyöntiä.

Luettavan koodin ohjeet Luento 1 (8m3s)

Tehtävä 2.7

Kokeile erilaisia kommentteja seuraavaan ohjelmaan eri paikkoihin.10pt

```
1 public class HelloWorld
2 {
3     public static void Main()
4     {
5         System.Console.WriteLine("Hello World!");
6     }
7 }
```

Esimerkiksi kommenttijonon `/* kissa */` voit kirjoittaa kaikkiin samoihin paikkoihin, mihin aikaisemmassa harjoituksessa pystyit laittamaan välilyönnin. Vastaavasti et voi kirjoittaa jonoa paikkoihin, joihin ei saa laittaa 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 esimerkiksi nettiselaimen avulla tai tuottaa siitä siisti paperituloste.

Dokumentaatiokommentti olisi syytä kirjoittaa ennen jokaista luokkaa, pääohjelmaa, aliohjelmaa ja metodia (aliohjelmista ja metodeista puhutaan myöhemmin). Lisäksi jokainen C#-tiedosto pitäisi 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.

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.

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

```
/// <summary>Tämä on dokumentaatiokommentti</summary>
```

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:

```
1  /// @author  Antti-Jussi Lakanen
2  /// @version 28.8.2012
3  ///
4  /// <summary>
5  /// Esimerkkiohjelma, joka tulostaa tekstin "Hello World!"
6  /// </summary>
7  public class HelloWorld
8  {
9    /// <summary>
10   /// Pääohjelma, joka hoitaa varsinaisen tulostamisen.
11   /// </summary>
12   public static void Main()
13   { // Suoritus alkaa siis tästä, ohjelman "entry point"
14     // seuraava lause tulostaa ruudulle
15     System.Console.WriteLine("Hello World!");
16   } // Ohjelman suoritus päättyy tähän
17 }
```

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. Paina edellä `Document`-linkkiä ja tutki syntyvää dokumentaatiota painamalla siinä olevia linkkejä. Kaikki “muuttuva” teksti tuossa dokumentaatiossa kerätään ohjelmassa olevista `///` alkavista dokumentaatiokommenteista.

Dokumentaatiokommenttien ansiosta ohjelmasta saadaan aikanaan vastaava dokumentaatio kuin Jypelistä.

> Huomaa että dokumentaatiokomenttimerkkiä `///` ei käytetä muuta kuin dokumenttikommenteissa (eli aliohjelman tai luokan edessä). Koodin sisällä käytetään tavallista yhden rivin komenttimerkkiä `//` tai monen rivin kommenttimerkiä `/* ... */`.

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]

Tehtävä 2.8

Lisää ohjelmaan dokumentaatiokommentit luokan ja pääohjelman edelle.

```
1  public class Tyhjia
2  {
3      public static void Main()
4      {
5          int a = 3;
6          System.Console.WriteLine("a:n arvo on " + a);
7          a++; // a kasvaa yhdellä
8          System.Console.WriteLine("ja nyt se on yhtä isompi: " + a);
9      }
10 }
```

Tarkista tietosi

Mitkä seuraavista käsitteistä on hallussa? Kertaa tarvittaessa

	True	False
Ohjelman kääntäminen	☐	☐
Ohjelman ajaminen	☐	☐
Luokka	☐	☐
Pääohjelma	☐	☐
Lause	☐	☐
Kommentointi	☐	☐
Lähdekoodi	☐	☐
Lohko	☐	☐
Syntaksivirhe	☐	☐
Looginen virhe	☐	☐

Luku 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.
  1.4   Sulje hana.
2.  Keitä vesi:
  2.1.  Aseta pannu hellalle.
  2.2.  Kytke virta keittolevyyn.
  2.3.  Anna lämmetä, kunnes vesi kiehuu.
  2.4   Sammuta virta.
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

Tehtävä 3.1 Teen keittäminen

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? 10pt

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.

Tässä yritetään lisätä palloa ennen kuin se on luotu. Se ei ole mahdollista ja siksi ohjelma ei käänny. 10pt

```
1         Add(pallo);
2         Level.Background.Color = Color.Black;
3         PhysicsObject pallo = new PhysicsObject(200,200,Shape.Circle);
4         pallo.Color = Color.Yellow;
5         // Siirrä pallon lisäys tänne (eli eka rivi Add(pallo);)
```

```
!!! Error code 1
/prg.cs(8,13): error CS0841: A local variable `pallo' cannot be used
before it is declared
Compilation failed: 1 error(s), 0 warnings
```

Tässä määritellään taustaväri ja olion väri useaan kertaan. Viimeisin jää voimaan. 10pt

```
1
2         Level.Background.Color = Color.Black;
3         Level.Background.Color = Color.Blue;
4         PhysicsObject pallo = new PhysicsObject(200,200,Shape.Circle);
5         pallo.Color = Color.Yellow;
6         pallo.Color = Color.Black;
7         Add(pallo);
```

Otetaan seuraavaksi esimerkki eräästä algoritmista. Oletetaan, että sinulla on tilanne, jossa on taulukko lukuja ja kaikille taulukon luvuille pitäisi saada sama arvo kuin taulukon ensimmäiselle luvulle. Voit seuraavassa tehtävässä tehdä tälle “algoritmin” käyttämällä Tauno-ohjelmaa (=TAUlukot NOhevasti).

Taunossa raahaa taulukon alkioita niin, että sinulla on lopuksi haluamasi tulos. Katso samalla minkälaista koodia Tauno sinulle generoi. Tämä on C#-kielinen *algoritmi* tehtävän tekemiseksi. Jos haluat aloittaa Tauno-tehtävän alusta, piilota ja näytä Tauno uudelleen.

Taunon käytöstä löytyy myös video Luento 1 (3m10s)

Tauno

Mieti onko edellä tekemäsi Tauno-vastaus sellainen, missä suoritettavien lauseiden järjestyksen saisi vaihtaa? Jos on, koodi on tässä tapauksessa *rinnakkaistuvaa*, jos järjestyksen vaihtaminen taas rikkoisi “algortimin”, niin koodi on puhtaasti *peräkkäistä*.

Rinnakkaisuus tarkoittaa sitä, että periaatteessa lauseita voisi suorittaa yhtaikaa. Rinnakkainen ohjelmointi on kuitenkin haastavaa ja sitä ei käsitellä tällä kurssilla enempää.

> Tee Taunolla ohjelma, jolla kolme ensimmäistä alkiota ovat samoja kuin ensimmäinen alkio ja kolme viimeistä samoja kuin viimeinen. 10pt

Luku 4

Yksinkertainen graafinen C#-ohjelma

Seuraavissa esimerkeissä käytetään Jyväskylän yliopistossa kehitettyä *Jypeli-ohjelmointikirjastoa.* Alunperin kirjasto suunniteltiin ja toteutettiin *Nuorten Peliohjelmointi* -kurssille, mutta sen todettiin hyvin sopivan myös Ohjelmointi 1 -tasoiselle kurssille. Kirjaston voit ladata koneelle osoitteesta

https://tim.jyu.fi/view/kurssit/tie/ohj1/tyokalut/tyokalut,

4.1 Mikä on kirjasto?

C#-ohjelmat koostuvat luokista. Luokat taas sisältävät metodeja (ja aliohjelmia/funktioita), 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 jossakin vaiheessa tutustua, sillä sieltä löytyy monia todella hyödyllisiä metodeja. Dokumentaatio löytyy Microsoftin sivuilta osoitteesta

https://learn.microsoft.com/fi-fi/dotnet/.

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

Tehtävä 4.1 console-luokan metodit

Etsi `System`-nimiavaruuden `Console`-luokka. Mitä muita

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

Luentovideolta voi katsoa kuinka yksinkertaisen olion saa aikaiseksi: Video (8m34s)

Piirretään lumiukko käyttämällä Jypeli-kirjastoa. Katso sen tekeminen videolta Lumiukko Macillä (7m21s)

```
// Otetaan käyttöön Jyväskylän yliopiston Jypeli-kirjasto
using Jypeli;

/// @author  Vesa Lappalainen, Antti-Jussi Lakanen
/// @version 22.12.2011
///
///
/// <summary>
/// Luokka, jossa harjoitellaan piirtämistä lisäämällä ympyröitä ruudulle
/// </summary>
public class Lumiukko : PhysicsGame
{

  /// <summary>
  /// Pääohjelmassa laitetaan "peli" käyntiin Jypelille tyypilliseen tapaan
  /// Jos käytä dotnet-komentoa tai Rideria, pyyhi Main-aliohjelma pois
  /// </summary>
  public static void Main()
  {
    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.Background.Color = Color.Black;

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

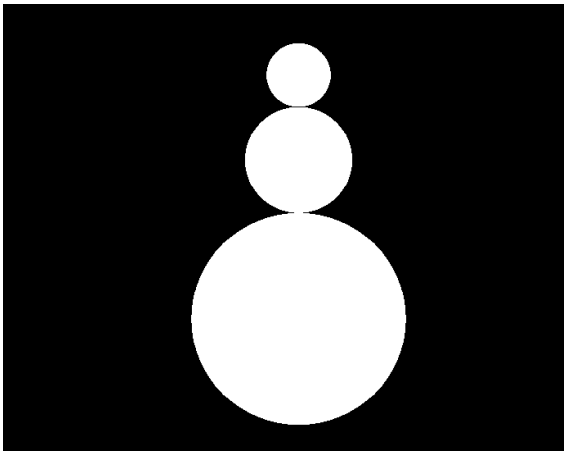
    PhysicsObject p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
    p2.Y = p1.Y + 100 + 50;
    Add(p2);

```

```
43      PhysicsObject p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
44      p3.Y = p2.Y + 50 + 30;
45      Add(p3);
46    }
47  }
```

Myöhemmässä selostuksessa viitataan tämän ohjelman rivinumeroihin. Ne saat näkyviin kun painat `Highlight`-linkkiä.

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



Kuva 2: Lumiukko Jypeli-kirjaston avulla piirrettynä

Jatkoa varten hieman lyhennämme ohjelmaa ja aina samanlaisena toistuvan pääohjelman kirjoitamme omaan erilliseen tiedostoonsa. Näin voimme paremmin keskittyä pelkästään itse ongelmaan. Kokeile lisätä lumiukkoon neljäs pallo.

Tehtävä 4.2 neljäs pallo

Lisää lumiukkoon neljäs pallo10pt

```
1       Camera.ZoomToLevel();
2       Level.Background.Color = Color.Black;
3
4       PhysicsObject p1 = new PhysicsObject(2*100.0, 2*100.0, Shape.Circle);
5       p1.Y = Level.Bottom + 200.0;
6       Add(p1);
7
8       PhysicsObject p2 = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
9       p2.Y = p1.Y + 100 + 50;
10      Add(p2);
11
12      PhysicsObject p3 = new PhysicsObject(2 * 30.0, 2 * 30.0, Shape.Circle);
13      p3.Y = p2.Y + 50 + 30;
14      Add(p3);
```

4.3.1 Ohjelman suoritus

Ohjelman suoritus aloitetaan aina pääohjelman avaavasta aaltosulusta, ja sitten edetään rivi riviltä ylhäältä alaspäin aina pääohjelman sulkevaan aaltosulkuun saakka, ellei erikseen joillakin

ohjauslauseilla (kuten `if`, `while` tms.) muuta sanota. Tässä ohjelmassa ei sanota. Pääohjelmassa (samoin kuin kaikissa muissakin aliohjelmissa) voi olla myös aliohjelmakutsuja, jolloin siirrytään pääohjelmasta suorittamaan aliohjelmaa ja palataan sitten takaisin pääohjelman (kutsuvan aliohjelman) suoritukseen. Aliohjelmista puhutaan enemmän luvussa 6. Itse asiassa edellisissä esimerkeissäkin kutsu `Add(p1)` oli aliohjelmakutsu.

Tarkastellaan ohjelman oleellisimpia kohtia.

```
02 using Jypeli;
```

Aluksi meidän täytyy kertoa kääntäjälle, että haluamme ottaa käyttöön koko Jypeli-kirjaston. Nyt Jypeli-kirjaston kaikki luokat (ja niiden metodit) ovat käytettävissämme. Itse asiassa meidän ei olisi pakko kirjoittaa tätä `using`-lausetta. Mutta jos jätämme sen pois, ei kääntäjä enää tunne mikä on esimerkiksi sana `PhysicsGame`. Ongelma voitaisiin kiertää sanomalla että se löytyy kirjastosta `Jypeli`:

```
11 public class Lumiukko : Jypeli.PhysicsGame
```

Ja samalla tavalla `Jypeli.` pitäisi lisätä kaikkien muidenkin `Jypeli`ssä olevien sanojen eteen. Eli helpotamme omaa kirjoittamistamme sanomalla, että käytetään `Jypeli`ä. Itse asiassa, jos olisimme `HelloWorld.cs` -tiedostossa sanoneet alussa:

```
using System;
```

olisi riittänyt kirjoittaa tulostamista varten:

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

Mutta jatketaan ohjelman tutkimista:

```
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. Näin `Lumiukko`-luokka saa käyttöönsä kaikki `PhysicsGame`-luokan ominaisuudet ja voi itse lisätä siihen uusia ominaisuuksia. Tässä lisäämme tuon `Begin`-metodin toiminnan, eli mitä "pelin" alussa piirretään. `Begin` onkin tavallaan Jypeli-ohjelman "pääohjelma".

Tuon `PhysicsGame`-luokan avulla objektien piirtäminen, myöhemmin liikuttelu ruudulla ja fysiikan lakien hyödyntäminen on vaivatonta.

```
14   /// <summary>
15   /// Pääohjelmassa laitetaan "peli" käyntiin Jypelille tyypilliseen tapaan.
16   /// </summary>
17   public static void Main()
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 tarvitse juurikaan koskea. Ohitamme tässä vaiheessa pääohjelman sisällön mainitsemalla vain, että pääohjelmassa `Lumiukko`-luokasta luodaan uusi olio (eli uusi "peli"), joka sitten laitetaan käyntiin `peli.Run()`-kohdassa. Käytettäessä dotnet-alustaa, Jypelin mallit luovat erikseen `Ohjelma.cs`-tiedoston, jossa on pääohjelma. Varsinainen muu koodi on omassa esimerkiksi Lumiukko.cs -nimisessä tiedostossa. Jypeli-kirjaston rakenteesta johtuen kaikki varsinainen peliin liittyvä koodi kirjoitetaan omiin aliohjelmiinsa. Seuraavaksi käsiteltävään `Begin`-aliohjelmaan kirjoitetaan se, mitä tapahtuu "pelin" alkaessa.

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

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

Näistä kahdesta rivistä ensimmäisellä kutsutaan `Camera`-olion `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.

Huomattakoon että `Camera` ja `Level` -oliot ovat `Lumiukko`-luokasta luodun pelin (pääohjelmassa `peli`) omia olioita. Oikeastaan pitäisikin kirjoittaa:

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

mutta viitattaessa olion omiin ominaisuuksiin, voidaan `this.` -itseviittaus jättää kirjoittamatta. Jotkut ohjelmoijat kirjoittavat silti selvyyden vuoksi myös tuon itseviittauksen näkyviin, vaikka sitä ei välttämättä tarvittaisi. Tämä on tyypillinen makuasia ohjelmoinnissa.

```
33      PhysicsObject p1 = new PhysicsObject(2*100, 2*100, Shape.Circle);
34      p1.Y = Level.Bottom + 200;
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. Jos x-koordinaatin (tai y-koordinaatin) arvoa ei anneta, on se oletuksena 0.

Tarkemmin sanottuna luomme uuden `PhysicsObject`-*olion* eli `PhysicsObject`-luokan *ilmentymän*, johon viittaavan muuttujan 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 (`Circle`), jonka säde on `100 (leveys = 2 * 100 ja korkeus = 2 * 100)`. Muita `Shape`-kokoelmasta löytyviä muotoja ovat muiden muassa kolmio (`Triangle`), ellipsi (`Ellipse`), suorakaide (`Rectangle`), sydän (`Heart`) jne. Olioista puhutaan lisää luvussa 8.

Tehtävä 4.3 olion muoto

Kokeile muuttaa olion muotoa:10pt

```
1         Level.Background.Color = Color.Black;
2         PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
3         pallo.Color = Color.Yellow;
4         Add(pallo);
```

Kokeile muuttaa olion kokoa:10pt

```
1         Level.Background.Color = Color.Black;
2         PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
3         pallo.Color = Color.Yellow;
4         Add(pallo);
```

Seuraavalla rivillä asetetaan olion paikka Y-arvon avulla:

```
34      p1.Y = Level.Bottom + 200;
```

Huomaa että `Y` kirjoitetaan isolla kirjaimella. Tämä on `p1`-olion ominaisuus. 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.

Kokeile muuttaa olion X- ja Y-koordinaatteja. Kuinka saisit pallo-olion oikeaan yläkulmaan?10pt

```
1         Level.Background.Color = Color.Black;
2         PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
3         pallo.Color = Color.Yellow;
4         pallo.Y = 0;
5         pallo.X = 0;
6         Add(pallo);
```

Koordinaatti voidaan antaa myös vektori-muodossa, jolloin annetaan koordinaatin molemmat komponentit samalla kertaa. Esimerkiksi edellisessä tehtävässä pallo voitaiisiin sijoittaa paikkaan `x=20`, `y=50` myös koodilla:

```
1         pallo.Position = new Vector(20,50);
```

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

```
35      Add(p1);
```

Tarkkaan ottaen tässäkin pitäisi kirjoittaa että lisäämme olion tähän peliin, eli:

```
35      this.Add(p1);
```

mutta kuten edellä sanottiin, itseviittaukset voidaan jättää myös kirjoittamatta.

Tästä esimerkistä puuttuu Add-metodin kutsu, eikä kentälle siksi lisätä mitään. Lisää metodin kutsu koodin loppuun ja aja ohjelma uudelleen. Kokeile laittaa kutsun eteen myös itseviittaus this.10pt

```
        Level.Background.Color = Color.Black;
        PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
        pallo.Color = Color.Yellow;
```

Metodeille annettavia tietoja sanotaan *parametreiksi* (parameter). `ZoomToLevel`-metodi ei ota vastaan yhtään parametria, mutta `Add`-metodi sen sijaan ottaa yhden parametrin: `PhysicsObject`-tyyppisen olion, joka halutaan kentälle lisätä. `Add`-metodille voidaan antaa toinenkin parametri: *tasonnumero*, jolle olio lisätään. Tasojen avulla voidaan hallita, missä järjestyksessä oliot piirretään ruudulle. Tasolla ei siis ole fysiikan ominaisuuksia (eli törmäyksien) kannalta merkitystä, ainoastaan kappaleiden ollessa päällekkäin, kumpi näkyy päälimmäisenä. Tasoparametri voidaan myös jättää antamatta, jolloin kappale lisätään oletuksena tasoon 0.

Tässä on tehty kaksi oliota, mutta toinen peittää toisen. Olioiden tasonumerot ovat samat (0) ja siksi neliö peittää pallo-olion. Vaihda pallon tasonumeroksi 1 ja aja ohjelma uudelleen. 10pt

```
        Level.Background.Color = Color.Black;

        PhysicsObject nelio = new PhysicsObject(200, 200, Shape.Rectangle);
        nelio.CollisionIgnoreGroup = 1; // Ei haluta että kappaleet törmäävät ↩
    toisiinsa.
        Add(nelio, 0);

        PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
        pallo.Color = Color.Red;
        pallo.CollisionIgnoreGroup = 1; // Samaan ryhmään kuuluvat eivät törmää
        Add(pallo, 0);
```

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.

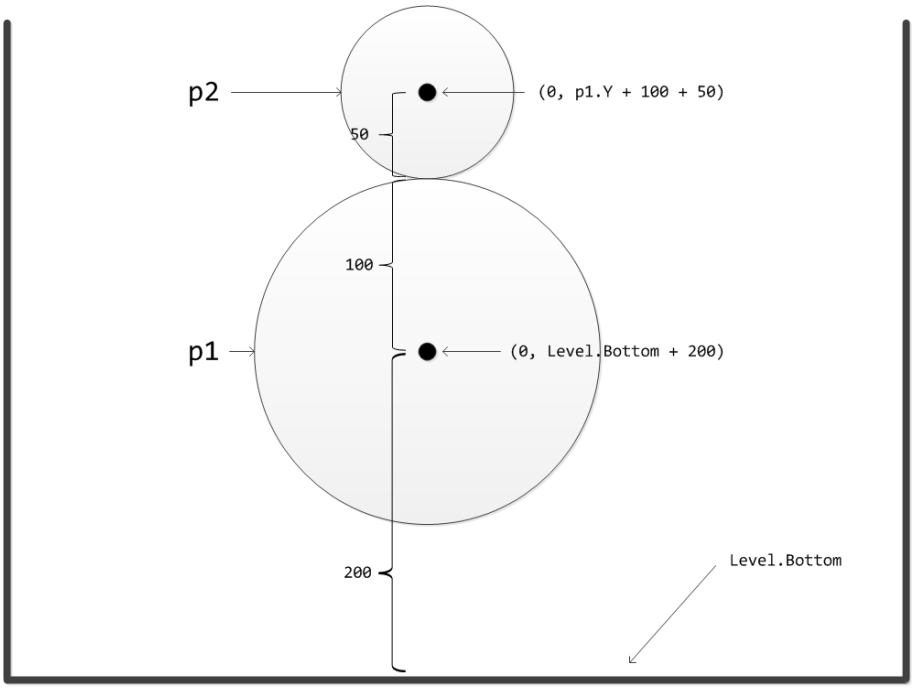
Lumiukko-esimerkissä koordinaattien laskemiseen on käytetty C#:n *aritmeettisia operaatioita*. Voisimme tietenkin laskea koordinaattien pisteet myös itse, mutta miksi tehdä niin, jos tietokone voi laskea pisteet puolestamme? C#:n aritmeettiset perusoperaatiot ovat summa (`+`), vähennys (`-`), kerto (`*`), 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, 2 * 50, 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/.

Tehtävä 4.4 olion paikka vektorilla

Kokeile olion paikan vaihtamista kutsulla

```
1        Level.Background.Color = Color.Black;
2        PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
3        pallo.Color = Color.Yellow;
4        Add(pallo);
```

Tässä ohjelma piirtää nopan. Kokeile muuttaa nopalle muita silmälukuja. 10pt

```
1        Level.Background.Color = Color.Black;
2
3        double koko = 200;
4        GameObject nelio = new GameObject (koko, koko, Shape.Rectangle);
5        Add(nelio);
6
7        GameObject simmu1 = new GameObject(koko/4, koko/4, Shape.Circle);
8        simmu1.Color = Color.Black;
9        simmu1.X = nelio.X - koko/4;
10       Add(simmu1,1);
11
12       GameObject simmu2 = new GameObject(koko/4, koko/4, Shape.Circle);
```

```
13          simmu2.Color = Color.Black;
14          simmu2.X = nelio.X + koko/4;
15          Add(simmu2,1);
```

4.4 Harjoitus

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

Tehtävä 4.5 paikan arvonta

Tutki miten pallo sijoittuu eri ajokerroilla. Kokeile osaatko laittaa pallolle satunnaisen värin.10pt

```
1           Level.Background.Color = Color.Black;
2           PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
3           pallo.X = RandomGen.NextInt(-200, 200);
4           pallo.Y = RandomGen.NextInt(-200, 200);
5           Add(pallo);
6           System.Console.WriteLine(pallo.X +" " + pallo.Y);
```

Seuraavassa esimerkissä on kerrottu, miten käytetään suoraan C#-kirjaston satunnaislukugeneraattoria.

Tehtävä 4.6 jakauma

Alla olevalla koodilla tutkitaan minkälainen jakauma tulee, kun arvotaan lukuja välille [0,MAX[. Kokeile. Miten kävisi, jos tekisit rahanheittopelin?10pt

```
1           int MAX = 6;
2           System.Random rnd = new System.Random();
3           int[] t = new int[MAX];
4           for (int i=0;i<1000; i++)
5           {
6               int n = rnd.Next(0,MAX);
7               t[n]++;
8           }
9           System.Console.WriteLine(string.Join(" ",t));
```

```
169 188 157 174 151 161
```

Kun ajat edellisen ohjelman tulostuu taulukon t alkiot. Kukin niistä vastaa sitä, kuinka monta kertaa tämä luku arvottiin. Miltä noiden suhde näyttää?10pt

4.5 Kääntäminen ja luokkakirjastoihin viittaaminen

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

Ennen komentoriviltä kääntämistä tarvitaan mm. eri Jypelin kirjastoja käyttöön. Nyt osa kirjastoista voi olla eri nimisiä, aikaisemmin tarvittiin mm:

- Jypeli.dll
- Jypeli.Physics2d.dll
- MonoGame.Framework.dll

Meidän täytyy vielä välittää kääntäjälle tieto siitä, että Jypeli-kirjastoa tarvitaan Lumiukko-koodin kääntämiseen. Tämä tehtiin aikaisemmlla csc-ohjelman versiolla `/reference`-parametrin avulla. Lisäksi tarvittiin referenssi Jypelin käyttämään MonoGame-kirjastoon. Silloin kääntämiskomento oli

```
csc Lumiukko.cs /reference:Jypeli.dll;Jypeli.Physics2d.dll;MonoGame.Framework.dll
```

Koska näin komennoista tulisi varsin pitkiä ja sitä varten Microsoft on tehnyt `dotnet`-nimisen ohjelman, jolla voidaan hallita näitä tarvittavien kirjastojen suhteita. Tämän ohjelman avulla kääntämisen vaiheet ovat seuraavat

1. Yhden kerran asennetaan Jypelin tarvitsemat kirjastot, eli annetaan komentoriviltä komento

   ```
   dotnet new install Jypeli.Templates
   ```

   Tätä ei tarvitse enää antaa toista kertaa

2. Siirrytään luodaan tarvittaessa ja siirrytään hakemistoon, johon uusi projekti halutaan

   ```
   cd HAKEMISTOPOLKU
   ```

3. Luodaan uusi projekti Lumiukkoa varten

   ```
   dotnet new Fysiikkapeli -n Lumiukko
   ```

4. Tässä syntyy `Lumiukko`-hakemistoon mm `Lumiukko.cs` niminen tiedosto, joka muokataan halutulla tavalla toimivaksi.

5. Käännetään ja ajetaan ohjelma

   ```
   dotnet run
   ```

6. Jos ei toimi halutulla tavalla, muokataan tiedostoa ja käännetään ja ajetaan uudelleen.

Sama asia käsiteltynä luennolla: Luento 2 (7m50s)

Lisätietoa `dotnet`- kommenon toiminnasta ja sen tuottamista tiedostoista löydät dokumentista dotnet tarkemmin.

Luku 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*-ympäristöä, tapahtuu kääntäminen sisäisesti kahdessa vaiheessa. Ohjelma käännetään ensin välikielelle, *CIL*:lle (Common Intermediate Language), joka ei ole vielä suoritettavissa millään käyttöjärjestelmällä. Tästä välivaiheen koodista käännetään ajon aikana valmis ohjelma halutulle käyttöjärjestelmälle ja prosessorille. Käyttöjärjestelmä voi olla esimerkiksi Windows, macOS, iOS, Android tai Linux. Prosessori voi olla esimerkiksi joku Intel x86-arkkitehtuurin mukainen prosessori tai mobiileissa vaikka ARM. Tämä ajonaikainen kääntäminen suoritetaan 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]

HelloWorld-tyylisen ohjelman kääntäminen tehtiin Windowsissa komentorivillä (esim Git Bash) käyttämällä komentoa

```
csc Tiedostonnimi.cs
```

tai hyödyntämällä edellisessä luvussa esiteltyä dotnet-komentoa tekemällä pelkkä käännös

```
dotnet build
```

5.2 Suorittaminen

C#-kääntäjä tuottaa siis lähdekoodista suoritettavan (tai “ajettavan”) tiedoston. Tämä tiedosto sisältää käyttöjärjestelmästä riippumattomalle välikielelle käännetyn ohjelman. Ohjelman suorittamiseen tarvitaan käyttöjärjestelmäkohtainen *.NET-ajoympäristö*, joka kääntää ajon aikana välikielen käyttöjärjestelmän ja prosessorin ymmärtämään muotoon.

C#-kääntäjää voi myös ohjeistaa tuottamaan käyttöjärjestelmäriippuvaisen suoritettavan tiedoston. Tämä tiedosto on suoritettavissa vain sillä alustalla, johon käännös on tehty. Toisin sanoen, Windows-ympäristössä käännetyt C#-ohjelmat eivät ole välttämättä ajettavissa macOS:ssa, ja toisin päin. Tässä tilassa .NET-ajoympäristöä ei tarvitse erikseen asentaa, vaan se on pakattu mukaan suoritettavaan ohjelmaan.

Samoin kuin C#-kielestä, eräistä muistakin ohjelmointikielistä niiden kääntäjät voivat tuottaa 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]

Luku 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. Aliohjelma voi kutsua toista aliohjelmaa ja joskus jopa itseään (tällöin puhutaan rekursiosta). Oikea ohjelma koostuu useista aliohjelmista joista jokainen suorittaa oman pienen tehtävänsä. Näin iso tehtävä voidaan jakaa joukoksi pienempiä helpommin hallittavia alitehtäviä.

Aliohjelmia tehdään, koska

- niiden avulla voidaan jakaa ohjelma pienempiin osiin
- niiden avulla voidaan jäsentää ohjelmaa
- ne auttavat uudelleenkäytössä
- pienemmät osat helpottavat testaamista

Nykyisten oliokielten oliot ovat oikeastaan kokoelma olion sisäisiä muuttujia (attribuutteja) ja niitä käsitteleviä aliohjelmia (metodeja). Lisäksi nykyisten kielten API (*Application Programming Interface*) on usein huomattavasti itse kieltä suurempi. Kieleen kuuluvien aliohjelmakirjastojen lisäksi usein käytetään sovelluskohtaisia kirjastoja, jotka voivat olla hyvinkin laajoja. Tällä kurssilla esimerkkinä tällaisesta on `Jypeli`. Valmiin kirjaston käyttö helpottaa ohjelman kirjoittajaa ja hänen ei tarvitse kirjoittaa itse kaikkea.

Toisaalta myös itse kirjoitetaan aliohjelmia. Käytännössä usein käy niin, että ohjelmaan kirjoitetaan osa, joka kohta toistuu lähes samanlaisena. Tällöin ohjelmoija pyrkii löytämään koodin yhteisen osan ja siirtää sen aliohjelmaksi. Jos toiminnat eivät samankaltaisissa osissa olleet täysin samanlaiset, toimitetaan ero aliohjelmille parametreina. Näin sama aliohjelma voi eri kutsuilla tehdä hieman eri asioita. Otamme tästä kohta esimerkin.

Toisaalta monesti aliohjelmia tulee myös siitä, että ohjelmaa kirjoitettaessa ajatellaan tyyliin: *"nyt pitäisi löytää taulukon suurin luku"*. Useimmiten ei ole järkevää tällöin lähteä itse etsimistä kirjoittamaan, vaan esitetään toive: *"olisipa meillä aliohjelma joka tekee tuon"*. Ja kirjoitetaan:

```
iso = Suurin(taulukko);
```

Myöhemmin sitten toteutetaan tuo `Suurin` -aliohjelma (funktio tässä tapauksessa, koska se palauttaa arvon). Nyt jos sama tehtävä pitää tehdä uudelleen, ei tarvitse enää kirjoittaa muuta kuin kutsu tuohon aliohjelmaan (*uudelleenkäyttö*).

Usein samaa aliohjelmaa kutsutaan ohjelmasta useita kertoja, mutta koodin selkeyden vuoksi voi olla järkevää kirjoittaa aliohjelmaksi myös itsenäisiä kokonaisuuksia (*jäsentäminen*), vaikkei niitä kutsuttaisikaan kuin kerran koko ohjelmasta.

Seuraavana esimerkki jäsentämisestä, uudelleenkäytöstä ja selkeyttämisestä.

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

```
1  using Jypeli;
2
3  /// <summary>
4  /// Piirretään lumiukko.
5  /// </summary>
6  public class Lumiukko : PhysicsGame
7  {
8    /// <summary>
9    /// Aliohjelma, jossa
10   /// piirretään ympyrät.
11   /// </summary>
12   public override void Begin()
13   {
14     Camera.ZoomToLevel();
15     Level.Background.Color = Color.Black;
16
17     PhysicsObject p1, p2, p3;
18
19     // Eka ukko
20     p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
21     p1.Y = Level.Bottom + 200;
22     Add(p1);
23
24     p2 = new PhysicsObject(2 * 50, 2 * 50, Shape.Circle);
25     p2.Y = p1.Y + 100 + 50;
26     Add(p2);
27
28     p3 = new PhysicsObject(2 * 30, 2 * 30, Shape.Circle);
29     p3.Y = p2.Y + 50 + 30;
30     Add(p3);
31
32     // Toinen ukko
33     p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
34     p1.X = 200;
35     p1.Y = Level.Bottom + 300;
36     Add(p1);
37
38     p2 = new PhysicsObject(2 * 50, 2 * 50, Shape.Circle);
39     p2.X = 200;
40     p2.Y = p1.Y + 100 + 50;
41     Add(p2);
42
43     p3 = new PhysicsObject(2 * 30, 2 * 30, Shape.Circle);
44     p3.X = 200;
45     p3.Y = p2.Y + 50 + 30;
46     Add(p3);
47   }
48 }
```

Huomataan, että ensimmäisen ja toisen lumiukon piirtäminen tapahtuu lähes samanlaisella

koodilla. Itse asiassa ainoa ero on, että jälkimmäisen lumiukon pallot saavat ensimmäisestä lumiukosta eroavat koordinaatit. Ensimmäinen vaihe on yrittää saada molempien lumiukkojen piirtämisestä täysin samanlainen koodi.

Aluksi voisimme kirjoittaa koodin 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. Muistetaan lisäksi, että voimme kirjoittaa olion omiin ominaisuuksiin viitattaessa `this` -viitteen.

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

// Tehdään ensimmäinen lumiukko
x = 0; y = Level.Bottom + 200;
p1 = new PhysicsObject(2*100, 2*100, Shape.Circle);
p1.X = x;
p1.Y = y;
this.Add(p1);

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

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

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

```
// Tehdään toinen lumiukko
x = 200; y = Level.Bottom + 300;
p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
p1.X = x;
p1.Y = y;
this.Add(p1);

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

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

Tarkastellaan nyt muutoksia hieman tarkemmin.

```
double x, y;
```

Yllä olevalla rivillä esitellään kaksi liukuluku*tyyppistä 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.

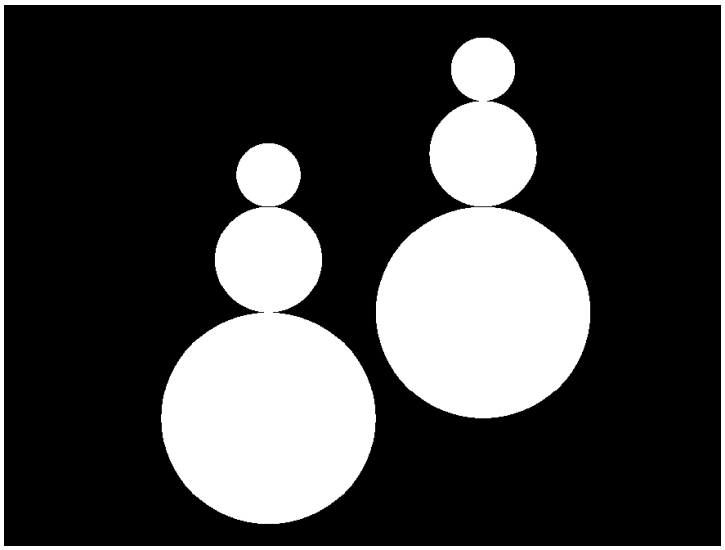
```
x = 0; y = Level.Bottom + 200;
```

Yllä olevalla rivillä on kaksi lausetta. Ensimmäisellä asetetaan muuttujaan `x` arvo 0 ja toisella muuttujaan `y` arvo 50 (jos `Level.Bottom` sattuu olemaan vaikka -150). Nyt voimme käyttää lumiukon pallojen laskentaan näitä muuttujia.

```
x = 200; y = Level.Bottom + 300;
```

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** rivistä `x=` eteenpäin.

Uusien lumiukkojen piirtäminen olisi nyt jonkin verran helpompaa, sillä meidän ei tarvitse kuin ilmoittaa ennen piirtämistä uuden lumiukon paikka, ja varsinainen lumiukkojen piirtäminen onnistuisi kopioimilla ja liittämällä koodia (copy-paste). Kuitenkin, jos koodia kirjoittaessa joutuu tekemään suoraa kopiointia, pitäisi pysähtyä miettimään, 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 vain yhteen paikkaan (hyvä) vai joudutaanko muutoksia tekemään useaan paikkaan (huono).

6.1 Aliohjelman kutsuminen

Parametrittoman metodin tekeminen Luento 2 (4m56s)

TeeLumiukko-metodi parametreilla Luento 2 (8m39s)

Näytelmä siitä mitä aliohjelman kytsominen tarkoittaa Luento 3, 2018s (33m38s)

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 pelikentältä kutsumalla metodia `Destroy()`, eli kirjoittaisimme:

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

Aja ensin ohjelma. Pitäisi piirtyä neliö ja ympyrä (pallo). Lisää ohjelman loppuun rivi, jolla tuhoat ympyrän kutsumalla Destroy-metodia ja aja uudelleen. Tuhoa vielä myös neliö.10pt

```
1  Level.Background.Color = Color.Black;
2  PhysicsObject nelio = new PhysicsObject(200, 100, Shape.Rectangle);
3  nelio.X = -200;
4  nelio.Color = Color.Blue;
5  Add(nelio, 0);
6
7  PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
8  pallo.X = nelio.X + 250;
9  pallo.Color = Color.Yellow;
10 Add(pallo, 0);
```

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.

6.1.1 Aliohjelmakutsun kirjoittaminen

Päätetään, että aliohjelman nimi on `PiirraLumiukko`. Sovitaan myös, että aliohjelman ensimmäinen parametri on tämä 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)`, seuraavalla kutsulla:

```
PiirraLumiukko(this, 0, Level.Bottom + 200);
```

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);
```

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.Background.Color = Color.Black;

  PiirraLumiukko(this, 0, Level.Bottom + 200);
  PiirraLumiukko(this, 200, Level.Bottom + 300);
}
```

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.

Lisätietoja aliohjelmien kutsumisesta löydät dokumentista `Aliohjelmien kutsuminen`:

https://tim.jyu.fi/view/kurssit/tie/ohj1/materiaali/aliohjelmienKutsuminen.

6.2 Aliohjelman kirjoittaminen

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

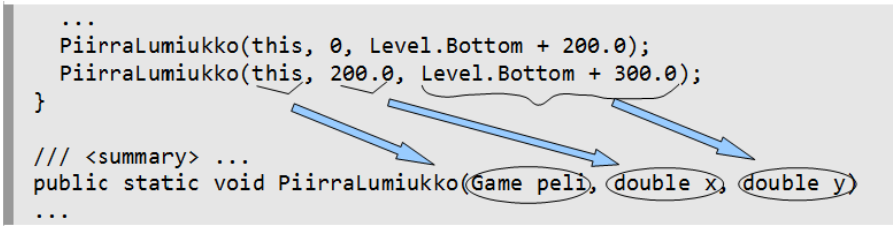
Lisätään ohjelmaamme aliohjelman runko. Dokumentoidaan aliohjelma myös saman tien.

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

  PiirraLumiukko(this, 0, Level.Bottom + 200);
  PiirraLumiukko(this, 200, Level.Bottom + 300);
}

/// <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.



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 esitellään aliohjelman *näkyvyys* julkiseksi (*public*). Kun näkyvyys on julkinen, aliohjelmaa voidaan kutsua eli käyttää myös muissa luokissa.

Tässä aliohjelma esitellään myös staattiseksi (`static`). Staattisen aliohjelman toteutuksessa ei voi käytää `this`-viitettä, sillä se ei ole minkään olion oma. Hyötynä on kuitenkin se, että silloin aliohjelmaa voidaan kutsua mistä tahansa ohjelman osasta ja se ei ole riippuvainen esimerkiksi tässä tapauksessa meidän pelistämme, vaan jonkin muunkin pelin tekijä voisi kutsua aliohjelmaa. Jos emme määrittelisi aliohjelmaa staattiseksi, olisi se metodi eli olion toiminto (ks. luku 8.5).

> Staattisen aliohjelman pitää pystyä tekemään kaikki toimensa pelkästään parametreina tuodun tiedon perusteella.

Tosin staattinen aliohjelma voi käyttää myös staattisia (globaaleja) muuttujia ja vakioita. Staattisten muuttujien käyttö ei ole suositeltavaa. Vakioita voi toki käyttää.

Aliohjelmalle on annettu palautusarvoksi `void`, mikä tarkoittaa sitä, että aliohjelma ei palauta mitään arvoa. Aliohjelma voisi nimittäin myös lopettaessaan palauttaa jonkun arvon, jota tarvitsisimme ohjelmassamme. Tällaisista aliohjelmista, eli funktioista, puhutaan luvussa 9. `void`-määrityksen jälkeen aliohjelmalle on annettu nimeksi `PiirraLumiukko`.

> Huomaa! C#:ssa aliohjelman nimet 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*. Parametrina 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`. Tietotyyppi `double` kuvastaa reaalilukua, eli arvo voi olla esimerkiksi `10.7`, toki tarkkakin arvo `200` kelpaa myös.

Vielä kertauksena esittelyrivin sanat:

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

Sana	Selitys
`public`	aliohjelma on julkinen ja sitä voi kutsua kuka tahansa
`static`	aliohjelma tarvitsee vain parametrina tuotuja tietoja
`void`	aliohjelma ei palauta mitään arvoa
`PiirraLumiukko`	aliohjelmalle itse keksitty nimi
`Game`	tietotyyppi pelille
`peli`	itse keksitty nimi 1. parametrille
`double`	tietotyyppi x-koordinaatille
`x`	itse keksitty nimi x-koordinaatille (2. parametri), voisi olla muukin
`double`	tietotyyppi y-koordinaatille
`y`	itse keksitty nimi y-koordinaatille (3. parametri)

Koska päätimme kutsua aliohjelmaa 3:lla todellisella parametrilla tyyliin:

```
PiirraLumiukko(this, 200, Level.Bottom + 300);
```

on esittelyrivillä oltava kolme muodollista parametria samassa järjestyksessä ja esiteltynä vastaavan tyyppisinä muuttujina. Toki `200` on kokonaisluku, mutta kokonaisluku voidaan sijoittaa reaalilukuun ja siksi yleiskäyttöisyyden vuoksi tässä tapauksessa `x` ja `y` on esitelty reaalilukuina. Tämä ansiosta aliohjelmaa voitaisiin kutsua myös:

```
PiirraLumiukko(this, 10.3, 200.723);
```

Tietotyypeistä voit lukea lisää kohdasta 7.2 ja luvusta 8.

Parametrit erotellaan toisistaan pilkulla sekä kutsussa (todelliset parametrit), että esittelyrivillä (muodolliset parametrit).

Huomaa! Aliohjelman muodollisten parametrien nimien ei tarvitse olla samoja kuin kutsussa. Nimien kannattaa kuitenkin olla mahdollisimman kuvaavia.

Huomaa! Parametrien tyyppien ei tarvitse olla keskenään samoja, kunhan kukin parametri on sijoitusyhteensopiva kutsussa olevan vastinparametrin kanssa. Esimerkkejä funktioista löydät dokumentista Aliohjelminen kirjoittaminen:
https://tim.jyu.fi/view/kurssit/tie/ohj1/materiaali/aliohjelmienKirjoittaminen.

Itse asiassa edellä kutsussa oleva `this` on tyyppiä `Lumiukot` joka on peritty luokasta `PhysicsGame`, mutta koska `PhysicsGame` periytyy tavallisesta pelistä `Game`, voidaan sekä `Lumiukot` että `PhysicsGame`-tyyppinen muuttuja sijoittaa `Game`-tyyppiselle muuttujalle. Aliohjelman esittelyrivillä voitaisiin toki esitellä `peli` myös `Lumiukot` tai `PhysicsGame`-tyyppiseksi, mutta tällöin aliohjelmalla ei voitaisi piirtää lumiukkoa `Game`-tyyppiseen (`Game`-luokasta perittyyn) peliin. Eli tässä on kyseessä vähän samanlainen yleistys kuin se että `200` (kokonaisluku) voidaan sijoittaa reaaliluku- tyyppiseen muuttujaan (`double`).

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

```
aliohjelman peli = this;
aliohjelman x = 200;
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" (tynkä). 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. Tämä siksi, että oikeastaan alkuperäisessä lumiukon piirtämisessä meidän olisi pitänyt kirjoittaa aina:

```
this.Add(p1);
this.Add(p2);
jne.
```

Alkuperäisessä lumiukossa kirjoitimme `Lumiukko` luokan omaa metodia `Begin` ja siinä halusimme sanoa, että pallot lisätään nimenomaan tähän (`this`) peliin (peliolioon, joka on `Lumiukko` luokan ilmentymä). Useissa oliokielissä viitattaessa olion omiin metodeihin (tässä `Add`) tai attribuutteihin, voidaan `this` jättää kirjoittamatta, tai sen saa kirjoittaa. Tässä jokainen voi valita oman tyylinsä, mutta tässä monisteessa `this` jätetään usein kirjoittamatta. Nyt vastaavasti `PiirraLumiukko` aliohjelma ei ole minkään olion oma aliohjelma (`static` aiheuttaa tämän), ja

siksi sille täytyy viedä parametrina tieto siitä, mihin peliin haluamme lumiukon piirtää. Meidän esimerkissämme veimme parametrina nimenomaan tuon `this` -arvon. Siksi meidän esimerkissämme aliohjelmaa suoritettaessa

```
peli.Add(p1);
```

on juurikin

```
this.Add(p1);
```

Lopuksi `Begin`-metodi ja `PiirraLumiukko` -aliohjelma kokonaisena:

```
1     /// <summary>
2     /// Kutsutaan PiirraLumiukko-aliohjelmaa
3     /// sopivilla parametreilla.
4     /// </summary>
5     public override void Begin()
6     {
7         Camera.ZoomToLevel();
8         Level.Background.Color = Color.Black;
9
10        PiirraLumiukko(this, 0, Level.Bottom + 200);
11        PiirraLumiukko(this, 200, Level.Bottom + 300);
12    }
13
14    /// <summary>
15    /// Aliohjelma piirtää lumiukon
16    /// annettuun paikkaan.
17    /// </summary>
18    /// <param name="peli">Peli, johon ukko lisätään.</param>
19    /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
20    /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
21    public static void PiirraLumiukko(Game peli, double x, double y)
22    {
23        PhysicsObject p1, p2, p3;
24        p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
25        p1.X = x;
26        p1.Y = y;
27        peli.Add(p1);
28
29        p2 = new PhysicsObject(2 * 50, 2 * 50, Shape.Circle);
30        p2.X = x;
31        p2.Y = p1.Y + 100 + 50;
32        peli.Add(p2);
33
34        p3 = new PhysicsObject(2 * 30, 2 * 30, Shape.Circle);
35        p3.X = x;
36        p3.Y = p2.Y + 50 + 30;
37        peli.Add(p3);
38    }
```

C#-kielessä, kuten ei monessa muussakaan kielessä, ole väliä sillä onko ensin kirjoitettu pääohjelma (tässä tapauksessa`Begin`) vaiko ensin aliohjelma (tässä tapauksessa `PiirraLumiukko`). Oleellista on että ne muodostavat kokonaisuuksia (eli aaltosulkeisiin `{}` suljetut lohkot).

Aliohjelmia ei suoriteta siinä järjestyksessä kuin ne esiintyvät koodissa vaan siinä järjestyksessä missä niitä kutsutaan. Ohjelman suoritus aloitetaan aina `Main`-aliohjelmasta ja Jypeli-tapauksessa sieltä kutsutaan `Begin`-metodia josta voidaan kutsua muita aliohjelmia, joista voidaan taas kutsua muita aliohjelmia. Kun aliohjelma on valmis, palataan siihen kohtaan, mistä aliohjelmaa kutsuttiin.

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 aikaisemmassa 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.

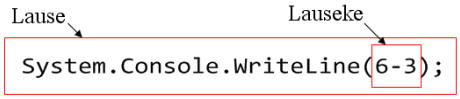
```
PiirraLumiukko(this, 0, Level.Bottom + 200);
```

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`. 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 yksinkertaisia: muuttujan nimi (`this`), kokonaislukuarvo (`0`) ja reaalilukuarvo (`Level.Bottom + 200`. Jos näyttö olisi vaikkapa 800 pikseliä korkea, olisi origo, eli piste `(0,0)` näytön keskellä ja silloin`Level.Bottom` olisi `-400` ja lausekkeen arvo olisi siis `-400 + 200`, eli `-200`). 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.

6.2.1 Valmis kokonaisuus

Kokonaisuudessaan ohjelma näyttää nyt seuraavalta:

```
1  using Jypeli;
2
3
4  /// @author  Antti-Jussi Lakanen, Vesa Lappalainen
5  /// @version 22.8.2012
6  ///
7  /// <summary>
8  /// Piirretään lumiukkoja ja harjoitellaan aliohjelman käyttöä.
9  /// </summary>
10 public class Lumiukot : PhysicsGame
11 {
12     /// <summary>
13     /// Kutsutaan PiirraLumiukko-aliohjelmaa
14     /// sopivilla parametreilla.
15     /// </summary>
16     public override void Begin()
17     {
18         Camera.ZoomToLevel();
19         Level.Background.Color = Color.Black;
20
21         PiirraLumiukko(this, 0, Level.Bottom + 200);
22         PiirraLumiukko(this, 200, Level.Bottom + 300);
23     }
24
25     /// <summary>
26     /// Aliohjelma piirtää lumiukon
27     /// annettuun paikkaan.
28     /// </summary>
29     /// <param name="peli">Peli, johon ukko lisätään.</param>
30     /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
31     /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
32     public static void PiirraLumiukko(Game peli, double x, double y)
33     {
34         PhysicsObject p1, p2, p3;
35         p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
36         p1.X = x;
37         p1.Y = y;
38         peli.Add(p1);
39
40         p2 = new PhysicsObject(2 * 50, 2 * 50, Shape.Circle);
41         p2.X = x;
42         p2.Y = p1.Y + 100 + 50;
43         peli.Add(p2);
44
45         p3 = new PhysicsObject(2 * 30, 2 * 30, Shape.Circle);
46         p3.X = x;
47         p3.Y = p2.Y + 50 + 30;
48         peli.Add(p3);
49     }
50 }
```

Kutsuttaessa aliohjelmaa siirtyy 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 lauseesta. 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.

Tehtävä 6.1 lisää lumiukkoja

Lisää ohjelmaan kaksi muuta lumiukkoa10pt

```
1         PiirraLumiukko(this, 0, Level.Bottom + 200);
2         PiirraLumiukko(this, 200, Level.Bottom + 300);
```

Parametrit voidaan antaa myös nimettyinä, jolloin niiden järjestystä

```
1          PiirraLumiukko(peli:this, y:Level.Bottom + 200, x:0);
2          PiirraLumiukko(this, x:200, y:Level.Bottom + 300);
```

C#:ssa aliohjelmia ja funktioita voidaan kuormittaa (eng. overload) parametrien suhteen. Tämä tarkoittaa, että ohjelmassa voi olla monta samannimistä aliohjelmaa, joilla on eri määrä (tai eri tyyppisiä) parametreja. Lisää luvussa 6.5.

Lisätietoa kuormittamisesta myös videolla Lumiukon kuormitus (12m54s)

Tehtävä 6.2 järjestele toimivaksi

Muokkaa ohjelma toimivaksi. Laita pääohjelma ennen muita aliohjelmia. 10pt

```
1 public class Tulostus
2 {
3    public static void Main()
4    {
5        TulostaLuvut(0, -99);
6    }
7    public static void TulostaLuvut(double p1, double p2)
8    {
9        System.Console.WriteLine(p1 + " " +  p2);
10   }
11 }
```

Lisätietoa aliohjelminen kirjoittamisesta löydät kurssin lisätietosivulta.

6.3 Aliohjelmien dokumentointi

Hyvän ohjelmointitavan mukaan 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 tagien avulla seuraavasti:

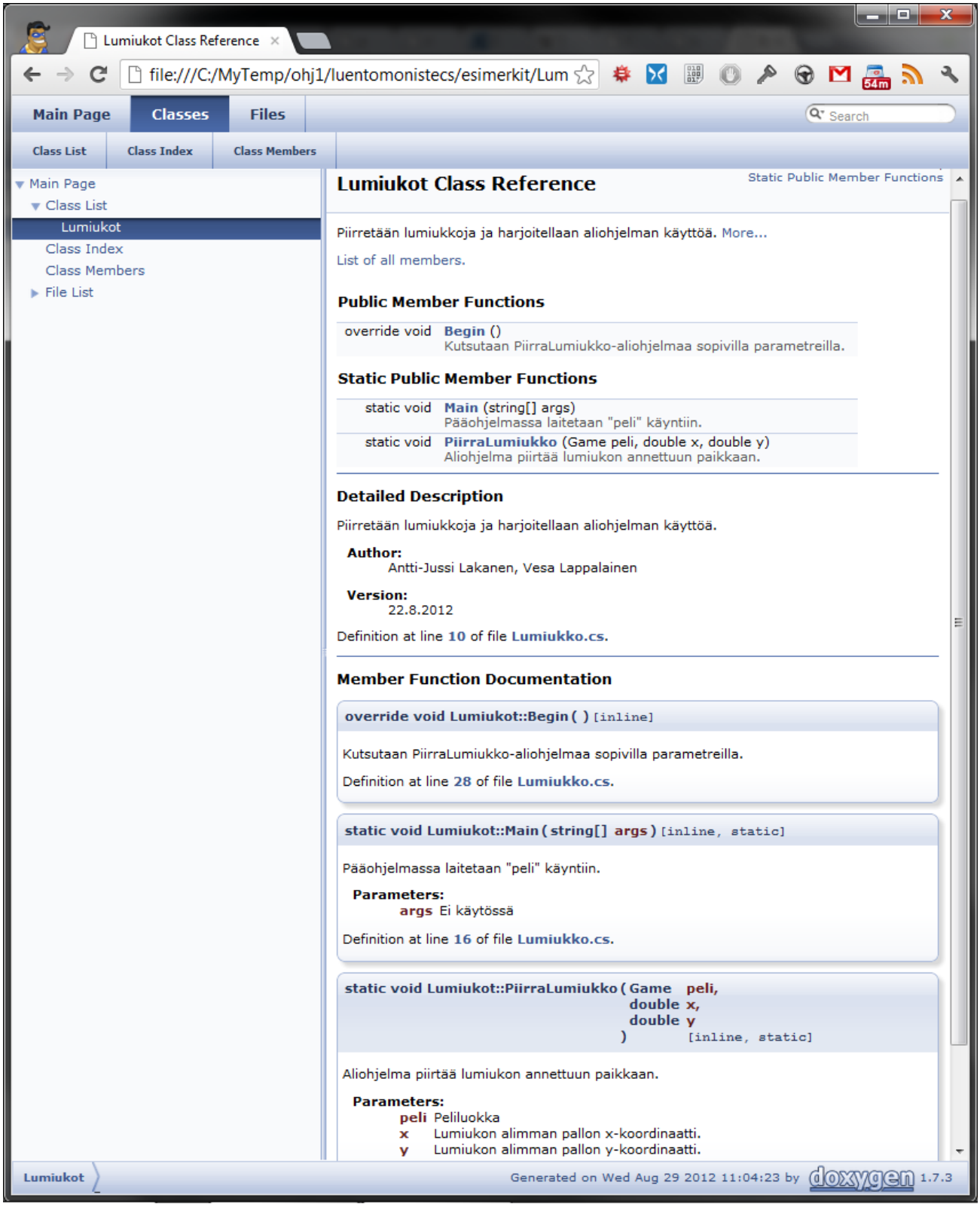
- Dokumentaatiokommentin alkuun laitetaan `summary`-tagien väliin lyhyt ja selkeä kuvaus aliohjelman toiminnasta.
- Jokainen parametri selitetään omien `param`-tagien väliin ja
- paluuarvo `returns`-tagien väliin.

PiirraLumiukko-aliohjelman dokumentaatiokommentit ovat edellisessä esimerkissämme riveillä 36-42.

```
36    /// <summary>
37    /// Aliohjelma piirtää lumiukon
38    /// annettuun paikkaan.
39    /// </summary>
40    /// <param name="peli">Peli, johon ukko lisätään.</param>
41    /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
42    /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
```

Voit kokeilla dokumentaatiota edellisessä täydellisessä Lumiukko-esimerkissä painamalla `Document`-linkkiä. Sitten kokeile syntyvässä dokumentaatiossa eri linkkejä, niin näet mitä niiden takaa löytyy. Alla sama vielä kuvana.

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



Kuva 7: Osa Lumiukot-luokan dokumentaatiosta.

Dokumentaatiossa näkyvät 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 C++:n nimeämistapaa muistuttava tapa. 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. Tällöin aliohjelma esitellään `static` -tyyppiseksi.

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, joilla 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 esimerkkinä C#-kieltä. Päätavoitteena on siis oppia ohjelmoimaan ja työkaluna meillä sen opettelussa on C#-kieli, mutta C#-kielen erityisominaisuuksiin ei kurssilla juurikaan puututa.

Aliohjelmamme `PiirraLumiukko` ei palauttanut mitään arvoa (`void`). 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.

Kerrataan vielä lyhyesti aliohjelman, funktion ja metodin erot.

Aliohjelma: Yleisnimenä mikä tahansa aliohjelma, funktio tai metodi. Aliohjelma ei ota nimenä kantaa parametrien määrään tai paluuarvon tyyppiin. `void`-aliohjelmassa, eli aliohjelmassa joka ei palauta arvoa, voi olla `return`-lause, mutta sen perässä ei silloin ole lauseketta (vrt. `return`-lause funktiossa). Tällöin `return`-lauseen rooliksi jää vain hypätä aliohjelmasta pois.

Joissakin kielissä, esimerkiksi C++:ssa, kaikista aliohjelmista käytetään yleisnimeä funktio. Java-kirjallisuudessa kaikista aliohjelmista käytetään usein yleisnimeä metodi.

Tällä kurssilla käytetään yleisnimeä aliohjelma silloin kun ei erikseen haluta korostaa että kyseessä on erityisesti funktio tai metodi. Tarkennetaan funktion ja metodin käsitteitä seuraavaksi.

Erisnimenä aliohjelma tarkoittaa tällä kurssilla staattista void-tyyppistä aliohjelmaa.

Funktio: Aliohjelma, joka palauttaa arvon, esimerkiksi kahden luvun keskiarvon. Tämän määritelmän mukaan funktiossa on aina vähintään yksi `return`-lause, jonka perässä on lauseke, esimerkiksi `return (a+b)/2.0;`

Tässä määritelmässä ei oteta kantaa parametrien määrään.

Funktion on useimmiten syytä olla `static`. Ihannetilanteessa puhtaalla funktiolla ei ole sivuvaikutuksia, eli se ei esimerkiksi muuta parametrina vietyä taulukkoa.

Metodi: Aliohjelma, joka tarvitsee tehtävän suorittamiseksi kohteena olevan olion omia tietoja. Metodeja käytetään tällä kurssilla (esimerkiksi `merkkijono.IndexOf`), mutta ei tehdä itse muuten kuin peliluokan metodeja (esimerkiksi `Begin`). Joku voi myös mahdollisesti tehdä loppukurssilla uuden luokan, jolle sitten kirjoitetaan omia metodeja. Käytännössä metodissa tarvitaan `this`-viitettä ja se ei saa silloin olla `static`.

Metodi voi myös funktion tapaan palauttaa arvon tai `void`-aliohjelman tapaan olla palauttamatta.

6.4.1 Aliohjelmien kirjoittaminen

Aliohjelman kirjoittamiseksi kannattaa aina edetä seuraavasti (kunhan ensin opitaan testaaminen, TDD, *Test Driven Development*, huomaa että tämä on eri asia kuin debuggaaminen):

1. Jaa ongelma osiin.
2. Mieti millaisella aliohjelmakutsulla pistät tietyn osaongelman ratkaisun käyntiin.
3. Kirjoita aliohjelman kutsurivi ja mieti sen tarvitsemat parametrit.
4. Kirjoita (aluksi manuaalisesti, myöhemmin generoi automaattisesti) aliohjelman esittelyrivi (otsikkorivi, eng. *header*).
   - mieti tarve `public`, `static` - sanoille
   - aliohjelman paluutyyppi `void` vai jotakin muuta?
   - aliohjelman nimi
   - parametrin lukumäärä sama kuin kutsussa
   - parametrien tyyppi sijoitusyhteensopivaksi kutsun kanssa.
5. Tee aliohjelmasta syntaktisesti oikea tynkä joka kääntyy, esimerkiksi funktioaliohjelmassa pitää olla return-lause joka palauttaa lausekkeen (vaikka yksi luku) joka on samaa tyyppiä (tai muuntuu automaattisesti samaksi) kuin funktion tyyppi.
6. Dokumentoi aliohjelma (nyt unohda mistä sitä kutsuttiin, sitä et enää saa ajatella).
7. Kirjoita testit (TDD).
8. Aja testit (pitää "feilata" = NÄE PUNAISTA).
9. Muuta aliohjelma toimivaksi
10. Aja testit (toista kohdat 8-10 kunnes toimii, = NÄE VIHREÄÄ)
11. Siirry seuraavaan aliohjelmaan.

Lue lisää dokumentista Aliohjelmien kirjoittaminen.

- Aliohjelmien kirjoittaminen ndash; 36m45s (44m41s)

Edellä on kirjoitettu yleinen "resepti" aliohjelminen kirjoittamiseksi. Siinä puhutaan testeistä, mutta tämän kurssin tiedoilla voidaan testejä tehdä vain funktioille, joista puhutaan tässä dokumentissa myöhemmin luvussa Aliohjelman paluuarvo. Pelkästään tulostavia ohjelmia ei osata testata tämän kurssin tiedoilla. Eli em. "reseptiä" voidaan kunnolla tällä kurssilla soveltaa vasta funktioiden opiskelun jälkeen.

Ohjelmassa on pääohjelma valmiina ja aliohjelma alustettuna. Aliohjelma ei kuitenkaan vielä tee mitään. Laita se tulostamaan "Hello World" 10pt

```
1 public class HelloWorld
2 {
3     public static void Main()
4     {
5
6         TulostaHelloWorld();
```

```
7
8       }
9
10      ///<summary>
11      ///Tulostaa "Hello World!"
12      ///</summary>
13      public static void TulostaHelloWorld() {
14
15        // tänne tekstiä
16      }
17  }
```

Tehtävä 6.2

Valitse 'Näytä koko koodi' nähdäksesi ohjelman valmiit aliohjelmat. Miten tulostaisit "Hello World!" käyttäen vain parametrittomia aliohjelmakutsuja? Ensimmäinen aliohjelmakutsu on valmiina. 10pt

```
1 //
2         TulostaHe();
```

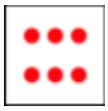
Toki edellisen tehtävän kaltaiset aliohjelmat eivät ole järkeviä, vaan järkevämpää olisi viedä aliohjelmille parametrina että mitä pitää tulostaa.

Tehtävä 6.3

Täydennä alla oleva ohjelma Noppa.cs toimimaan kuten kommenteissa on sanottu. 10pt

```
1  using System;
2  using Jypeli;
3
4  /// <summary>
5  /// Ohjelma piirtää kuusi palloa neliön sisälle siten että
6  /// niistä muodostuu nopan näköinen olio, jonka silmälukuna on 6.
7  /// </summary>
8  public class Peli : PhysicsGame
9  {
10     /// <summary>
11     /// Ruudulla näkyvä sisältö.
12     /// </summary>
13     public override void Begin()
14     {
15        Camera.ZoomToLevel();
16        Level.Background.Color = Color.Black;
17        double koko = 500;
18        GameObject nelio = new GameObject(koko, koko, Shape.Rectangle);
19        Add(nelio);
20        PiirraPallo(this, 0, 100);
21        PiirraPallo(this, 120, 100);
22        // Täydennä ...
23     }
24
25    /// <summary>
26    /// Piirtää pallon, jonka sivun pituus on 80.
27    /// </summary>
28    /// <param name="peli">Peli, johon neliö piirretään</param>
29    /// <param name="x">Pallon keskipisteen x-koordinaatti.</param>
```

```
30    /// <param name="y">Pallon keskipisteen y-koordinaatti.</param>
31    public static void PiirraPallo(Game peli, double x, double y)
32    {
33        GameObject p1 = new GameObject(80, 80, Shape.Circle);
34        //Täydennä
35        peli.Add(p1,1);
36    }
37 }
```



Tuloksen pitäisi näyttää jokseenkin tältä

6.4.2 Tehtävä: Termistöä

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

  PiirraLumiukko(this, 0, Level.Bottom + 200);
  PiirraLumiukko(this, 200, Level.Bottom + 300);
}
```

Tehtävä: Terminologiaa

Mitkä seuraavista väitteistä pitää paikkaansa koskien ylläolevaa ohjelmaa?

	True	False
Sisältää kaksi aliohjelmakutsua	☐	☐
Sisältää kaksi metodikutsua	☐	☐
PiirraLumiukko -aliohjelmalle viedään kolme parametria	☐	☐
Background on Level-olion attribuutti	☐	☐

6.5 Aliohjelman kuormittaminen

C#:ssa aliohjelmia ja funktioita voidaan kuormittaa (eli antaa lisää “kuormaa” samalle nimelle, engl. *overload*) parametrien suhteen. Tämä tarkoittaa sitä, että ohjelmassa voi olla monta samannimistä aliohjelmaa, joilla on eri määrä parametreja tai parametrit ovat eri tyyppisiä. Tätä

voidaan hyödyntää siten, että se funktio joka ottaa enemmän parametreja, osaa tehdä enemmän tai tarkemmin asioita kuin vähemmän parametreja ottava funktio.

6.5.1 Yksinkertaisin esimerkki

Otetaan aluksi mahdollisimman yksinkertainen esimerkki kuormittamisesta. Käytetään tapauksena funktioita, jotka osaavat lisätä lukuja toisiinsa.

Tehdään aluksi funktio, joka palauttaa kahden luvun summan.

```
public static double Summa(double a, double b)
{
    return a + b;
}
```

Tätä voitaisiin kutsua esimerkiksi `Main`-pääohjelmasta kirjoittamalla

```
    double summa = Summa(5, 10.5);
```

Sitten keksimme, että hei, tarvitsemme myös funktion, joka osaa summata kolme lukua, ja haluaisimme kutsua sitä kirjoittamalla

```
    double summa = Summa(5, 10.5, 30.9);
```

Kirjoitetaan *samanniminen* funktio, mutta annetaan sille funktion määrittelyrivillä (esittelyrivillä, otsikkorivillä, eng. *header row* tai *function signature*) kolme parametria kahden sijaan. Toteutetaan funktio myös saman tien.

```
public static double Summa(double a, double b, double c)
{
    return a + b + c;
}
```

Mutta nyt huomaamme, että meillä on melkein sama koodi näissä kahdessa funktiossa. Muutetaan ensimmäistä funktiota siten, että *kutsutaan* ensimmäisestä funktiosta (joka osaa vähemmän) toista funktiota (joka osaa enemmän). Annetaan kolmanneksi summattavaksi luvuksi (siis kolmanneksi parametriksi) 0.

```
public static double Summa(double a, double b)
{
    return Summa(a, b, 0);
}
```

Edelliset koottuna ilman kommentteja. Tulostuksessa on käytetty

```
1  public class KuormitusEsimerkki1
2  {
3      public static void Main()
4      {
5          double summa3 = Summa(5, 10.5, 30.9);
6          double summa2 = Summa(5, 10.5);
7          System.Console.WriteLine($"summa3 = {summa3}, summa2 = {summa2}");
8      }
9
10
11     public static double Summa(double a, double b, double c)
```

```
12      {
13          return a + b + c;
14      }
15
16
17      public static double Summa(double a, double b)
18      {
19          return Summa(a, b, 0);
20      }
21  }
```

Sama käyttäen C#:in oletusparametreja. Oletusparametrin idea on, että

```
1  public class KuormitusEsimerkki2
2  {
3      public static void Main()
4      {
5          double summa3 = Summa(5, 10.5, 30.9);
6          // kääntäjä tekee seuraavasta kutsun Summa(5, 10.5, 0.0)
7          double summa2 = Summa(5, 10.5);
8          System.Console.WriteLine($"summa3 = {summa3}, summa2 = {summa2}");
9      }
10
11
12     public static double Summa(double a, double b, double c=0.0)
13     {
14         return a + b + c;
15     }
16 }
```

Tämän esimerkin avulla näimme yksinkertaisella tavalla sen, mitä kuormittaminen tarkoittaa. Seuraava esimerkki valottaa kuormittamisen hyötyjä paremmin.

6.5.2 Vakiokokoinen lumiukko vs ukon koko parametrina

Voimme luoda vakiokokoisen lumiukon seuraavalla aliohjelmalla.

```
/// <summary>
/// Aliohjelma piirtää vakiokokoisen 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)
{
  PhysicsObject alapallo, keskipallo, ylapallo;
  alapallo = new PhysicsObject(2 * 100.0, 2 * 100.0, Shape.Circle);
  alapallo.X = x;
  alapallo.Y = y;
  peli.Add(alapallo);

  keskipallo = new PhysicsObject(2 * 50.0, 2 * 50.0, Shape.Circle);
  keskipallo.X = x;
```

```
        keskipallo.Y = alapallo.Y + 100 + 50;
        peli.Add(keskipallo);

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

Voimme kutsua tätä aliohjelmaa `Begin`:stä vaikkapa seuraavasti.

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

Mutta entäs jos haluaisimmekin piirtää tämän lisäksi joskus *eri kokoisiakin* ukkoja? Toisin sanoen voisi olla tarve, että `PiirraLumiukko` tekisi meille "vakiokokoisen" ukkojen lisäksi myös halutessamme jonkun muun kokoisen ukkelin. Kutsut `Begin`:ssä voisivat näyttää tältä.

```
        // Vakiokokoisen ukon kutsuminen (alapallon koko 2 * 100)
        PiirraLumiukko(this, -200, Level.Bottom + 300.0);

        // Samannimisen aliohjelman käyttäminen
        // pienemmän ukon tekemiseen (alapallon koko 2 * 50)
        PiirraLumiukko(this, 0, Level.Bottom + 200.0, 50.0);
```

Mutta nyt kääntäjä antaa esimerkiksi virheilmoituksen

```
No overload for method 'PiirraLumiukko' takes 4 arguments.
```

Joten kirjoitetaan uusi aliohjelma, jonka nimeksi tulee `PiirraLumiukko` (kyllä, samanniminen), mutta `peli`-parametrin ja paikan lisäksi parametrina annetaan myös *alapallon säde*.

```
    public static void PiirraLumiukko(Game peli, double x, double y, double sade)
    {
     // tähän kirjoitetaan kohta koodia...
    }
```

Siirretään nyt koodi alkuperäisestä aliohjelmasta tähän uuteen, ja laitetaan pallojen säde riippumaan parametrina annetusta säteestä. Lisäksi laitetaan keski- ja yläpallon paikat riippumaan pallojen koosta! Uusi (neljäparametrinen) aliohjelma näyttäisi nyt seuraavalta.

```
    public static void PiirraLumiukko(Game peli, double x, double y, double sade)
    {
        PhysicsObject alapallo, keskipallo, ylapallo;
        alapallo = new PhysicsObject(2 * sade, 2 * sade, Shape.Circle);
        alapallo.X = x;
        alapallo.Y = y;
        peli.Add(alapallo);

        // keskipallon koko on 0.5 * sade
        keskipallo = new PhysicsObject(2 * 0.5 * sade, 2 * 0.5 * sade, Shape.Circle);
        keskipallo.X = x;
        keskipallo.Y = alapallo.Y + alapallo.Height / 2 + keskipallo.Height / 2;
        peli.Add(keskipallo);
```

```
        // ylapallon koko on 0.3 * sade
        ylapallo = new PhysicsObject(2 * 0.3 * sade, 2 * 0.3 * sade, Shape.Circle);
        ylapallo.X = x;
        ylapallo.Y = keskipallo.Y + keskipallo.Height / 2 + ylapallo.Height / 2;
        peli.Add(ylapallo);
    }
```

Nyt voimme kutsua kolmeparametrisesta `PiirraLumiukko`-aliohjelmasta tuota "versiota", joka osaa tehdä asioita *enemmän* ilman, että copy-pastetamme koodia.

```
public static void PiirraLumiukko(Game peli, double x, double y)
{
    PiirraLumiukko(peli, x, y, 100);
}
```

(IMG: /images/kurssit/tie/ohj1/moniste/Ohjelmointi-1/1028339/ukot.png)

Koko esimerkki kuormiteutusta lumiukosta10pt

```
using System;
using System.Collections.Generic;
using Jypeli;
using Jypeli.Assets;
using Jypeli.Controls;
using Jypeli.Effects;
using Jypeli.Widgets;

/// @author Antti-Jussi Lakanen
/// @version 30.1.2014
///
/// <summary>
/// Aliohjelmien kuormittaminen
/// </summary>
public class Kuormittaminen : PhysicsGame
{
    /// <summary>
    /// Kutsutaan PiirraLumiukko-aliohjelmaa kahdella eri tavalla.
    /// Ensimmäisessä ei anneta parametrina kokoa, jolloin tulee "vakiokokoinen" lumiukko.
    /// Toisessa tavassa annetaan parametrina haluttu koko.
    /// </summary>
    public override void Begin()
    {
        Level.Background.Color = Color.Black;
        PiirraLumiukko(this, -200, Level.Bottom + 300.0);
        PiirraLumiukko(this, 0, Level.Bottom + 200.0, 50.0);

        PhoneBackButton.Listen(ConfirmExit, "Lopeta peli");
        Keyboard.Listen(Key.Escape, ButtonState.Pressed, ConfirmExit, "Lopeta peli");
    }

    /// <summary>
    /// Aliohjelma piirtää vakiokokoisen lumiukon
    /// annettuun paikkaan.
    /// </summary>
    /// <param name="peli">Peli, johon lumiukko tehdään.</param>
    /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
```

```
38      /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
39      public static void PiirraLumiukko(Game peli, double x, double y)
40      {
41          PiirraLumiukko(peli, x, y, 100);
42      }
43
44      /// <summary>
45      /// Aliohjelma piirtää annetun kokoisen lumiukon
46      /// annettuun paikkaan
47      /// </summary>
48      /// <param name="peli">Peli, johon lumiukko tehdään.</param>
49      /// <param name="x">Lumiukon alimman pallon x-koordinaatti.</param>
50      /// <param name="y">Lumiukon alimman pallon y-koordinaatti.</param>
51      /// <param name="sade"></param>
52      public static void PiirraLumiukko(Game peli, double x, double y, double sade)
53      {
54          PhysicsObject alapallo, keskipallo, ylapallo;
55          alapallo = new PhysicsObject(2 * sade, 2 * sade, Shape.Circle);
56          alapallo.X = x;
57          alapallo.Y = y;
58          peli.Add(alapallo);
59
60          keskipallo = new PhysicsObject(2 * 0.5 * sade, 2 * 0.5 * sade, Shape.↩
        Circle);
61          keskipallo.X = x;
62          keskipallo.Y = alapallo.Y + alapallo.Height / 2 + keskipallo.Height / 2;
63          peli.Add(keskipallo);
64
65          ylapallo = new PhysicsObject(2 * 0.3 * sade, 2 * 0.3 * sade, Shape.Circle↩
        );
66          ylapallo.X = x;
67          ylapallo.Y = keskipallo.Y + keskipallo.Height / 2 + ylapallo.Height / 2;
68          peli.Add(ylapallo);
69      }
70  }
```

.

Luku 7

Muuttujat

muuttujanTyyppi muuttujanNimi = muuttujanArvo;
muuttujanNimi = muuttujanUusiArvo;
Esim:
int maailmanTarkoitus = -42;
maailmanTarkoitus = -1 * maailmanTarkoitus;

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. Proseduaalisissa kielissä ilman muuttujia järkevä tiedon käsittely olisi oikeastaan mahdotonta. Funktio-ohjelmoinnissa tosin asiat ovat hieman toisin. Olemme jo ohimennen käyttäneetkin muuttujia, esimerkiksi Lumiukko-esimerkissä teimme `PhysicsObject`-tyyppisiä muuttujia `p1`, `p2` ja `p3`. Vastaavasti `PiirraLumiukko`-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 tarkoittaa hieman eri asioita. Tulet huomaamaan tämän seuraavien kappaleiden aikana.

Muuttuja arvo muuttuu vain sijoituslauseen suoritushetkellä:

```
int ika = 21;
int nyt = 2021;
int syntymavuosi = nyt - ika;  // arvoksi tulee 2000
nyt = 2022; // syntymävuosi on edelleen 2000
```

Muuttujan arvo ei muutu vaikka sen arvon tuottavissa lausekkeissa jokin myöhemmin muuttuisi. Esimerkiksi edellä `syntymavuosi` on edelleen `2000` vaikka jatkossa tehtäisiin sijoitus.

```
nyt = 2022;
```

Eli lausekkeen arvo lasketaan sillä hetkellä kun sijoitus tehdään.

Muuttujaan sijoitetaan sijoitusoperaattorilla `=`. Sijoituksessa muuttujan nimi on vasemmalla ja sijoitettava lauseke sijoitusmerkin oikealla puolella. Myös vakioarvo on lauseke.

```
muuttujannimi = lauseke;
```

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]

Koska kääntäjän pitää osata varata muuttujalle oikean kokoinen muistialue, pitää muuttujalle esitellä myös tyyppi. Muuttujan tyyppiä tarvitaan myös siksi, että tiedetään miten muistipaikkaan tallennettua tietoa pitää käsitellä. Jotta ymmärtäisimme erilaisien tietotyyppien erilaisia tallennustapoja, tutustumme myöhemmin muun muassa binäärilukuihin. Esimerkiksi kahdeksan bitin yhdistelmä, eli tavu `01000001` voidaan tulkita esimerkiksi kirjaimeksi A tai etumerkittömäksi kokonaisluvuksi 65.

Esimerkiksi lauseessa `Console.WriteLine(a)` ei voitaisi tietää mitä pitää tulostaa, mikäli ei tiedetä muuttujan `a` tyyppiä. Aivan vastaavasti kuin lauseesta `kuusi palaa`, ei voida tietää mitä sillä tarkoitetaan jos asiayhteys ei ole selvillä.

7.1 Muuttujan määrittely

Kun matemaatikko sanoo, että "n on yhtä suuri 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:

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

Ensimmäinen rivi tarkoittaa väljästi sanottuna, että *"lohkaise pieni pala - johon mahtuu* ***`int`****-kokoinen arvo - säilytystilaa tietokoneen muistista, ja käytä siitä jatkossa nimeä* ***`n`****"*. Toisella rivillä julistetaan, että *"talleta arvo 1 muuttujaan, jonka nimi on* ***`n`****, siten korvaten sen, mitä kyseisessä säilytystilassa mahdollisesti jo on"*.

Merkki `=` on sijoitusoperaattori ja siitä puhutaan enemmän myöhemmässä luvussa.

Mikä sitten on tuo edellisen esimerkin `int`?

Tehtävä: Sijoitus

Aukaise Tauno. Sijoita n-muuttujaan arvo 1 vetämällä sen päälle <–1 'laatikko'. Aja ohjelma. Tulostuksen tulisi olla n=1. Kokeile sitten kasvattaa (vedä +1 n:än päälle) ja vähentää (vedä -1) muuttujan arvoa ja seuraa minkälaista ohjelmakoodia Tauno kirjoittaa.10pt

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". Mutta mitä ikinä laitammekin, niin muuttujassa voi olla vain **yksi** arvo kerrallaan. Kun muuttujaan sijoitetaan uusi arvo, ei edelliseen arvoon pääse enää mitenkään käsiksi.

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 toki myös erikseen (joka on jopa suositeltavampi tapa):

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

Muuttujaan voi asettaa arvon myös jo määrittelyn yhteydessä. Tällöin puhutaan arvon alustamisesta. Huomattakoon että arvo voi tulla myös lausekkeen tuloksena.

```
muuttujanTietotyyppi muuttujanNimi = VAKIO;
muuttujanTietotyyppi muuttujanNimi = lausekeJokaTuottaaArvon;
```

```
//
        bool onkoKalastaja = true;
        char merkki = 't';
        int kalojenLkm = 0;
        double luku1 = 0, luku2 = 2.0, luku3 = 3+2.4;
```

Muuttujalle sijoitettavan arvon (tai lausekkeen arvon) tulee olla tyypiltään sellainen, että se voidaan sijoittaa muuttujaan. Yksinkertaisiin lainausmerkkeihin kirjoitettu kirjain on arvo, joka voidaan sijoittaa `char`-tyyppiseen muuttujaan. Esimerkiksi int-tyyppiseen muuttujaan ei voi sijoittaa reaalilukua:

```
//
        int ika = 2.5; // TÄMÄ KOODI EI KÄÄNNY
```

mutta reaalilukuun voi sijoittaa kokonaisluvun

```
//
        double hinta = 20000;
```

Huomattakoon, että reaalilukujen (mm. `double`) desimaalierottimen ohjelmakoodissa on aina **piste** (`.`) eikä tuhaterottimia käytetä.

Mitkä sallittuja?

Mitkä seurvaavista muuttujien määrittelyistä ovat sallittuja:

	True	False
int 4;	☐	☐
int = 4;	☐	☐
int luku;	☐	☐
double luku 4,5;	☐	☐
int luku = 5.6;	☐	☐
int a	☐	☐
int henkilonIka, double pituus;	☐	☐
int henkilonIka = 5;	☐	☐
double pituus = 150.0;	☐	☐
double pituus = 350;	☐	☐
leveys double = 350;	☐	☐

7.2 Alkeistietotyypit

C#:n tietotyypit voidaan jakaa alkeistietotyyppeihin (*primitive types*, perustyyppi, perustietotyyppi) 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#-tyyppi	Koko bitteinä	Selitys	Arvoalue
bool	1 (8)	kaksiarvoinen tietotyyppi	true tai false
sbyte	8	yksi tavu	-128..127
byte	8	yksi tavu (etumerkitön)	0..255
char	16	yksi merkki	kaikki merkit
short	16	pieni kokonaisluku	-32,768..32,767
ushort	16	pieni kokonaisluku (etumerkitön)	0..65,535
int	32	kokonaisluku	-2,147,483,648 .. 2,147,483,647
uint	32	kokonaisluku (etumerkitön)	0..4,294,967,295
float	32	liukuluku	noin 7 desimaalin tarkkuus, $\pm 1.5 \times 10^{-45}$.. $\pm 3.4 \times 10^{38}$
long	64	iso kokonaisluku	$-2^{63}..2^{63}-1$
ulong	64	iso kokonaisluku (etumerkitön)	0..18,446,744,073,709,615
double	64	liukuluku	noin 15 desimaalin tarkkuus, $\pm 5.0 \times 10^{-324}$.. $\pm 1.7 \times 10^{308}$
decimal	128	tarkempi kuin double	Noin 28 numeron tarkkuus

Tällä kurssilla tärkeimmät alkeistietotyypit ovat: `bool`, `char`, `int` ja `double`. Huomaa että vaikka `bool` on informaatiosisältönä 1 bittiä, vie se muistia kuitenkin yhden tavun, eli 8 bittiä.

Tässä monisteessa suositellaan, että desimaalilukujen talletukseen käytetään aina `double`-tietotyyppiä (jossain tapauksissa jopa `decimal`-tyyppiä), vaikka monessa muussa lähteessä `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 yhdistelmiä, 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 perustietotyypeillä. 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]

Seuraavassa esimerkki mitä tapahtuu, kun lasketaan yhteen kaksi liian isoa kokonaislukua tai muuten lisätään muuttujaa liikaa.

Tehtävä 7.1

Aja ensin ohjelma muuttamatta. Poista sitten toisesta int-muuttujasta yksi 0. Aja. Mitä tapahtui. Laita takaisin 0. Vaihda tyypit niin, että laskut menevät oikein.10pt

```
1         int luku1 = 1000000000;
2         int luku2 = 2000000000;
3         int summa = luku1 + luku2;
4         byte b = 254;
5         b++; b++;
6         sbyte sb = 127;
7         sb++;
```

7.3 Arvon asettaminen muuttujaan

Muuttujaan asetetaan arvo sijoitusoperaattorilla (*assignment operator*) =. Lauseita, joilla asetetaan muuttujille arvoja, sanotaan sijoituslauseiksi (*assignment statement*). On tärkeää huomata, että sijoitus tapahtuu aina oikealta vasemmalle: sijoitettava on yhtäsuuruusmerkin oikealla puolella ja kohde merkin vasemmalla puolella.

Aukaise Tauno. Tee uusi muuttuja, sen nimeksi ika ja arvoksi ikäsi. Tee myös toinen muuttuja, jonka nimeksi tulee opiskeluvuodet ja haluamasi arvo. Katso taunon tekemää koodia. Huomaa kuinka se lisää automaattisesti muuttujan tietotyypin int. 10pt

Tehtävä 7.2

Esittele muuttujat alla olevia sijoituksia varten niin, että ohjelma kääntyy ja toimii. Esim. int b;10pt

```
1
2
3
4
5         x = 20.0;
6         henkilonIka = 23;
7         paino = 80.5;
8         pituus = 183.5;
9         // 80.5 = paino;  // kokeile, tämä ei toimi!
```

Huomaa että reaalilukuvakioissa käytetään **desimaalipistettä**, ei pilkkua.

Tehtävä 7.3

Laita muuttujien tyyppi sijoitusriville.10pt

```
1         x = 20.0;
2         henkilonIka = 23;
3         paino = 80.5;
4         pituus = 183.5;
5         valovuosiKm = 9460730472580;
6         summa = 128;
7         merkki = '7';
```

Muuttuja täytyy olla määritelty tietyn tyyppiseksi ennen kuin siihen voi asettaa arvoa. Muut-

tujaan 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 ovat reaalilukujen osajoukko. Alla sijoitamme arvon 4 muuttujaan nimeltä luku2, ja kolmannella rivillä luku2-muuttujan sisältämän arvon (4) muuttujaan, jonka nimi on luku1.

```
1         double luku1;
2         int luku2 = 4;
3         luku1 = luku2;
```

Toisinpäin tämä ei onnistu: double-tyyppistä arvoa ei voi sijoittaa int-tyyppiseen muuttujaan. Alla oleva koodi ei kääntyisi:

```
1 // TÄMÄ KOODI EI KÄÄNNY!
2         int luku1;
3         double luku2 = 4.0;
4         luku1 = luku2;
```

Jos edellä oleva sijoitus `int <- double` halutaan välttämättä tehdä, niin silloin on käytettävä tyypin muunnosta eli typecastia (kokeile edelliseen, vaihda myös `4.0` tilalle `4.8`). Tosin tyypinmuunnokseen turvautuminen on aina huono ratkaisu.

```
luku1 = (int)luku2;  // pakotetaan luku 2 int-tyyppiseksi. Katkaisu.
```

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 ja `long`-tyyppisten perään `L`.

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

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

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

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

```
1         string omaNimi = "Antti-Jussi";
```

Sijoituslause voi sisältää myös monimutkaisiakin lausekkeita, esimerkiksi aritmeettisia operaatioita:

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

Sijoituslause voi sisältää myös muuttujia.

```
        double huoneenPituus = 5.40;
        double huoneenLeveys = huoneenPituus;
        double huoneenAla = huoneenPituus * huoneenLeveys;
```

Eli sijoitettava voi olla mikä tahansa lauseke, joka tuottaa muuttujalle kelpaavan arvon. Yhdistämällä muuttujia ja operaatoita voi lauseke olla edellisiäkin "monimutkaisempi":

```
        double alku = 30;
        double nopeus = 80;
        double matka = alku + (nopeus-10)*5 + System.Math.Sin(0.5);
```

Huomaa edellä, että vaikka paperilla kaavoja kirjoitettaessa ei tarvita kertomerkkiä, niin ohjelmointikielissä käytetään `*` -merkkiä kertomerkkinä.

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()
    {
        int ika;
        System.Console.WriteLine(ika);
    }
}
```

Virheilmoitus näyttää tältä:

```
Esimerkki.cs(7,34): error CS0165: Use of unassigned local variable 'ika'
```

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

Koita ajaa ohjelma. Se antaa varoituksen: The variable 'ika' is assigned but its value is never used. Tämä ei estä ohjelmaa kääntymästä, kuten virheilmoitukset tekevät. Poistamalla tulostuslauseen edestä kommenttimerkit //, on muuttuja käytössä, eikä virheilmoitusta tule. 10pt

```
public class Esimerkki
{
    public static void Main()
    {
        int ika = 5;
        // System.Console.WriteLine(ika);
    }
}
```

7.3.1 Sijoituksen kohde on aina vasemmalla

Muuttujan jolle sijoitetaan on lauseessa aina vasemmalla puolella. Sijoitusmerkin `=` oikealla puolella on jokin lauseke, jonka arvo lasketaan ennen sijoitusta ja tämä arvo sijoitetaan muuttujalle.

7.3.2 Tehtävä 7.4 a:n arvon sijoitus b:lle

Vastaa aluksi alla olevaan monivalintakysymykseen ja sitten kirjoita tähän miten sijoitat a:n arvon b:hen kirjoitamalla uuden ohjelmarivin (älä siis muuta kahta olemassa olevaa). Tämän jälkeen aja ohjelma ja katso että se tulostaa 310pt

```
1         int a = 3;
2         int b;
```

Tarkista tietosi

Miten edellisten alkuperäisten kahden rivin jälkeen voidaan a:n arvo sijoittaa b:lle?

	True	False
int b = a;	☐	☐
b = a;	☐	☐
a = b;	☐	☐
ei väliä kumminko päin kirjoitetaan	☐	☐

7.3.3 Muuttujan arvo muuttuu vain kun siihen sijoitetaan

Muuttujan arvo muuttuu vain kun siihen sijoitetaan. Alkeismuuttujaan sijoitetaan aina arvo. Jos muuttujaan sijoitetaan toisen muuttujan arvo, niin muuttuja saa sen arvon, mikä toisella muuttujalla on sijoitushetkellä. Kokeile seuraavalla esimerkillä miten sijoituksen jälkeen i:n arvon muuttaminen ei enää vaikuta `summa`-muuttujaan:

Aukaise Tauno. Sijoita i:n arvo summa-muuttujaan. Kasvata i:n arvoa vetämälle sen päälle +1 'laatikko'. Muuttuuko summa-muuttujan arvo kun kasvatat i:tä? 10pt

7.3.3.1 Tehtävä 7.5 i:n kasvatus, mitä ohjelma tulostaa

Älä vielä aja ohjelmaa, vastaa ensin alla olevaan kysymykseen.10pt

```
1         int i = 2;
2         int summa = i;
3         System.Console.Write(summa + " ");
4         i += 1; // tai i++;
5         System.Console.WriteLine(summa);
```

Mikä muuttuu?

Mita ohjelma tulostaa?

	True	False
2 3	☐	☐
3 2	☐	☐
2 2	☐	☐
0 1 2 3	☐	☐

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 voi C#:ssa alkaa numerolla.

C#:n koodauskäytänteiden mukaan muuttujan nimi alkaa pienellä kirjaimella. 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 käyttöä 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 koodaustapoja (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 avainsanat

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

Taulukko 2: C#:n avainsanat eli "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	

```
1 // TÄMÄ OHJELMA EI KÄÄNNY!!!!!!!!
2 public class Esimerkki
3 {
4     public static void Main()
5     {
6         int event;
7         event = 52;
8         System.Console.WriteLine(event);
9     }
10 }
```

Mitkä määrittelyt oikein?

Mitkä seuraavista muuttujien määrittelyistä ovat sekä syntaktisesti että koodaustapojen mukaan oikein

	True	False
int 12pistetta = 7;	☐	☐
int size = 5;	☐	☐
int tassa on pisteet = 7;	☐	☐
int public = 7;	☐	☐
int Pisteet = 7;	☐	☐
bool true = true;	☐	☐
int x = 7;	☐	☐
double ympyranSade = 0;	☐	☐
double decimal = 0.5;	☐	☐
int default = 5;	☐	☐
PhysicsObject object = new PhysicsObject(20, 20);	☐	☐
int tassa,on,pisteet = 7;	☐	☐

7.5 Muuttujien näkyvyys

Muuttujien näkyvyydellä (eng. *scope*) tarkoitetaan sitä, missä tilanteessa muuttuja on käytettävissä. Jos muuttuja "on näkyvissä" (*in scope*), niin voimme koodissamme kyseisessä kohdassa käyttää muuttujaa.

Muuttujaa voi käyttää (lukea ja asettaa arvoja) vain siinä lohkossa, missä se on määritelty. Lohko alkaa aaltosululla { ja päättyy aaltosululla }.

```
{
    int luku = 5;
}
```

Muuttujat ovat olemassa niin kauan kuin lohkosta ei olla poistuttu. Aliohjelmakutsun aikana lohkosta ei ole poistuttu, koska lohkoon palataan kun aliohjelma on suoritettu. Sisempi lohko ei myöskään aiheuta poistumista.

```
{
    int luku = 5;
    AliohjelmaKutsu();
    {
        luku++;
    }
```

```
    }
```

Muuttujan määrittelyn täytyy aina olla ennen (koodissa ylempänä) kuin sitä ensimmäisen kerran käytetään. Saman lohkon sisälläkin muuttuja tulee esitellä ennen sen käyttöä, sillä muuttuja alkaa *näkyä* vasta esittelynsä jälkeen.

```
1         luku++;  // EI TOIMI, muuttujaa ei ole vielä
2         {
3           luku++; // EI TOIMI, muuttujaa ei vielä esitelty
4
5           int luku = 5; // Nyt on esitelty
6
7           luku++; // TOIMII
8           System.Console.WriteLine(luku);
9         }
10        luku++; // EI TOIMI, ei ole vaikutusalueessa
```

Seuraavassa muuttujat `luku` ja `d` ovat nähtävissä ja muutettavissa vain pääohjelmassa (paitsi jos viedään C#:issa out-parametrina). Kaikki pääohjelmassa (tai missä tahansa muussakin aliohjelmassa) esitellyt muuttujat elävät pääohjelman loppusulkuun } saakka. Tässä muuttujan luku arvo kopioidaan aliohjelman vastinmuuttujaan. Aliohjelma ei mitenkään "näe" pääohjelman muuttujaa, vaan aliohjelma saa tiedokseen sille välitetyn arvon.

Aliohjelman sisällä määritelty muuttuja ei näy muissa aliohjelmissa ja sitä kutsutaan *lokaaliksi muuttujaksi*. Muuttujat `luku` ja `d` ovat pääohjelman (`Main`) lokaaleja muuttujia.

```
1 //
2     public static void Main()
3     {
4         int luku = 9;
5         double d = 0.5;
6         Muuta(2, luku);
7         System.Console.WriteLine("luku = {0}, d = {1}",luku,d);
8     }
```

Esimerkissä parametrimuuttujan nimi on sama `luku` kuin pääohjelmassakin (ks. `Näytä koko koodi`), mutta nimi voisi olla mikä tahansa muukin. Oleellista on, että kutsussa aliohjelman vastaavassa paikassa olevaan muuttujaan sijoitetaan sama arvo kuin kutsuvassakin ohjelmassa. Vaikka muuttujaan `luku` sijoitettaisiin jotakin, se ei vaikuta kutsuvaan ohjelmaan, koska `luku` on oma lokaalimuuttuja aliohjelmassa ja on olemassa vain siihen saakka kun kunnes tullaan aliohjelman loppusulkuun `}`.

Edellä on käytetty tulostuksessa versiota, jossa annetaan ensin muotoilujono ja sitten muotoiltavat lausekkeet pilkuilla eroteltuna. Tästä myöhemmin lisää.

```
1 //
2     public static void Muuta(int ika, int luku)
3     {
4         ika--;  // vaikka parametrimuuttujien arvoja voi muuttaa, se ei ole hyvä ↩
      tapa
5         int uusiarvo;
6         uusiarvo = luku +3;
7         luku = 12;  // tämäkään ei ole, hyvä että muuttaa parametrin arvoa
8     }
```

Käännettäessä ohjelma tulee varoitus siitä, että aliohjelman muuttujaa `uusiarvo` ei käytetä enää sen jälkeen kun sille on sijoitettu arvo. Mikäli aliohjelmaa kutsuttaisiin uudelleen, syntyisi uudelle kutsukerralla oma `uusiarvo` -muuttuja, eikä sillä olisi enää mitään tekemistä edellisen kutsukerran vastaavan arvon kanssa.

Edellä apumuuttuja uusiarvo on näkyvissä aliohjelmassa esittelyrivinsä jälkeen, mutta lakkaa olemasta kun tullaan aliohjelman loppusulkuun }. Tähän muuttujaan tehdyt muutokset (vaikka jossakin olisi samanniminenkin muuttuja) eivät millään tavalla vaikuta mihinkään muuhun paikkaan kuin tähän muuttujaan. Pääohjelma tai kukaan muukaan ei pääse käsiksi tähän muuttujaan millään tavalla (paitsi tässä tapauksessa kun tuo arvo riippuu parametrina tuodun lukumuuttujan arvosta).

Parametrimuuttujien muuttamista ei yleisesti pidetä hyvänä tyylinä. Jos parametrimuuttujia pitää muuttaa, parempi on tehdä niistä lokaali kopio ja muuttaa sitä, näin aliohjelman lopussa parametreilla on samat arvon kuin aliohjelmaan tultaessakin.

Tässä on edellä esitetyt aliohjelmat luokan sisällä. Kaikki muuttujat ovat lokaaleja muuttujia.

```
1  public class LokaalitMuuttujat
2  {
3      public static void Main()
4      {
5          int luku = 9;
6          double d = 0.5;
7          Muuta(2, luku);
8          System.Console.WriteLine("luku = {0}, d = {1}",luku,d);
9      }
10
11     public static void Muuta(int ika, int luku)
12     {
13         ika--;
14         int uusiarvo;
15         uusiarvo = luku +3;
16         luku = 12;
17     }
18 }
```

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**.

```
1  public class GlobaalitMuuttujat
2  {
3      public static int pisteet; // erittäin paha tapa!!!
4      public static int tulos;   // erittäin paha tapa!!!
5
6      public static void Main()
7      {
8          tulos = 10;
9          System.Console.WriteLine("pisteet = {0}, tulos = {1}",pisteet,tulos);
10         Muuta();
11         System.Console.WriteLine("pisteet = {0}, tulos = {1}",pisteet,tulos);
12     }
13
14     public static void Muuta()
```

```
15      {
16          tulos += 10;
17          pisteet = 15;
18      }
19  }
```

Edellä olevat muuttujat `pisteet` ja `tulos` ovat globaaleja muuttujia, koska ne esitellään aliohjelmien ulkopuolella. Ne ovat käytössä myös luokan ulkopuolisista luokista, koska ne on **valitettavasti** esitelty myös avainsanalla `public`. Mikäli sana `static` puuttuisi muuttujien esittelystä, ei niitä voisi käyttää staattisista aliohjelmista. Silloin muuttujat olisivat attribuutteja ja niiden käyttämiseksi pitäisi luoda olio, jonka sisälle attribuutit syntyvät. Tämä menee ohi tämän kurssin varsinaisesta sisällöstä.

```
1   /// <summary>
2   /// Tutkitaan muuttujinen näkyvyyttä
3   /// </summary>
4   public class MuuttujienNakyvyys
5   {
6       /// <summary>
7       /// Missä pääohjelman muuttujat näkyvät
8       /// </summary>
9       /// <param name="args">ei käytössä</param>
10      public static void Main(string[] args)  // args näkyy pääohjelmassa
11      {
12          int luku = 9;   // Näkyy vain pääohjelmassa
13          double d = 5.5;  // Näkyy vain pääohjelmassa
14          System.Console.WriteLine("Ennen muutosta: {0}, {1}", luku, d);
15          Muuta(2, luku);
16          {                           // apulohko, jossa omia muuttujia
17              int uusi = 3;           // muuttuja joka näkyy vain tässä lohkossa
18              System.Console.WriteLine("uusi: " + uusi);
19          }                           // nyt uusi-muuttuja lakkaa olemasta
20          // Nyt muuttujaa uusi ei ole olemassakaan
21          System.Console.WriteLine("Muutosten jälkeen: {0}, {1}", luku, d);
22
23      }
24      /// <summary>
25      /// Yritetään muuttaa pääohjelman lokaaleja muuttujia aliohjelmassa
26      /// </summary>
27      /// <param name="uusiArvo">muuttujalle annettava uusi arvo,
28      //      näkyy vain aliohjelmassa, muuttaminen ei vaikuta kutsuvaan ↩
    ohjelmaan</param>
29      /// <param name="luku">muuttuja, jonka arvoa muutetaan,
30      //     näkyy vain aliohjelmassa, sama nimi ei haittaa,
31      //     muuttaminen ei vaikuta kutsuvaan ohjelmaan</param>
32      public static void Muuta(int uusiArvo, int luku)
33      {
34          uusiArvo--;             // ei vaikuta pääohjelmaan
35          int uusiarvo;          // aliohjelman lokaali muuttuja
36          uusiarvo = luku + 3;
37          luku = 12;             // ei vaikuta pääohjelmaan
38      }
39
40  }
```

Kokeile edellä mitä tapahtuu jos kirjoitat aliohjelmaan `Muuta` sijoituksen `d = 4`.

Samaa muuttujan nimeä voidaan käyttää uudelleen eri näkyvyysalueessa. Kussakin näkyvyysalueessa se on kuitenkin eri muuttuja.10pt

```
public class SamaNimi
{
    public static void Main()
    {
        int i = 4;
        System.Console.WriteLine("Pääohjelman i = {0}",i);
        Ali1(i);
        System.Console.WriteLine("Pääohjelman i = {0}",i);
        Ali2();
        System.Console.WriteLine("Pääohjelman i = {0}",i);
    }


    public static void Ali1(int i)
    {
        System.Console.WriteLine("Ali1:n i = {0}",i);
        i++;
        System.Console.WriteLine("Ali1:n i = {0}",i);
    }

    public static void Ali2()
    {
        int i = 8;
        System.Console.WriteLine("Ali2:n i = {0}",i);
        i++;
        System.Console.WriteLine("Ali2:n i = {0}",i);
    }
}
```

C# ei kuitenkaan salli sisäkkäisen lohkon käyttää samaa nimeä, mitä on käytetty ulommassa lohkossa. Kuitenkin jos globaalilla ja lokaalilla muuttujalla on sama nimi, niin lokaali muuttuja näkyy omassa lohkossaan.

Kokeile seuraavassa kommentoida pois rivi `i=9` niin ohjelma kääntyy ja tulostaa pääohjelman lokaalin `i`:n. Jos myös rivin `i=5` kommentoi pois, niin tulostuu globaali `i`.

```
public class SisalohkossaSama
{
    public static int i = 6;

    public static void Main()
    {
        int i = 5; // peittää globaalin
        System.Console.WriteLine("Ulkolohkon i = {0}",i);
        {
            int i = 9; // TÄMÄ EI KÄÄNNY
            System.Console.WriteLine("Sisälohkon i = {0}",i);
        }
    }
}
```

Lisätietoa muuttujien näkyvyydestä löydät kurssin lisätietosivulta.

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.

```
1         const int KUUKAUSIEN_LKM = 12;
2         // KUUKAUSIEN_LKM = 13; // Kokeile poistaa tämä kommenteista
```

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.

Tehtävä 7.6

Ohjelmassa esitellään yhteensä 10 muuttujaa ja yksi vakio. Lisää jokaisen muuttujan/-vakion perään kommentti, jossa ilmoitetaan, onko se muuttuja vai vakio ja sen tyyppi (globaali, lokaali, parametri) 10pt

```
1  public class Esimerkki
2  {
3
4      static int luku1 = 1;
5      static int luku2 = 2;
6
7      public static void Main()
8      {
9          {
10             const int LUKU3  = 3;
11             int luku4 = 4;
12         }
13         //TÄSTÄ
14
15         int luku5 = 5;
16
17         //TÄHÄN
18         {
19             int luku3 = 3;
20             int luku4 = 4;
21         }
22     }
23
24     public static void Aliohjelma(int luku5, int luku6)
25     {
26         int luku7 = luku5;
27         int luku8 = luku6;
28     }
29 }
```

Tarkista tietosi

Edellisessä tehtävässä on kommentit '//TÄSTÄ' ja '//TÄHÄN'. Mitkä muuttujat ovat näkyvissä näiden kommenttien sisällä?

	True	False
luku1	☐	☐
luku2	☐	☐
LUKU3	☐	☐
luku3	☐	☐
luku4	☐	☐
luku5	☐	☐
luku5 ja luku 6	☐	☐
luku7 ja luku 8	☐	☐

Tarkista tietosi

Mitkä seuraavista väitteistä pitää paikkaansa?

	True	False
Globaali muuttuja on esitelty aliohjelmien ulkopuolella.	☐	☐
Globaali muuttuja voidaan esitellä pääohjelmassa.	☐	☐
Globaalin muuttujan arvo voidaan muuttaa aliohjelmassa.	☐	☐
Jos pääohjelmassa esitellään muuttuja, on se lokaali muuttuja.	☐	☐
Jos globaalia muuttujaa käytetään aliohjelmassa, siitä tulee lokaali muuttuja.	☐	☐
Lokaali muuttuja näkyy vain lohkon sisällä.	☐	☐
Lokaaleja muuttujia kannattaa suosia globaalien sijasta	☐	☐
Vakion arvoa ei voi muuttaa, paitsi jos se on lokaali muuttuja.	☐	☐

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	`Console.WriteLine(1+2); // 3`
`-`	vähennyslasku	`Console.WriteLine(1-2); // -1`
`*`	kertolasku	`Console.WriteLine(2*3); // 6`
`/`	jakolasku	`Console.WriteLine(6 / 2); // 3`
		`Console.WriteLine(7 / 2); //Huom! 3`
		`Console.WriteLine(7 / 2.0); // 3.5`
		`Console.WriteLine(7.0 / 2); // 3.5`
`%`	jakojäännös (modulo)	`Console.WriteLine(18 % 7); // 4`

Huom: `18/7 = 2, jää 4`. Kokonaisluvuille tehtävä jakolasku palauttaa tuon 2, kun taas jakojäännös palauttaa `4`. Jakojäännöstä käytetään usein sen testaamiseen, onko luku jaollinen jollakin luvulla, esim:

Animaatio: Suorita aritmeettisia operaatioita

Askella silmukan suoritusta vihreällä nuolella Tutki operaatioiden toimintaa

```
1         int vuosi = 2001;
2         if ( vuosi % 4 != 0 )
3            System.Console.WriteLine("Vuosi ei ole karkausvuosi");
```

Tehtävä 7.7

Kokeile + -merkin tilalle kaikkia em. operaattoreita. Mieti ennen ajoa mitä ohjelma tulostaa ja kirjoita 'arvauksesi' alempana olevaan tehtävään. Ajon jälkeen kirjoita viereen mitä oikeasti tuli.10pt

```
1         int luku1 = 17;
2         int luku2 = 2;
3         int tulos = luku1 + luku2;
```

Kirjoita mitä tulostaa milläkin operaattorilla10pt

```
1 +   tulos = 19
2 -   tulos =
3 *   tulos =
4 /   tulos =
5 %   tulos =
```

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;`
		`Console.WriteLine(luku++);  // tulostaa 0`
		`Console.WriteLine(luku++);  // tulostaa 1`
		`Console.WriteLine(luku);    // tulostaa 2`
		`Console.WriteLine(++luku);  // tulostaa 3`

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

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

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

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

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

`%=`	Jakojäännösoperaatio	`int luku = 9;`
		`luku %= 5; // luku-muuttujan arvo on 4`
		`luku = 9;`
		`luku %= 2; // luku-muuttujan arvo on 1`

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

```
1         int luku1 = 5;
2         int luku2 = 5;
3         System.Console.WriteLine(++luku1); // tulostaa 6;
4         System.Console.WriteLine(luku1++); // tulostaa 6;
5         System.Console.WriteLine(luku2 + 1 ); // tulostaa 6;
6         System.Console.WriteLine(luku1); // 7
7         System.Console.WriteLine(luku2); // 5
8         System.Console.WriteLine(9%2); // 1
9         int luku = 9;
10        luku %= 2;
11        System.Console.WriteLine(luku); // 1
```

7.7.4 Aritmeettisten operaatioiden suoritusjärjestys

Aritmeettisten operaatioiden *presedenssi*, eli missä järjestyksessä operaatiot lasketaan, on vastaava kuin matematiikan laskujärjestys. Kerto- ja jakolaskut (myös jakojäännös) suoritetaan ennen yhteen- ja vähennyslaskua. Laskujärjestystä voi muuttaa suluilla; sulkeiden sisällä olevat lausekkeet suoritetaan ensin.

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

7.8 Huomautuksia

7.8.1 Kokonaisluvun tallentaminen liukulukumuuttujaan

Laskutoimituksen lopputulos riippuu tyypeistä, joita operoidaan. Kahden int-tyyppisen luvun jakolaskun lopputulos on int-tyyppinen. Niinpä esimerkiksi `4/3 == 1` (`4` ja `3` ovat `int`-lukuja), joten vastaus on `int`-tyyppinen. Toisaalta `(4.0 + 3) / 3 == 2.333...`, koska lausekkeen `4.0 + 3` tyyppi on `double`, ja `double` jaettuna `int`-arvolla tuottaa `double`-tyyppisen arvon.

```
1         double laskunTulos = 5 / 2;
2         System.Console.WriteLine(laskunTulos); // tulostaa 2
```

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

```
1         double laskunTulos = 5 / 2.0;
2         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:

```
1         int laskunTulos = 5 / 4;
2         System.Console.WriteLine(laskunTulos); // tulostaa 1
3
4         laskunTulos = 5 / 6;
5         System.Console.WriteLine(laskunTulos); // tulostaa 0
6
7         laskunTulos = 7 / 3;
8         System.Console.WriteLine(laskunTulos); // tulostaa 2
```

Kokonaisluvuilla laskettaessa lukuja ei siis pyöristetä lähimpään kokonaislukuun, vaan desimaaliosa menee C#:n jakolaskuissa ikään kuin "hukkaan". Jos sekä jakaja että jaettava ovat kokonaislukumuuttujissa, niin jakolasku siis katkeaa kokonaisluvuksi. Ongelmaa voi kiertää niin, että aloittaa koko laskutoimituksen reaaliluvulla.

```
1 //
2         int luku1 = 5;
3         int luku2 = 2;
4         double laskunTulos = luku1 / luku2;
5         System.Console.WriteLine(laskunTulos); // tulostaa 2
6         laskunTulos = 1.0 * luku1 / luku2;
7         System.Console.WriteLine(laskunTulos); // tulostaa 2.5
```

Tehtävä 7.7.1 Mitä sulut vaikuttavat

Mitä tapahtuu jos edellä laitetaan luku1/luku2 sulkuihin?10pt

7.8.1.1 Tehtävä 7.8

Alla on ensin esiteltynä kaikki vastausvaihtoehdot. Mieti ensin kysymyksien kohdalla mikä on tulos ja katso vasta sitten oikea vastaus sitten luentovideolta.

Numero	1	2	3	4	5	6	7	8	9
Vastaus	0	1	1.5	2	7	8	9	13	Ohjelma kaatuu

7.8.1.1.1 Mitä seuraavien lausekkeiden tulos on?

- 7 % 7 Vastaus ndash; 52m31s (1m27s)
- 8 - 7 % 7 Vastaus ndash; 54m45s (2m3s)
- 13-5 % 3-2 Vastaus ndash; 57m42s (13s)
- 3 / 2 Vastaus ndash; 58m31s (6m4s)

7.8.1.1.2 Mitä muuttujien arvot ovat?

- int a = 5 + 10 % 6 / 3 + 1; a:n arvo tämän jälkeen? Vastaus ndash; 1h5m50s (49s)

- double d = 5 + 10 % 6 / 3 + 1; d:n arvo tämän jälkeen? Vastaus ndash; 1h7m10s (10s)
- double e = 5.0 + 10 % 6 / 3 + 1; e:n arvo tämän jälkeen? Vastaus ndash; 1h7m56s (1m49s)
- double e = 5.0 + 10.0 % 6 / 3 + 1; e:n arvo tämän jälkeen? Vastaus ndash; 1h10m20s (39s)

7.8.2 Lisäys- ja vähennysoperaattoreista

On neljä tapaa kasvattaa luvun arvoa yhdellä.

```
1         int a = 3;
2         int b,c;
3         b = ++a; // Huom!  Ei olisi pakko sijoittaa mihinkään!
4         ++a;     // eli näin voi kasvattaa
5         c = a++; // idiomi.  c saa a:n alkuperäisen arvon ja sitten a kasvaa.
6         a++;  // tätäkin voi käyttää (ja paljon käytetään) ilman sijoittamista
7         a += 1;
8         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.

```
1         int a = 10;
2         a -= 2;
3         a += -2;  // lisättävä voisi olla lausekekin. Luku voi olla myös ↩
      negatiivinen
4         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.

Animaatio: Suorita operaattoreita

Askella ohjelmaa vihreällä nuolella. Mutta tässä esimerkissä on huonoa tuo että sijoitetaan result = result++; koska niin ei oikeasti koskaan tehdä Tutki operaattoreita.

7.8.3 Varo nollalla jakamista

Yksi yleisiä ohjelmointivirheitä on nollalla jakaminen. Tämä ei ole syntaksivirhe, koska sitä ei useinkaan voida havaita käännösaikana. Eli nollalla jakaminen on looginen, vasta ohjelman ajon aikana ilmenevä virhe. Ohjelmoijan on aina itse pidettävä ennen jakolaskua huolta siitä, että jakaja ei voi olla nolla. Tässä tosin tarvitaan apuna myöhemmin esiteltävää ehtolausetta (`if`):

Kokeile mitä tapahtuu kun ohjelma ajetaan.10pt

```
1 //
2         double tulos;
3         int jakaja = 3;
```

```
4         int jaettava = 7;
5         tulos = jaettava / (jakaja - 3);
```

7.8.4 Numeeristen tietotyyppien arvo-alueet

Numeeristen tietotyypin pienin ja suurin mahdollinen arvo saadaan

```
tietotyyppi.MinValue
tietotyyppi.MaxValue
```

Reaalilukutyypeille on myös

```
tietotyyppi.Epsilon
```

joka kertoo pienimmän positiivisen arvon jonka muuttuja voi saada. Tästä seuraava pienempi arvo on 0.

```
1         byte pieninByte  =  byte.MinValue;
2         byte suurinByte  =  byte.MaxValue;
3         int pieninInt  =  int.MinValue;
4         int suurinInt  =  int.MaxValue;
5         long pieninLong  =  long.MinValue;
6         long suurinLong  =  long.MaxValue;
7         float pieninFloat  =  float.MinValue;
8         float suurinFloat  =  float.MaxValue;
9         float nollaaLahinFloat = float.Epsilon;
10        double pieninDouble =  double.MinValue;
11        double suurinDouble  =  double.MaxValue;
12        double nollaaLahinDouble = double.Epsilon;
13        Console.WriteLine($"Pienin byte on {pieninByte}, suurin on {suurinByte}")↩
      ;
14        Console.WriteLine($"Pienin int on {pieninInt}, suurin on {suurinInt}");
15        Console.WriteLine($"Pienin long on {pieninLong}, suurin on {suurinLong}")↩
      ;
16        Console.WriteLine($"Pienin float on {pieninFloat}, suurin on {suurinFloat↩
      }");
17        Console.WriteLine($"Nollaa lähin {nollaaLahinFloat}");
18        Console.WriteLine($"Pienin double on {pieninDouble}, suurin on {↩
      suurinDouble}");
19        Console.WriteLine($"Nollaa lähin {nollaaLahinDouble}");
```

Näitä voidaan käyttää hyväksi esimerkiksi siten, että kun etsitään vaikkapa kokonaislukutaulukon suurinta lukua, laitetaan ehdokas funktion aluksi pienempään mahdolliseen arvoonsa, jolloin kuka tahansa “voittaa sen:

```
int ehdokas = int.MinValue;
```

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
/// @version 22.8.2012
///
/// <summary>
/// Ohjelma, joka laskee painoindeksin
/// pituuden (m) ja painon (kg) perusteella.
/// </summary>
public class Painoindeksi
{
    /// <summary>
    /// Pääohjelma, jossa painoindeksi tulostetaan ruudulle.
    /// </summary>
    public static void Main()
    {
      double pituus = 1.83;
      double paino = 75.0;
      double painoindeksi = paino / (pituus*pituus);
      System.Console.WriteLine("Painoindeksisi on {0:0.00}",painoindeksi);
   }
}
```

Tehtävä 7.9

Muuta edellinen esimerkki siten, että painoindeksi lasketaan aliohjelmassa, jota kutsutaan pääohjelmasta. Aliohjelma voi myös tulostaa tuloksen, mutta tällöin nimessä tulisi lukea se, esimerkiksi TulostaPainoindeksi. Lisää dokumentaatiokommentit. 10pt

Jypelin luokkaluettelo: http://kurssit.it.jyu.fi/npo/material/latest/documentation/html/classes.html

Tehtävä 7.10

Tutki jypelin luokkaluetteloa. Etsi Level-luokka ja listaa sen attribuutteja eli ominaisuuksia tähän. Kerro myös ominaisuuden paluuarvo. 10pt

Tehtävä 7.11

Aikaisemmin tehtiin aliohjelma, joka tulostaa automaattisesti tekstin ”Hello World”. Tee nyt aliohjelma, joka tulostaa parametrina viedyn tekstin. Lisää myös dokumentaatiokommentit. 10pt

```
using System;

public class Tulostus
{
    public static void Main()
    {
        String teksti = "Jeps Jeps";
        TulostaTeksti();

    }


    public static void TulostaTeksti() {
```

```
14
15
16      }
17 }
```

Luku 8

Oliotietotyypit

C#:n alkeistietotyypit antavat melko rajoittuneet puitteet ohjelmointiin. Niillä pystytään tallentamaan ainoastaan lukuja (int, double, jne.), yksittäisiä merkkejä (char) ja totuusarvoja (bool). Vähänkään monimutkaisemmissa ohjelmissa kuitenkin 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 (engl. *object*) on tietorakenne, jolla pyritään ohjelmoinnissa kuvaamaan reaalimaailman ilmiöitä. Luokkapohjaisissa kielissä (kuten C#, Java ja C++) olion rakenteen ja käyttäytymisen määrittelee luokka, joka kuvaa siitä luodun olion attribuutit ja metodit. Attribuutit ovat olion ominaisuuksia ja metodit olion toimintoja. Olion sanotaan olevan luokan *ilmentymä*. Yhdestä luokasta voi siis (yleensä) luoda useita olioita, joilla on samat ominaisuudet ja toiminnallisuudet. 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 teppo = 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 oliomuuttujan arvo (eli viitteen arvo) on *null*. Oliomuuttujan, joka sisältää null-viitteen, käyttäminen aiheuttaa ajonaikaisen virheen. Oliomuuttujan arvo voidaan myös joissain erikoistilanteissa tarkoituksellisesti asettaa null-arvoksi sanomalla olionNimi = null.

Uusi Tyontekija-olio voitaisiin luoda esimerkiksi seuraavasti. Parametrit riippuisivat nyt siitä, kuinka olemme toteuttaneet Tyontekija-luokan konstruktorin. Tässä tapauksessa annamme nyt parametreina 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* tai *olioviitteeksi*. Viitemuuttujat eroavat oleellisesti alkeistietotyyppisistä muuttujista.

8.3 Arvopohjaiset tietotyypit ja viitepohjaiset tietotyypit

C#:n tyyppijärjestelmä jakaa tietotyypit kahteen kategoriaan: *arvopohjaisiin* tyyppeihin ja *viitepohjaisiin* tyyppeihin.

C#:n sisäänrakennettuja arvopohjaisia tyyppejä ovat muun muassa `int`, `double`, `char` ja `bool`. Täydellisen listan näet C#:n dokumentaatiosta. Viitepohjaisia tyyppejä (tai lyhyesti viitetyyppejä) ovat taulukot, kuten `int[]` sekä merkkijonot, kuten `string` ja `StringBuilder`. Myös kaikki luokista tehdyt oliot, kuten `PhysicsObject`-oliot, ovat viitetyyppejä.

- Arvopohjaiset tyypit sisältävät datan "suoraan". Esimerkiksi lauseen `int a = 3;` seurauksena syntyvä muuttuja `a` sisältää arvon `3`.
- Viitetyypit sisältävät arvon, joka on viite johonkin toiseen paikkaan muistissa. Esimerkiksi lauseen `int[] taulukko = { 1, 2, 3 };` seurauksena syntyvä muuttuja `taulukko` sisältää viitteen toiseen sijaintiin (käytännössä osoite tietokoneen keskusmuistissa), jossa varsinainen sisältö `1, 2, 3` on.

Muuttujien luominen ohjelmassa vaatii muistitilaa tietokoneen keskusmuistista. C# varaa muistista tilaa muuttujan sisältämälle tiedolle (yllä olevassa esimerkissä `3` ja `{ 1, 2, 3 }`) jommasta kummasta kahdesta muistialueesta: pino tai keko. Tällä kurssilla pääsääntö on seuraava: arvopohjaisten tietotyyppien data sijaitsee pinossa ja viitetyyppien data sijaitsee keossa.

Tarkasti ottaen arvopohjaisten muuttujien arvot voivat sijaita joko pinossa tai keossa riippuen siitä, missä kontekstissa muuttuja on määritelty. Esimerkiksi Henkilö-luokka (viitepohjainen, sijaitsee keossa) voisi sisältää int-tyyppisen ikä-attribuutin. Tässä tilanteessa myös ikä sijaitsisi keossa, ei pinossa.

Yleensä meidän ei tarvitse olla kovin huolissamme siitä, käytämmekö arvopohjaista tietotyyppiä vai viitetyyppiä (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.

```
1         int luku1 = 10;
2         int luku2 = luku1;
3         luku1 = 0;
4         System.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:

```
1         PhysicsObject p1 = new PhysicsObject(2*100.0, 2*100.0, Shape.Circle);
2         Add(p1);
3         p1.X = -200;
4
5         PhysicsObject p2 = p1;
6         p2.X = 100;
```

Yllä oleva koodi piirtää seuraavan kuvan:

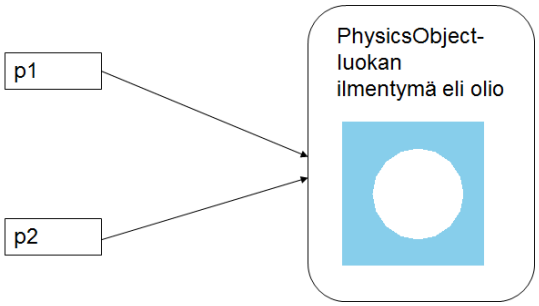


Kuva 8: Molemmat muuttujat, p1 ja p2, liikuttelevat samaa ympyrää. Lopputuloksena ympyrä seisoo pisteessä x=100.

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 metodit. Olion julkisia metodeja voidaan kutsua muualtakin kuin itse olion (luokan) koodista. 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.

```
1        string lause = "Pekka meni kauppaan";
2        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 parametreina 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 (`this`) esitellyn `pallo` -olion tietoja, joten enää ei ole kyse staattisesta aliohjelmasta, ja siksi `static`-sana jätetään pois. Metodi sen sijaan pystyy käyttämään *olion* omia "ominaisuuksia", attribuutteja, metodeja ja ns. ominaisuus-kenttiä (property fields). Muista, että olion omiin "asioihin" voisi viitata myös:

```
this.pallo.X = Mouse.PositionOnWorld.X;
```

eli jos aliohjelma tarvitsee `this` -viitettä, se on metodi (eli ei-staattinen).

8.6 Olion tuhoaminen ja roskienkeruu

Kun olioon ei enää viittaa yhtään muuttujaa (olioviitettä), 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 *puhdistusoperaatio* (kutsutaan usein myös nimellä roskienkerääjä, *garbage collector*) vapauttaa merkittyjen olioiden muistipaikat.

Kaikissa ohjelmointikielissä näin ei ole (esim. alkuperäinen C++), vaan muistin vapauttamisesta ja olioiden tuhoamisesta tulee useimmiten huolehtia itse. Näissä kielissä on yleensä destruktori (*destructor* = hajottaja), joka suoritetaan aina kun olio tuhotaan. Itse kirjoitettavasta destruktorista on tapana kutsua olion elinaikanaan luomien olioiden tuhoamista tai muiden resurssien vapauttamista. Vertaa konstruktoriin, joka suoritettiin kun olio luodaan. Haastavaksi näiden kielien yhteydessä tuleekin se, että joissakin tapauksissa olioiden elinkaari on automaattista ja

joissakin ei. Tästä seuraa helposti muistivuoto, eli jokin muistialue unohtuu vapauttaa, mutta siihen ei ole enää yhtään osoitinta, jolla siihen päästäisiin käsiksi ja näin muistialue jää varatuksi koko ohjelman loppuajaksi. Siksi muistivuodot ovat erittäin yleisiä aloittelevilla C++ -ohjelmoijilla. Javan ja C#:in kaltaiset kielet ovat tuoneet valtavan helpotuksen muistivuotojen välttämiseen.

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 viimeistään 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 resursseihin (esim tiedostoon, tosin yleensä tiedostoa ei ole hyvä pitää avoinna niin kauan aikaa kuin jonkin olion elinkaari voi olla).

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 https://learn.microsoft.com/en-us/dotnet/api/system.string?view=net-7.0, jossa on muun muassa lista *jäsenistä* eli käytössä olevista konstruktoreista, attribuuteista (fields), ominaisuuksista (property) ja metodeista.

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.

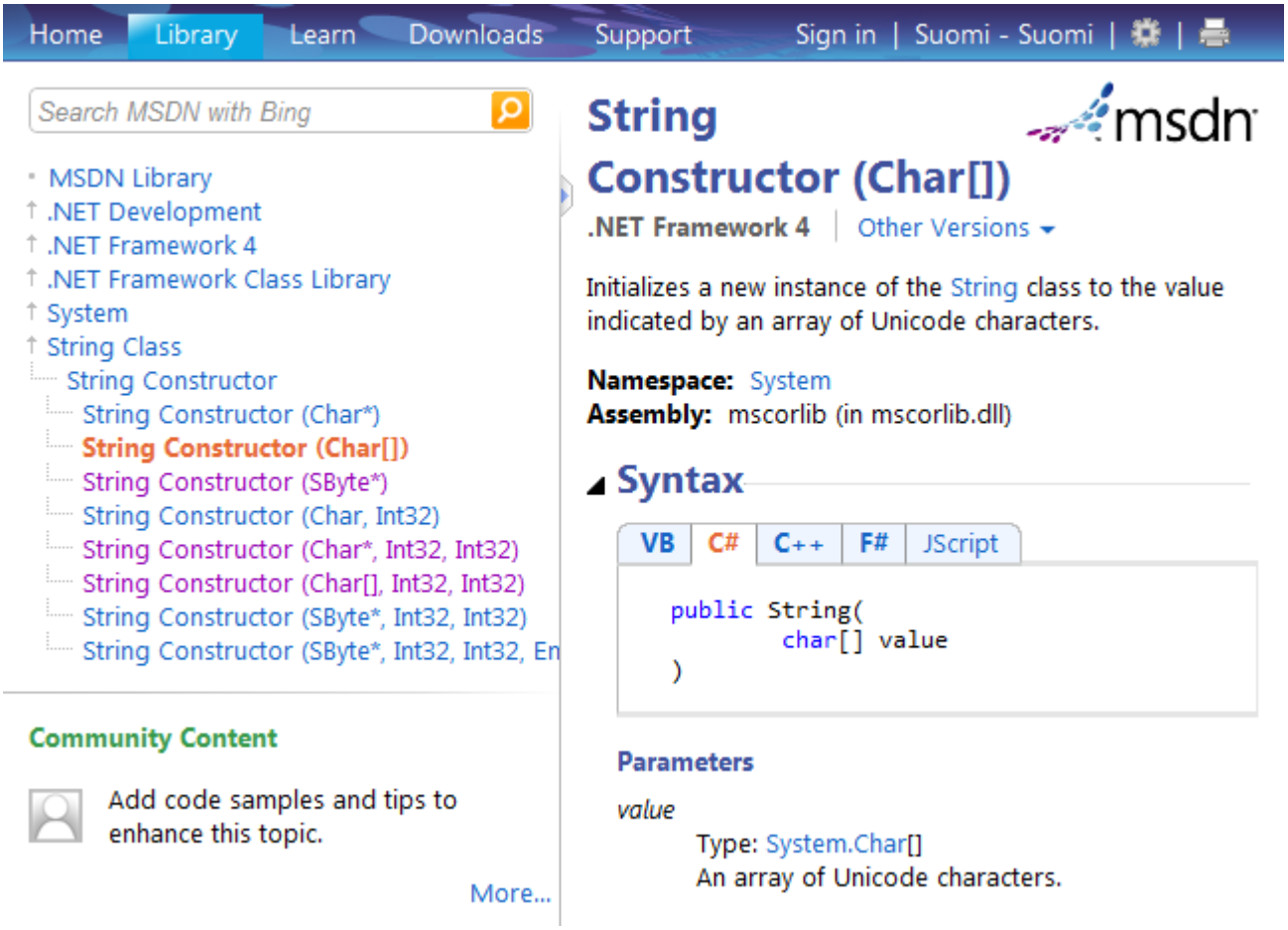
Kunkin konstruktorin kohdalla on lyhyesti kerrottu mitä se tekee, ja sen jälkeen minkä tyyppisiä ja montako parametria konstruktori ottaa vastaan. Kaikista konstruktoreista saa lisätietoa klikkaamalla konstruktorin esittelyriviä. Esimerkiksi linkki

[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.



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

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.

```
1         string nimi = new String(new char [] {'J', 'a', 'n', 'n', 'e'});
2         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 lause on vastaava kuin edellisessä kohdassa, mutta lyhyempi kirjoittaa.

```
1         string nimi = "Janne";
2         Console.WriteLine(nimi); // Tulostaa 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 -> Luokkalista -> Jypeli -> PhysicsObject), löydämme useita eri konstruktoreita (ks. kohta *Staattiset julkiset jäsenfunktiot*, jotka alkavat sanalla PhysicsObject). Konstruktoreista järjestyksessä toinen saa parametreina kaksi lukua ja muodon. Tätä konstruktoria käytimme jo lumiukkoesimerkissä.

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.

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

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 versio on 6, ja onkin syytä varmistua, että löytämäsi dokumentaatio koskee juuri oikeaa versiota. Voit esimerkiksi käyttää hakutermissä versionumeroa tähän tapaan: "c# string documentation .net 6". 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*, tai *type cast*).

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

```
1        // kokonaisluku merkkijonoksi
2        int kokonaisluku = 24;
3        string intMerkkijonona = kokonaisluku.ToString();
```

```
1        // liukuluku merkkijonoksi
2        double liukuluku = 0.562;
3        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* (struct), josta löytyy alkeistietotyyppien käsittelyyn hyödyllisiä metodeita. Rakenteet sijaitsevat System-nimiavaruudessa, ja tästä syystä ohjelman alussa tarvitaan lause

```
using System;
```

Alkeistietotyyppejä vastaavat rakenteet löytyvät 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).

```
1         int luku1 = 5;
2         Int32 luku2 = 6;
```

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.

Kun olet kokeillut, kokeile vaihtaa jonoon jotakin mikä ei ole numero. Mitä tapahtuu?10pt

```
1         string jono = "24";
2         int luku2 = int.Parse(jono);
```

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.

```
1         string jono = "2.45";
2         double luku = Double.Parse(jono);
```

Käytännössä jos tieto saadaan ihmisen syöttämänä, niin on erittäin todennäköistä, että se ei muodosta laillista numeroa. Siksi usein kannattaa käyttää funktiota `TryParse`:

```
1         string jono = "2.45";
2         double luku = 5;
3         bool onnistui;
4         onnistui = Double.TryParse(jono,out luku);
```

Asiaa vielä monimutkaistaa se, että käyttöjärjestelmän desimaalierotin saattaa olla pilkku (,) tai piste (.).

Luku 9

Aliohjelman paluuarvo

Muokkaa ohjelma toimivaksi. Laita pääohjelma ennen muita aliohjelmia. 10pt

```
1  public class Vahennys
2  {
3          public static void Main()
4          {
5              int luku = 102;
6              System.Console.WriteLine(Vahenna(luku, 3000));
7          }
8          public static double Vahenna(double luku, double montakoVahennetaan)
9          {
10             double tulos = luku - montakoVahennetaan;
11             return tulos;
12         }
13 }
```

Aliohjelmat-luvussa tekemämme Lumiukko-aliohjelma ei palauttanut mitään arvoa. Usein on kuitenkin hyödyllistä, että lopettaessaan aliohjelma palauttaa jotain tietoa aliohjelman 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 lähes aina 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 (eli palataan takaisin kutsuvaan ohjelman osaan).

Yleensä aliohjelmaa joka palauttaa arvon, sanotaan funktioksi.

9.1 Keskiarvon laskeva funktio

Luvun sisältö videona, jota voit katsoa samaan aikaan kun luet tätä lukua:

- Keskiarvo-funktion kirjoittaminen ja kutsuminen Luento 5 ndash; 35m0s (50m0s)

Ennen funktion toteuttamista suunnitellaan, että sitä kutsuttaisiin seuraavasti:

```
double keskiarvo;
keskiarvo = Keskiarvo(3, 4);
```

Eli kun funktiosta palataan, se palauttaa laskemansa tuloksen, ja kutsuva sijoittaa saamansa tuloksen apumuuttujaan.

Toteutetaan nyt kyseinen funktio.

```
1     public static double Keskiarvo(int a, int b)
2     {
3        double keskiarvo;
4        keskiarvo = (a + b) / 2.0; // Huom 2.0, jotta reaaliluku
5        return keskiarvo;
6     }
```

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 parametreina 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.

9.2 Funktion kutsuminen

Aliohjelmaa voitaisiin nyt käyttää pääohjelmassa esimerkiksi alla olevalla tavalla.

Kokeile laskea muidenkin lukujen keskiarvoja. Kokeile myös kutsua Keskiarvo(2+3, 4+7)10pt

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

Kutsu voitaisiin kirjoittaa myös lyhyemmin:

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

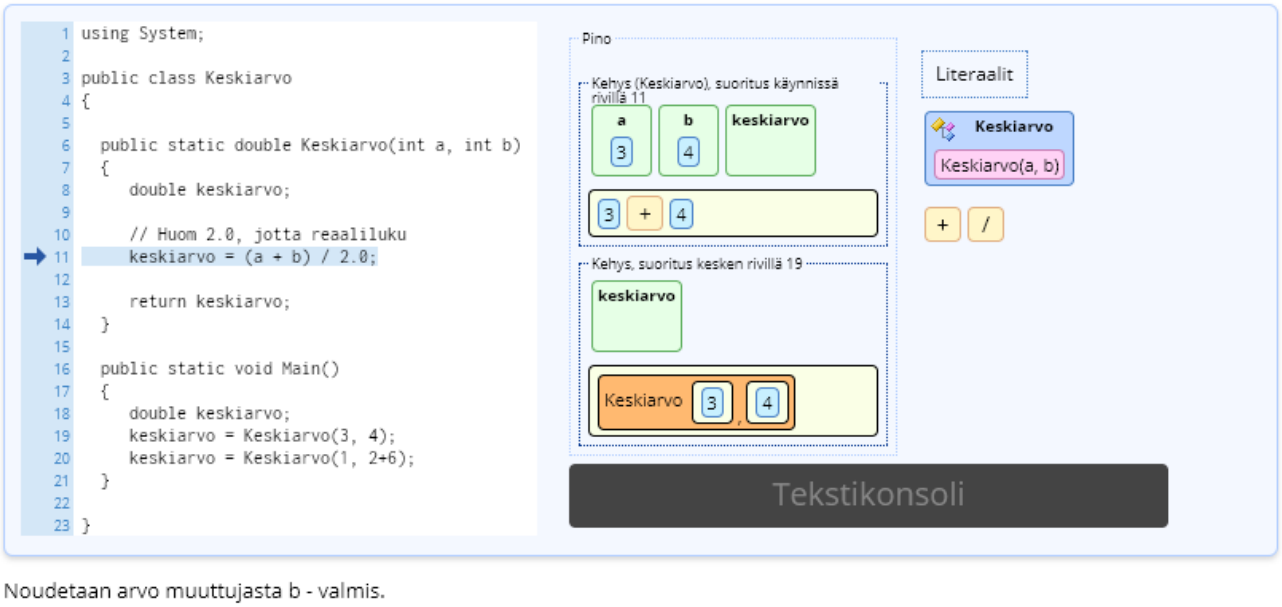
Koska Keskiarvo-aliohjelma palauttaa aina double-tyyppisen liukuluvun, voidaan kutsua käyttää kuten mitä tahansa double-tyyppistä arvoa. Se voidaan esimerkiksi tulostaa tai tallentaa muuttujaan.

Alla olevassa animaatiossa on ensin kirjoitettu funktio ja sitten pääohjelma. Näiden järjestyksellähän ei ole väliä C#-kielessä. Ohjelman suoritus aloitetaan aina pääohjelmasta, ja aliohjelmia suoritetaan niiden kutsumisjärjestyksessä, olipa aliohjelmien lähdekoodi kirjoitettu mihin kohtaan tahansa luokan sisällä.

Alla olevassa animaatiossa on kaksi peräkkäistä kutsua, jotka havainnollistavat aliohjelman kutsuja eri arvoilla. Jälkimmäisessä kutsussa nähdään miten kutsun yhteydessä lasketaan lausekkeen arvo. Eli funktion (ja minkä tahansa aliohjelman) kutsussa voi olla mitä tahansa lausekkeita, jotka tuottavat tyypiltään sellaisen arvon, joka voidaan sijoittaa vastinparametrille. Tässä tapauksessa `2+6` on lauseke, jonka arvo on `int` ja aliohjelman vastinparametri, nimeltään `b`, on myös tyypiltään `int`. Jatkossa huomaamme että lauseke voi sisältää myös funktiokutsuja.

Animaatio: Tutki funktion kutsua

Askella silmukan suoritusta vihreällä nuolella Tutki funktion kutsua



Animaatio: Keskiarvo

9.3 Funktion kirjoittaminen toisella tavalla

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.

Jos funktiota tarvitsee debugata, silloin se on helpointa mikäli osatuloksia on laskettu apumuuttujiin. Tällöin voi olla että yhdelle riville kirjoitettua funktiota voi joutua paloittelemana takaisin osiin.

Testien yksi tarkoitus on pitää huolta siitä, että vaikka toteutusta muuttaa, niin tuloksen oikeellisuus on helpompi tarkistaa. Pitää tosin silti muistaa, että testit eivät koskaan todista että joku toimii kaikissa tapauksissa! Katso edellisessä esimerkissä testit painamalla `Näytä koko koodi` ja ja myös testit painamalla `Test`. Katso myös syntyvät dokumentaatio painamalla `Document`.

9.4 Useita return-lauseita

Aliohjelmassa voi olla myös useita `return`-lauseita. Tästä esimerkki kohdassa: 13.5.1. Mikäli koodissa on useita `return`-lauseita, pitää niistä "ylimääräisten" olla ehdollisesti suoritettavia.

Usein pidetään kuitenkin riskinä koodia, jossa on useita return-lauseita. Hyvä esimerkki on sellainen, missä esimerkiksi ensin on tehty koodi, joka jossakin tilanteessa laskee jotakin ja palauttaa sen:

```
// ...
if ( a < 0 ) return summa / lkm;
// ...
return summa/lkm;
```

Kun koodia on testattua useilla arvoilla huomataankin, että `lkm` voi olla nolla ja muutetaan koodia:

```
// ...
if ( a < 0 ) return summa / lkm;
// ...
if ( lkm == 0 ) return 0;
return summa/lkm;
```

Mikä nyt menee pieleen? Se, että ensimmäisessäkin `return`-lauseessa voi olla tilanne missä `lkm` on nolla.

On makuasia välttääkö useita poistumiskohtia vaiko ei. Usein return-lauseiden kanssa saa myös koodista selkeämpää, kun ei tule paljoa sisäkkäisiä lohkoja.

9.5 Funktio palauttaa yhden arvon

Aliohjelma voi palauttaa kerrallaan vain yhden arvon, kuten yhden `int`-luvun, yhden `string`-jonon tai yhden `PhysicsoObject`-olion.

Tätä rajoitetta voidaan kiertää muutamalla tavalla. Ensinnäkin, voidaan tehdä tietorakenne, joka sisältää useita arvoja. Funktio voi sitten palauttaa tämän (yhden) tietorakenteen. Toinen keino olisi luoda olio, joka sisältäisi useita arvoja. Tästä esimerkkinä on PhysicsObject: se sisältää useita eri arvoja, kuten leveyden, korkeuden, massan ja värin.

C#:ssa on olemassa kolmaskin keino, jota vain sivuamme tällä kurssilla: `ref`- ja `out`-parametrit.

Metodeita ja aliohjelmia, jotka ottavat vastaan parametreja ja palauttavat arvon, sanotaan *funktioiksi*. Nimitys ei ole hullumpi, jos vertaa Keskiarvo-aliohjelmaa vaikkapa matematiikan funktioon $f(x, y) = (x + y)/2$.

Funktioiden tulisi olla sellaisia, että ne toimivat parametreina saatujen tietojen avulla, eivätkä tarvitse toimiakseen muuta tietoa ohjelmasta. Vastaavasti parametrina saatujen arvojen muuttamista pitäisi välttää. Puhtaasti funktionaalisessa ohjelmoinnin ajattelutavassa funktiolla ei ole sivuvaikutuksia. Sivuvaikutuksia ovat esimerkiksi ruudulle tulostaminen tai ohjelman tilan muuttaminen. Olio-ohjelmointiin perustuvassa ajattelussa (ja myös tällä kurssilla) tästä vaatimuksesta joudutaan joissain kohdissa hieman tinkimään. Esimerkiksi Jypeli-peleissä funktiot usein muuttavat pelin tilaa esimerkiksi lisäämällä pelikentälle uuden olion, ja siten ne eivät ole täysin vapaita sivuvaikutuksista.

9.6 Funktion kutsu maksaa

Mitä eroa on tämän

```
1         double tulos = Keskiarvo(5, 2); // lasketaan Keskiarvo
2         Console.WriteLine(tulos); //tulostaa 3.5
3         Console.WriteLine(tulos); //tulostaa 3.5
```

ja tämän

```
1         Console.WriteLine(Keskiarvo(5, 2)); //tämäkin tulostaa 3.5
2         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 tietenkään käytännön merkitystä, mutta mikäli `Keskiarvo`-aliohjelmaa kutsuttaisiin hyvin monta kertaa, se alkaisi jossain vaiheessa näkyä ohjelman suoritusajassa. Kannattaa opetella tapa, ettei ohjelmassa tehtäisi *mitään* turhia suorituksia.

9.7 YmpyranAla, esimerkki yhden parametrin funktiosta

Edellisessä esimerkissä on funktiolle viety kaksi parametria. Paramaterien määrä riippuu ihan tarpeesta ja voi olla mitä tahansa 0:sta n:ään. Tosin parametrittomat funktiot ovat aika harvinaisia.

Seuraavaksi vielä esimerkki yhden parametrin funktiosta:

```
1 //
2     /// <summary>
3     /// Kutsutaan malliksi funktioita
4     /// </summary>
5     public static void Main()
6     {
7         double ala;
8         ala = YmpyranAla(2);
9         Console.WriteLine("Ympyrän ala on {0:0.00}", ala);
10    }
11
12    /// <summary>
13    /// Lasketaan ympyrän pinta-ala
14    /// </summary>
15    /// <param name="r">ympyrän säde</param>
16    /// <returns>ympyrän pinta-ala</returns>
17    /// <example>
18    /// <pre name="test">
19    ///   YmpyranAla(1) ~~~ 3.1415926;
20    ///   YmpyranAla(2) ~~~ 12.5663706;
```

```
21      /// </pre>
22      /// </example>
23      public static double YmpyranAla(double r)
24      {
25          return Math.PI * r * r;
26      }
```

Muista että edellistä funktiota voisit kutsua myös millä tahansa seuraavista tavoista (kokeile esimerkkiin):

```
...
double sade = 2.1;
ala = YmpyranAla(sade);  // luonnollisesti muuttujalla
...
ala = YmpyranAla(sade + 7.0); // ja millä tahansa lausekkeella joka tuottaa doubl
...
YmpyranAla(12);  // näinkin voi kutsua, mutta tässä ei sinällään ole järkeä
                 // tässä tapauksessa kun tulosta ei oteta vastaan.
```

9.8 Tehtäviä funktioista

Kysymyksiä paluuarvosta

Mitkä seuraavista kommenteista pitää paikkaansa:

	True	False
public static void Main() // ei palauta mitään. 	☐	☐
public static int PalautaSuurin() // palauttaa kokonaisluvun 	☐	☐
public static char Tulosta() //palauttaa kirjaimia. 	☐	☐
public static string PoistaVika() // palauttaa merkkijonon. 	☐	☐

public static int Lisaa(int ika, double pituus)
 // palauttaa kokonaisluvun
 // ja liukuluvun.
 	
	☐	☐

9.8.1 Harjoituksia aliohjelmista

Muuttujat-luvun lopussa tehtiin ohjelma, joka laski painoindeksin. Tee ohjelmasta uusi versio, jossa painoindeksin laskeminen tehdään funktiossa. Funktio saa parametreina pituuden ja painon ja palauttaa painoindeksin. Tuloksen tulostaminen tapahtuu pääohjelmassa.

Tehtävä: Painoindeksi

```
1  public class Painoluokka
2  {
3      public static void Main()
4      {
5        double pituus = 1.83;
6        double paino = 75.0;
7        double painoindeksi = paino / (pituus*pituus);
8        System.Console.WriteLine(painoindeksi);
9     }
10 }
```

Mallivastaus

Mallivastaus

```
1  public class Painoluokka
2  {
3      public static void Main()
4      {
5        double pituus = 1.83;
6        double paino = 75.0;
7        double painoindeksi = Painoindeksi(pituus, paino);
8        System.Console.WriteLine(painoindeksi);
9     }
10
11
12    public static double Painoindeksi(double pituus, double paino)
13    {
14        double tulos = paino / (pituus*pituus);
15        return tulos;
16    }
17 }
```

Tehtävä: Aliohjelmat

Kirjoita kutsuja vastaavat funktiot niin että ohjelma toimii. Älä muuta aliohjelmakutsuja, kirjoita pelkät aliohjelman toteutukset. Uusia aliohjelmia/funktioita tarvitaan yhteensä neljä.10pt

```
1  public class Funktioita
2  {
3      public static void Main()
4      {
5        double summa = LaskeSumma(5, 6.6, 7);
6        double erotus = LaskeErotus(7, 9);
7        int luku = MiinustaYksi(9);
8        Tulosta(summa, erotus, luku);
9     }
10 }
```

- Katso mallivastaus videona (12m48s)

Mallivastaus

```
public class Funktioita
{
    public static void Main()
    {
      double summa = LaskeSumma(5, 6.6, 7);
      double erotus = LaskeErotus(7, 9);
      int luku = MiinustaYksi(9);
      Tulosta(summa, erotus, luku);
   }


   public static void Tulosta(double a, double b, int c)
   {
       System.Console.WriteLine($"{a} {b} {c}");
   }


   public static int MiinustaYksi(int luku)
   {
       return luku - 1;
   }


   public static double LaskeErotus(double a, double b)
   {
       return a - b;
   }


   public static double LaskeSumma(double a, double b , double c)
   {
       double tulos = a + b + c;
       return tulos;
   }
}
```

Tehtävä: Kuormittaminen

Kirjoita aliohjelmat nyt niin että lukujen summan voi laskea kahdesta tai kolmesta luvusta. Älä muuta aliohjelmakutsuja, kirjoita pelkät aliohjelman toteutukset. Kertaa tarvittaessa asiaa aliohjelmien kuormittamisesta.10pt

```
public class Funktioita
{
    public static void Main()
    {
      double summa = LaskeSumma(5, 6.6, 7);
      double summa2 = LaskeSumma(7, 9);
      System.Console.WriteLine($"{summa} {summa2}");
   }
}
```

- Katso mallivastaus videona (4m22s)

Mallivastaus

Mallivastaus 1

```
public class Funktioita
{
    public static void Main()
    {
      double summa = LaskeSumma(5, 6.6, 7);
      double summa2 = LaskeSumma(7, 9);
      System.Console.WriteLine($"{summa} {summa2}");
   }


   public static double LaskeSumma(int a, int b)
   {
       return a + b;
   }


   public static double LaskeSumma(int a, double b, int c)
   {
       return a + b + c;
   }
}
```

Mallivastaus 2

```
public class Funktioita
{
    public static void Main()
    {
      double summa = LaskeSumma(5, 6.6, 7);
      double summa2 = LaskeSumma(7, 9);
      System.Console.WriteLine($"{summa} {summa2}");
   }


   public static double LaskeSumma(double a, double b, double c=0)
   {
       return a + b + c;
   }
}
```

9.8.2 Harjoituksia funktioista

Olkoon meillä seuraavanlainen ohjelma, jonka (funktio)aliohjelma on vielä kesken.

```
XXX YYY ZZZ KolmionAla (??? luku1, IIII luku2) {}.
```

Mieti alla olevan pääohjelmassa olevan kutsun perustella oikeat sanat kuhunkin kohtaan. Täydennä sen jälkeen lopullinen funktio `KolmionAla` toimimaan oikealla tavalla. Katso sitten alla olevalta videolta oikea vastaus.

Tehtava: KolmionAla

Täydennä lopullinen ohjelma tähän: 10pt

```
using System;

/// <summary>
/// Esimerkki aliohjelmista
/// </summary>
public class FunktioitaNC
{
    /// <summary>
    /// Lasketaan keskiarvoja
    /// </summary>
    public static void Main()
    {
        double kanta = 15.0;
        double korkeus = 10.0;
        double ala;

        ala = KolmionAla(kanta, korkeus);
        Console.WriteLine(ala);
    }


    /// <summary>
    /// Lasketaan kolmion pinta-ala
    /// </summary>
    /// <param name="luku1">kanta</param>
    /// <param name="luku2">korkeus</param>
    /// <returns>pinta-ala</returns>
    XXX YYY ZZZ KolmionAla(??? luku1, IIII luku2)
    {
    }
}
```

Vastausvaihtoehdot

0	1	2	3	4	5	6
void	static	public	int	double	char	string

Vastaukset

- Katso mallivastaus videona (10m5s)

Mallivastaus

Mallivastaus

```
using System;

/// <summary>
/// Esimerkki aliohjelmista
/// </summary>
public class FunktioitaNC
{
    /// <summary>
```

```
9       /// Lasketaan keskiarvoja
10      /// </summary>
11      public static void Main()
12      {
13          double kanta = 15.0;
14          double korkeus = 10.0;
15          double ala;
16
17          ala = KolmionAla(kanta, korkeus);
18          Console.WriteLine(ala);
19      }
20
21
22      /// <summary>
23      /// Lasketaan kolmion pinta-ala
24      /// </summary>
25      /// <param name="a">kanta</param>
26      /// <param name="b">korkeus</param>
27      /// <returns>pinta-ala</returns>
28      /// <example>
29      /// <pre name="test">
30      ///     KolmionAla(0.0, 0.0) ~~~ 0.0;
31      ///     KolmionAla(2.0, 1.0) ~~~ 1.0;
32      ///     KolmionAla(2.0, 3.0) ~~~ 3.0;
33      ///     KolmionAla(1.0, 1.0) ~~~ 0.5;
34      /// </pre>
35      /// </example>
36      public static double KolmionAla(double a, double b)
37      {
38          return a * b / 2.0;
39      }
40  }
```

Tehtävä 9.8.5 Henkilön ikä

Kirjoita kokonainen luokka, jossa on pääohjelma ja aliohjelma. Aliohjelma palauttaa henkilön iän, kun sille viedään parametreina tämä vuosi sekä syntymävuosi. Kirjoita myös dokumentaatiokommentit. Syntyneen dokumentaation näet Document-linkistä.10pt

- Katso mallivastaus videona (8m53s)

Mallivastaus

Mallivastaus

Kirjoita kokonainen luokka, jossa on pääohjelma ja aliohjelma. Aliohjelma palauttaa henkilön iän, kun sille viedään parametreina tämä vuosi sekä syntymävuosi. Kirjoita myös dokumentaatiokommentit.10pt

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  /// @author vesal
7  /// @version 6.11.2021
```

```
8  /// <summary>
9  /// Kokeillaan Ika-funkiton toimintaa
10 /// </summary>
11 public class LaskeIka
12 {
13     /// <summary>
14     /// Kutsutaan Ika-funktiota
15     /// </summary>
16     public static void Main()
17     {
18         int vuosi = 2021;
19         int syntymavuosi = 1959;
20         int ika;
21
22         ika = Ika(vuosi, syntymavuosi);
23         Console.WriteLine($"Henkilön ikä on {ika}");
24     }
25
26
27     /// <summary>
28     /// Laskee iän syntymävuoden ja annetun vuoden avulla
29     /// </summary>
30     /// <param name="vuosi">vuosi jona ikä lasketaan</param>
31     /// <param name="syntymavuosi">henkilön syntymävuosi</param>
32     /// <returns>ikä</returns>
33     /// <example>
34     /// <pre name="test">
35     ///     Ika(2021, 2021) === 0;
36     ///     Ika(2021, 2000) === 21;
37     ///     Ika(2021, 1959) === 62;
38     ///     Ika(2020, 1959) === 61;
39     /// </pre>
40     /// </example>
41     public static int Ika(int vuosi, int syntymavuosi)
42     {
43         int tulos = vuosi - syntymavuosi;
44         return tulos;
45     }
46
47 }
```

Luku 10

Ohjelmoijan työkaluja: Git, IDE

10.1 Git

Käytännön ohjelmistokehitystyössä ohjelmistojen kehittämiseen osallistuu aina useampi henkilö yhteistyönä. Yhtenäisten koodien varmistamiseksi käytetään niin kutsuttuja versionhallintatyökaluja, joista nykyisin yleisimmin käytetty on Git. Git on kehitetty Linus Torvaldsin toimesta.

Git-versiohallinnan pääidea on mahdollistaa ohjelmakoodiin tehtyjen muutosten seuranta ja hallinta. Versiohallintaan tallennetaan ohjelmistoon tehdyt muutokset aikajärjestyksessä. Tämä mahdollistaa yhteistyön useiden kehittäjien kesken samanaikaisesti ja tarjoaa mahdollisuuden palata aiempiin versioihin tarvittaessa. Git tallentaa kunkin kehittäjän tekemät muutokset erillisinä "commiteina", jotka voidaan yhdistää pääkehityshaaraan ("main"), tai ns. haarojen ("branch") kautta, mikä helpottaa uusien ominaisuuksien lisäämistä ja virheiden korjaamista eristyksissä.

Tällä kurssilla käytämme vain muutamia Git-versiohallinnan ominaisuuksia, mutta suosittelemme tutustumaan myös muihin ominaisuuksiin, kuten haarojen käyttöön, mikäli aiot jatkaa ohjelmistokehityksen parissa.

Netissä on lukuisia palveluita, joissa voidaan säilyttää ja julkaista Git-versiohallintaa käyttäviä projekteja. Eräitä tunnettuja ovat GitHub ja GitLab. Näihin palveluihin on rakennettuna myös tikettijärjestelmä, johon lisätään havaittuja bugeja ja kehitysehdotuksia kortteina. Kehittäjä valitsee näistä korteista yhden tai useampia ja työskentelee niiden parissa. Kun kortti on valmis, se siirretään valmiiden korttien joukkoon, ja haarassa oleva koodi yhdistetään versiohallinnan pääkehityshaaraan. Tikettijärjestelmä ei siis ole osa Git-versiohallintaa, vaan yksi lisäpalvelu käytettäväksi Gitin ohessa. Git on ilmainen, mutta lisäpalvelut usein maksavat.

Voit tarkastella esimerkiksi TIMin GitHubiin kirjattuja tikettejä tästä linkistä.

Ohjeet Git-versiohallinnan asentamiseksi ja käyttämiseksi tällä kurssilla löydät työkalut-sivulta.

10.2 Integroitu kehitysympäristö, IDE

Tässä monisteessa olemme tähän saakka kirjoittaneet ohjelmat suoraan TIMin koodausikkunoihin, sekä mahdollisesti tekstieditoriin (Luku 2.1). Ohjelman koon kasvaessa kannattaa ottaa käyttöön sovelluskehitin eli IDE (*Integrated Development Environment*).

IDE kokoaa yhteen monia eri työkaluja, kuten

- tekstieditorin (joka yleensä ymmärtää kohdekieltä tavallista editoria paremmin),
- kielen kääntäjän,
- ns. assettien, kuten kuvien ja äänien hallinnan,
- linkitystyökalut,
- versionhallintatyökalut, ja
- virheenjäljitystyökalut, eli debuggerin.

C#:lle hyviä IDEjä ovat JetBrains Rider (tämän kurssin suositus) sekä Visual Studio Community. Työkalut-sivulla on linkit uusimpiin versioihin ja asennusohjeisiin.

Kaikenlaiset pilvipalvelut ovat yleistyneet, ja myös pilvipohjaisia kehitysympäristöjä on olemassa. Kuitenkin edelleen yleinen käytäntö ohjelmoinnin opiskelussa, kuten myös Ohjelmointi 1 -kurssilla, on asentaa kehitysympäristö omalle paikalliselle tietokoneelle. Tämä lähestymistapa tarjoaa useita merkittäviä etuja.

- Suorituskyky: Paikallisella asennuksella hyödynnät tietokoneesi tehon täysimääräisesti, mikä yleensä tarkoittaa nopeampaa koodin kääntämistä, suorittamista ja vianmääritystä verrattuna etäympäristöihin.
- Täysi kontrolli: Sinulla on täysi hallinta asetuksista ja konfiguraatiosta. Voit mukauttaa ympäristöä omiin tarpeisiisi ja työskennellä ilman riippuvuutta ulkopuolisista palveluista.
- Hinta: Pilvipohjaiset kehitysympäristöt, jotka täyttävät Ohjelmointi 1 -kurssin osaamistavoitteet, kuten debuggaus, eivät yleensä ole ilmaisia. Omalle koneelle asennettu kehitysympäristö on maksuton.
- Toimialan standardi: Paikallisen kehitysympäristön asentaminen vastaa alan normeja ja toimintatapoja.

10.3 IDEn käyttö

10.3.1 Käyttöönotto, solutionit ja projektit

Riderin käyttöönottoon sekä solutionien ja projektien hallintaan on ohjeet kurssin työkalut-sivulla.

10.3.2 Ohjelman kirjoittaminen

Kannattaa aina muuttaa kooditiedoston (esim. ConsoleMain.cs) nimi kuvaavampaan. Klikkaamalla `Solution Explorerissa` kooditiedoston päällä hiiren oikealla napilla ja valitsemalla Edit -> Rename voit valita tiedostolle uuden nimen.

10.3.3 Ohjelman kääntäminen ja ajaminen

Ohjelman kääntäminen ja ajaminen tapahtuu joko Run- tai Debug-painikkeella. Debug-painikkeesta Rider kääntää ja suorittaa ohjelman ns. debug-tilassa, ja vastaavasti Run-painikkeella ohjelma käännetään ja suoritetaan ilman debug-tilaa. Ohjelman kehityksen aikana on kuitenkin usein hyödyllistä ajaa ohjelma nimenomaan debug-tilassa, jolloin mahdolliset ohjelman ajonaikaiset virhetilanteet saadaan näkyviin IDEen.

Jos haluamme lopettaa ohjelman suorituksen jostain syystä kesken kaiken, onnistuu se painamalla Stop-painiketta tai näppäimistöllä `Shift+F5`.

10.4 Debuggaus

Virheet ohjelmakoodissa ovat valitettava tosiasia. On olemassa pieniä virheitä, jotka eivät vaikuta ohjelman toimintaan, mutta on olemassa myös vakavia virheitä. Tällaiset vakavat virheet kaatavat ohjelman tai muutoin estävät sen oikean toiminnan.

Syntaksivirheet 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 joissain olosuhteissa. Tällaiset virheet, joissa 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.

IDE:n debuggeri mahdollistaa ohjelman tilan, kuten muuttujien arvojen tarkastelun ohjelman suorituksen aikana. Tämä auttaa huomattavasti virheen tai epätoivotun toiminnan syyn selvittämisessä. “Vanha tapa” tehdä samaa asiaa on lisätä ohjelmaan tulostuslauseita, jolloin esimerkiksi muuttujan tai lausekkeen arvo tulostetaan näytölle tai lokitiedostoon. Vanha tapa on kuitenkin edelleen sinänsä toimiva tapa tai joissain tilanteissa jopa ainoa tapa, koska debuggerin käyttö ei ole aina mahdollista. Esimerkiksi web-kehityksessä tulostusdebuggaus on edelleen hyvin tavallista.

Ohjelman tilan tutkiminen aloitetaan asettamalla ensin keskeytyskohta (engl. breakpoint) siihen kohtaan, jossa oletamme virheen olevan. Keskeytyskohta on kohta, johon haluamme ohjelman suorituksen väliaikaisesti pysähtyvän. Ohjelman pysähdyttyä voidaan tutkia ohjelman tilaa ja suorittaa ohjelmaa lause kerrallaan. Jos haluamme suorittaa lause kerrallaan ohjelman alusta saakka, asetetaan keskeytyskohta ohjelman alkuun.

Aseta keskeytyskohta kursorin kohdalle painamalla F9 tai klikkaa koodi-ikkunassa rivinumeroiden vasemmalle puolelle harmaalle alueelle. Keskeytyskohta näkyy punaisena pallona ja rivillä oleva (ensimmäinen) lause värjättynä punaisella.

Kun keskeytyskohta on asetettu, klikataan ylhäältä Debug-painiketta tai painetaan `F5`.

Ohjelman suoritus on nyt pysähtynyt siihen kohtaan, johon asetimme keskeytyskohdan. Avaa `Locals`-välilehti alhaalta, ellei se ole jo auki. Debuggaus-näkymässä `Locals`-paneelissa näkyvät kaikki tällä hetkellä näkyvillä olevat muuttujat (paikalliset, eli lokaalit muuttujat) ja niiden arvot. Keskellä näkyy ohjelman koodi ja keltaisella se rivi, jonka kohdalla 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 aliohjelmaan sisälle, ja 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.

Termi debug johtaa yhden legendan mukaan aikaan, jolloin tietokoneohjelmissa ongelmia ai-

heuttivat releiden väliin lämmittelemään päässeet luteet. Ohjelmien korjaaminen oli siis kirjaimellisesti hyönteisten (bugs) poistoa. Katso lisätietoja Wikipediasta:

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

- lue lisää debuggauksesta
  - sivulla on kerrottu myös vastaavat Mac painikkeet

10.5 Hyödyllisiä ominaisuuksia

10.5.1 Syntaksivirheiden etsintä

Luentovideo virheistä (1m13s)

Rider kääntää taustalla jatkuvasti koodia havaitakseen ja ilmoittaakseen mahdolliset syntaksivirheet. IDE:t kehittyvät hurjaa vauhtia, ja niin Rider kuin muutkin IDEt näyttävät jo kaikenlaisia muitakin ehdotuksia niin kirjoitustyylin parantamiseksi kuin potentiaalisten loogisten virheiden välttämiseksi. Riderissa näiden ehdotusten määrää voi säätää oikealla alhaalla olevasta värikynä-kuvakkeesta. On makuasia paljonko näitä ehdotuksia haluaa nähdä. Tällä kurssilla joistain ehdotuksista voi oppimisen kannalta olla jopa haittaa, joten voi olla järkevää säätää värikynä-kuvakkeesta liukusäädin Errors-kohtaan.

10.5.2 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.

10.5.3 Uudelleenmuotoilu

IDEen on tallennettu joukko sääntöjä, joilla IDE pyrkii automaattisesti muotoilemaan koodin tyyliä "kauniiksi" kirjoittamisen yhteydessä. Käyttäjä voi tarkoituksellisesti tai vahingossa rikkoa koodin näitä sääntöjä, kuten sisennyksiä. Tällöin koodi voidaan palauttaa vastaamaan IDEen tallennettuja sääntöjä komentamalla Code -> Reformat code.

Käyttäjä voi muuttaa tyyliin liittyviä sääntöjä kohdasta Settings -> Editor -> Code Style. "Oikeassa elämässä" tyylisääntöjä on aina syytä hienosäätää siten, että ne vastaavat tiimin tai

projektin konventioita. Tällä kurssilla oletustyylien muuttamiseen ei ole tarvetta.

10.5.4 TODO-tehtävien luettelo

Huomiota vaativa asia on tapana kirjoittaa koodiin muistiin `TODO:`-merkinnällä. Alla on esimerkki.

```
Console.WriteLine("Hello"); // TODO: Tässä pitäisi tulostaa Hello World!
```

IDE koostaa TODO-merkityistä kohdista tehtävälistan. Tehtävälistan saa tarvittaessa auki valikosta: View -> Tool Windows -> TODO. Rider varoittaa TODO-kohdista myös silloin, kun versiohallintaan tehdään commit.

10.6 Lisätietoja

- Riderin asennus, asetukset, sekä solutionin ja projektin luominen
- Debuggaus

Tarkista tietosi

Kävin kappaleen läpi ja asensin työkalut

- ○ Kyllä
- ○ Ei

Luku 11

Testaaminen

> "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

Ohjelman testaamisella tarkoitetaan ohjelman virheettömyyden tai laadun tutkimista. Testaamista voidaan tehdä käyttämällä ohjelmaa sellaisenaan, esimerkiksi kokeilemalla erilaisia käyttötapoja tai tulostamalla vaikkapa jonkin muuttujan tila ruudulle, ja siten tutkimalla toimiiko ohjelma odotetusti. Jo melko yksinkertaisten ohjelmien testaaminen tällaisilla tavoilla veisi kuitenkin paljon aikaa.

Tulostukset tai käsin kokeileminen pitäisi tehdä aina uudestaan, kun ohjelmoija tekee muutoksen ohjelman koodiin. Emme nimittäin voisi mitenkään tietää, että ennen muutosta tekemämme testit toimisivat vielä muutoksen jälkeen. Yksi testaamista helpottava tekniikka on *yksikkötestaus*. Yksikkötestauksen idea on, että jokaiselle ohjelman komponentille, kuten aliohjelmalle tai metodille, kirjoitetaan oma testinsä, jotka voidaan sitten kaikki ajaa kerralla. Näin voimme suorittaa kaikki kerran kirjoitetut testit jokaisen pienenkin muutoksen jälkeen uudelleen.

Yksi merkittävä suunnittelutapa on TDD (engl. *Test Driven Development*). Tällä tarkoitetaan sitä, että ennen koodin kirjoittamista mietitään miten tekeillä oleva asia voidaan testata ja mieluimmin vielä automaattisilla testeillä. Näin testaamisen ajattelu ohjaa suunnittelua ja varsinaista koodin kirjoittamista. Yhdistettynä testien etukäteen kirjoittamien yksikkötesteihin, saadaan nykykäsityksen mukaan tuottavammin laadukasta ohjelmakoodia.

11.1 Comtest

Rider, Visual Studio ja muut IDE:t mahdollistavat yksikkötestien (*unit tests*) kirjoittamisen erillisiin testiprojekteihin. Ongelmana on, että testiprojektien luominen ja varsinaisten testien kirjoittaminen on melko työlästä. Tähän on apuna ComTest-työkalu, joka hyödyntää IDEn sisäänrakennettua testausjärjestelmää, mutta madaltaa käytännön kynnystä kirjoittaa yksikkötestejä. ComTest on kehitetty IT-tiedekunnassa.

ComTest-testaustyökalun idea on, että testit kirjoitetaan yksinkertaisella syntaksilla aliohjelmien dokumentaatiokommentteihin suoraan ohjelman kooditiedostoon. Kommenteista luodaan varsinaiset testiprojektit ja -tiedostot. Samalla kirjoitetut testit toimivat dokumentaatiossa esimerkkinä aliohjelman tai metodin toiminnasta. Koska testien kirjoittamiskynnystä on madallettu, suosii tämä testien kirjoittamista siinä vaiheessa kun mietitään mitä esimerkiksi funktion pitäisi tehdä milläkin parametrien arvoilla. Näin päästään lähelle TDD:n tavoitteita. ComTest:n

asennusohjeet löytyvät sivulta:

- https://tim.jyu.fi/view/kurssit/tie/ohj1/tyokalut/rider.

Aliohjelman kirjoittamisen ja testaamisen vaiheet oli katsottu luvussa Aliohjelmien kirjoittaminen. Kertaa nuo askeleet!

11.2 Käyttö

Comtestin käyttö Luento 6 (11m11s)

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 sarkain-näppäintä). Tällöin Visual Studio luo valmiiksi paikan, johon testit kirjoitetaan. Dokumentaatiokommentteihin pitäisi ilmestyä seuraavat rivit.

```
/// <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. Huomaa että reaaliluvuille testin pitää tapahtua “melkein yhtäsuuruutena” (`~~~`). Ja jotta myös dokumentaatio syntyisi pitää noita `~~~` sisältäviä testirivejä olla parillinen määrä Doxygenissä olevan virheen takia.

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

Oikein käytettynä testejä tehdään siten, että testit kirjoitetaan ennen aliohjelman toteutusta. Aluksi aliohjelman toteutus on tynkä (minimaalinen koodi, joka on syntaksiltaan oikein). Sitten ajetaan testit ja katsotaan että ne palauttavat punaista (eli eivät mene läpi). Kun testit palauttavat punaista, voidaan toteuttaa aliohjelman toimimaan niinkuin se suuniteltiin ja sitten testien pitäisi palauttaa vihreää.

Testien avulla voidaan samalla suunnitella mitä erikoistapauksia aliohjelmassa tulee ottaa huomioon ja miten niiden kohdalla toimitaan. Esimerkiksi mitä on tyhjän taulukon keskiarvo. Kun testit ovat aliohjelman kommenteissa, voidaan myös osa erikoistapausten dokumentaatiosta jättää sanallisesti kirjoittamatta, koska niiden käyttäytyminen selviää testiesimerkeistä.

- Katso myös lisäesimerkkejä ComTesteistä.

11.2.1 Kirjoita tynkä-toteutus

Jotta testien syntaktinen toiminta ja kyky havaita virhe saataisiin kokeiltua, tehdään aliohjelmasta ensin **tynkä**. Tynkä on on syntaktisesti oikein oleva aliohjelma, mutta ei toteuta annettu ongelmaa ainakaan kaikille testiarvoille. `void`-aliohjelmille tyngäksi riittää tyhjä toteutus. Funktiolle tynkä voi olla `return`-lause jossa on lauseke, joka palauttaa funktion tyyppiä olevan arvon. Kokonaislukufunktiolle, esimerkiksi 0 voisi olla hyvä arvo.

Esimerkkejä **tynkä**-toteutuksista:

```
return;                          // void aliohjelmalle
return 0;                        // int, double tyyppisille funktiolle
return "";                       // string-tyyppisille funktioille
return new StringBuilder("");    // StringBuilder-funktiolle
return olio;                     // funktiolle jolle tulee olio parametrina
                                 // ja sen pitää palauttaa samaa tyyppiä
                                 // oleva tulos.
return null;                     // tätäkin voidaan käyttää oliotyypeille
                                 // (siis myös string, taulukko, StringBuilder)
return new int[0];               // palauttaa tyhjän int-taulukon
```

Kirjoitetaan esimerkiksi `Yhdista`-aliohjelma, joka yhdistää kahden annetun ei-negatiivisen luvun numerot toisiinsa. Aluksi kirjoitetaan aliohjelman otsikkorivi, aaltosulut ja niiden aliohjelman väliin tynkä-toteutus:

```
public static int Yhdista(int a, int b)
{
   return 0;
}
```

11.2.2 Kirjoita dokumentaatio ja testit

Sitten lisätään aliohjelman dokumentaatio (Visual Studiossa pohjan saamiseksi riittää kun kirjoittaa `///` aliohjelman otsikkorivin yläpuolelle).

Seuraavaksi tehdään aliohjelmalle testit.

```
1
2     /// <summary>
3     /// Yhdistää kahden ei-negatiivisen luvun numerot toisiinsa.
4     /// </summary>
5     /// <param name="a">Ensimmäinen luku</param>
6     /// <param name="b">Toinen luku</param>
7     /// <returns>Yhdistetty luku</returns>
8     /// <example>
9     /// <pre name="test">
10    /// Yhdista(0, 0) === 0;
11    /// Yhdista(1, 0) === 10;
12    /// Yhdista(0, 1) === 1;
13    /// Yhdista(1, 1) === 11;
14    /// Yhdista(13, 2) === 132;
15    /// Yhdista(10, 0) === 100;
16    /// Yhdista(10, 87) === 1087;
17    /// Yhdista(10, 07) === 107;
18    /// </pre>
```

```
19      /// </example>
20      public static int Yhdista(int a, int b)
21      {
22          return 0;
23      }
```

Katso edellisessä esimerkissä myös syntyvää dokumentiota painamalla `Document`-linkkiä. Sitten klikkaa luokan nimeä `Laskuja` ja siellä linkkiä `Yhdista`. Nyt näet minkälaiset esimerkit generoituvat aliohjelman dokumentaatiosta.

Nyt kun edellä oleva testi ajetaan (paina edellä `Test`-painiketta), saadaan ilmoitus että rivin

```
24      /// Yhdista(1, 0) === 10;
```

testi epäonnistuu, koska siltä odotettiin arvoa `10` mutta saatin arvo `0`. Tämän ansiosta tiedämme että testit pystyvät havaitsemaan ainakin osan tapauksista, joissa aliohjelma toimii väärin. Tällaiset esimerkkeihin perustuvat testit eivät valitettavasti voi koskaan havaita kaikkia mahdollisia aliohjelman virheellisiä toimintoja.

11.2.3 Toteuta aliohjelma ja aja testit

Kun meillä on syntaktisesti oikein oleva aliohjelma ja sen tynkä-toteutus, voidaan kirjoittaa aliohjelman (tässä esimerkissä funktion) toteutus, jonka pitäisi toteuttaa annettu ongelma ja selvitä testitapauksista.

Tässä aliohjelman toteutuksessa on (naiivisti) oletettu, että parametreina annettavat luvut täyttävät varmasti annetun ehdon (ei-negatiivinen). Myöhemmin opimme käsittelemään myös sellaiset tilanteet, joissa tästä ehdosta ollaan poikettu.

```
1
2       /// <summary>
3       /// Yhdistää kahden ei-negatiivisen luvun numerot toisiinsa.
4       /// </summary>
5       /// <param name="a">Ensimmäinen luku</param>
6       /// <param name="b">Toinen luku</param>
7       /// <returns>Yhdistetty luku</returns>
8       /// <example>
9       /// <pre name="test">
10      /// Yhdista(0, 0) === 0;
11      /// Yhdista(1, 0) === 10;
12      /// Yhdista(0, 1) === 1;
13      /// Yhdista(1, 1) === 11;
14      /// Yhdista(13, 2) === 132;
15      /// Yhdista(10, 0) === 100;
16      /// Yhdista(10, 87) === 1087;
17      /// Yhdista(10, 07) === 107;
18      /// </pre>
19      /// </example>
20      public static int Yhdista(int a, int b)
21      {
22          string ab = a.ToString() + b;
23          int tulos = int.Parse(ab);
24          return tulos;
25      }
```

Edellä oleva toteutus ei ole tehokkain mahdollinen, mutta testien ansiosta sitä voidaan muuttaa paremmaksi ja silti nopeasti varmistua, että toiminta pysyy kunnossa. Aja nyt em. testit painamalla `Test`-painiketta.

Kokeile muuttaa edellä toteutusta vaikkapa niin, että vaihdat `return`-lauseen tilalle:

```
return tulos+1;
```

Aja sitten testit uudelleen. Palauta alkuperäinen muoto ja aja taas testit.

Tarkastellaan testejä nyt hieman tarkemmin.

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

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

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

Seuraavaksi testataan, että jos parametreista 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 Visual Studio -testiprojektin 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.

11.2.4 Yleistä testeistä

Testit ovat periaatteessa aivan tavallinen aliohjelman osa, jossa suoritetaan kutsut testattavaan aliohjelmaan. Yllä olevissa esimerkeissä kukin testi on ollut vain yksi rivi, mutta toki yksi testi voi olla useitakin rivejä, joissa aluksi valmistellaan testin parametejä ja sitten kutsutaan aliohjelmaa ja lopuksi katsotaan “testioperaattoreilla” `===` tai `~~~` että kaikki on kuten pitikin. Esimerkiksi `Yhdista`-funktion testejä olisi voitu kirjoittaa useammallekin riville. Alla muutama esimerkki miten testit olisi voitu kirjoittaa laveammin. Kuitenkin koska sama asia saadaan helposti yhdelle riville, on usein nopeampi lukea testejä kun ne on kirjoitettu kompaktimmin.

```
1
2     /// <summary>
3     /// Yhdistää kahden ei-negatiivisen luvun numerot toisiinsa.
4     /// </summary>
5     /// <param name="a">Ensimmäinen luku</param>
6     /// <param name="b">Toinen luku</param>
7     /// <returns>Yhdistetty luku</returns>
8     /// <example>
9     /// <pre name="test">
10    /// int alkuosa = 13;
11    /// int loppuosa = 2;
12    /// int tulos = Yhdista(alkuosa, loppuosa);
13    /// tulos === 132;
14    ///
15    /// alkuosa = 10;
16    /// loppuosa = 0;
17    /// tulos = Yhdista(alkuosa, loppuosa);
```

```
18      /// tulos === 100;
19      /// </pre>
20      /// </example>
21      public static int Yhdista(int a, int b)
22      {
23          string ab = a.ToString() + b;
24          int tulos = int.Parse(ab);
25          return tulos;
26      }
```

Myöhemmin taulukoiden, listojen ja `StringBuilder`-luokan yhteydessä tulee esimerkkejä joissa testikoodia joutuu jakamaa useammalle riville.

ComTestin ajamainen tarkoittaa käytännössä sitä, että kommenteissa oleva testikoodi muodostetaan "tavalliseksi" aliohjelmaksi (NUnit- testimetodeiksi) ja kaikista tiedostossa olevista testeistä tehdään yksi testiluokka (em. esimerkissä `YhdistaTest.cs`) joka sitten ajetaan NUnit-testausympäristön avulla niin, että ympäristö kutsuu jokaista testialiohjelmaa ja ajaa siellä olevan koodin ja mikäli joku ehto ei toteudu, ilmoittaa tästä punaisella. Kun opit hieman lisää, katso mitä `Test`-tiedostot pitävät sisällään.

Katso miltä kääntynyt NUnit-testitiedosto näyttää10pt

```
1 cat YhdistaTest.cs
```

Myös testit täytyy testata. Voihan olla, että kirjoittamissamme testeissä on myös virheitä. Osa tästä testistä tulee tehtyä kun testaa tynkä-aliohjelmaa. Kannattaa myös kokeilla kirjoittaa testeihin virhe tarkoituksella. Tällöin testeistä pitäisi saada tietysti punaista. 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".

Tyypillisiä ääritilanteita voi olla esimerkiksi että taulukon suurin alkio löytyy taulukon alusta, keskeltä tai lopusta. Usein myös yhden alkion taulukko ja tyhjä taulukko (tai merkkijono) ovat testaamisen arvoisia erikoistapauksia. Lisäksi esimerkiksi suurimman paikan etsimisessä voisi olla oleellista testata myös tilanne, jossa on useita suurimman alkion kanssa yhtäsuuria alkioita.

Esimerkkinä olevassa `Yhdista`-aliohjelmassa ääritilanteita ovat lähinnä nollat, kummallakin puolella erikseen ja yhdessä. Muutoin testiarvot on valittu melko sattumanvaraisesti. Lisäksi jossakin vaiheessa olisi syytä lisätä testit ja käsittely sille, mitä tapahtuu negatiivisilla luvuilla.

> Testit eivät todista että aliohjelma toimii! Testeillä voidaan todistaa vain että testitapausten tapauksessa aliohjelma toimii oikein.

Tehtava 11.1

Täydennä TÄHÄN-tekstien tilalle ohjelmaa kuvaavat tiedot. Lisää testirivejä. Yksi testirivi on jo valmiina10pt

```
1
2       /// <summary>
```

```
3       /// TÄHÄN
4       /// </summary>
5       /// <param name="a">TÄHÄN</param>
6       /// <returns>TÄHÄN</returns>
7       /// <example>
8       /// <pre name="test">
9       /// LisaaYksi(0) === 1;
10      ///
11      ///
12      ///
13      ///
14      /// </pre>
15      /// </example>
16      public static int LisaaYksi(int luku)
17      {
18           return luku+1;
19      }
```

11.3 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 toivotun arvon ja todellisen tuloksen välille täytyy sallia pieni virhemarginaali. Tehdään `Keskiarvo`-aliohjelma, joka osaa laskea kahden double-tyyppisen luvun keskiarvon, ja kirjoitetaan sille samalla dokumentaatiokommentit ja ComTest-testit.

Dokumentaation tuottavassa `Doxygen`-ohjelmassa olevan virheen takia pitää testeissä olla parillinen määrä rivejä, joissa `~~~` esiintyy, jotta testattavasta funktiosta syntyy dokumentaatio.

```
1
2       /// <summary>
3       /// Aliohjelma laskee parametreina saamiensa kahden
4       /// double-tyyppisen luvun keskiarvon.
5       /// </summary>
6       /// <param name="a">Ensimmäinen luku</param>
7       /// <param name="b">Toinen luku</param>
8       /// <returns>Lukujen keskiarvo</returns>
9       /// <example>
10      /// <pre name="test">
11      /// Keskiarvo(0.0, 0.0)   ~~~  0.0;
12      /// Keskiarvo(1.2, 0.0)   ~~~  0.6;
13      /// Keskiarvo(0.8, 0.2)   ~~~  0.5;
14      /// Keskiarvo(-0.1, 0.1)  ~~~  0.0;
15      /// Keskiarvo(-1.5, -2.5) ~~~ -2.0;
16      /// Keskiarvo(-1.5, 1.5) ~~~ 0.0;
17      /// </pre>
18      /// </example>
19      public static double Keskiarvo(double a, double b)
20      {
21          return (a + b) / 2.0;
22      }
```

Oletuksena virhemarginaali (vertailun tarkkuus) on 6 desimaalia. Virhemarginaalia voi vaihtaa `#TOLERANCE`-määrityksellä:

Kokeile vaihtaa eri toleransseja millä saat tuloksen oikeaksi tai vääräksi.10pt

```
    /// <example>
    /// <pre name="test">
    /// #TOLERANCE=0.05
    /// Lisaa(0.0, 0.001) ~~~  0.0;
    /// Lisaa(0.0, 0.01)  ~~~  0.0;
    /// </pre>
    /// </example>
    public static double Lisaa(double a, double b)
    {
        return (a + b);
    }
```

Liukulukuja testattaessa täytyy parametrit antaa desimaaliosan kanssa. Esimerkiksi jos yllä olevassa esimerkissä ensimmäinen testi olisi muotoa `Keskiarvo(0, 0) ~~~ 0.0`, niin tällöin kutsuttaisiin funktiota `Keskiarvo(int x, int y)`, *jos* sellainen on olemassa. Jos `int` parametreilla olevaa versiota ei ole, kutsutaan `double` parametreilla olevaa versiota.

Tehtävä 11.2

Lisää aliohjelmalle testit. 10pt

```

    public static double LaskeProsentti(double luku, double prosentti)
    {
         return (prosentti * 0.01)*luku;
    }
```

Tehtava 11.3

Tehtävässä 9.8.5 piti tehdä aliohjelma, jossa lasketaan henkilön ikä. Kopio vastaus tähän ja lisää siihen testit.10pt

Tehtävä 11.4

Selitä seuraavat termit: a) Aliohjelma, b) Muuttuja, c) Funktio, d) Luokka, e) Parametri f) Testit 10pt

Tarkista tietosi

Sain Comtestin toimimaan omalla koneella?

	True	False
Kyllä	☐	☐

Luku 12

Merkkijonot

Merkkijono on tietotyyppi tekstin esittämiselle. Merkkijonoilla on tärkeä rooli ohjelmoinnissa esimerkiksi tekstin tuottamiseen käyttöliittymiä varten, mutta joskus myös tiedon—vaikkapa DNA-ketjun—tallentamiseen. Lisäksi merkkijonoja tarvitaan, kun halutaan lukea käyttäjän sovellukselle syöttämää tekstiä.

Merkkijono on *jono peräkkäisiä merkkejä*. C#:ssa merkkijono on tavallaan `char`-merkeistä koostuva taulukko. Taulukoista on tässä monisteessa oma lukunsa myöhemmin. On kuitenkin tärkeä huomata, että merkkijonot ovat olioita, mikä tarkoittaa, että merkkijonomuuttuja on *viite* olion varsinaiseen sisältöön. Tämän käytännön merkitys huomataan myöhemmin esimerkkien avulla.

Merkkijonot voidaan jakaa *muuttumattomiin* (*immutable*) ja *muokattaviin* (*mutable*). Muuttumaton merkkijono on tyypiltään `string`, ja muokattava merkkijono on tyypiltään `StringBuilder`. Muuttumatonta merkkijonoa ei voi muuttaa luomisen jälkeen. Muokattavaa merkkijonoa sen sijaan voi. Muokattavan merkkijonon käsittely on joissakin tilanteissa mielekkäämpää. Vaikka `string`-olioita ei voikaan muuttaa, pärjäämme sillä monissa tilanteissa.

Video merkkijonoista Luento 6 (8m51s)

12.1 Alustaminen

Merkkijonon voi alustaa kahdella tavalla:

```
1         string henkilo1 = new string(new char[] {'A', 'k', 'u'});
2         string henkilo2 = "Kalle Korhonen";
```

Esimerkin rivillä 2 on käytetty alustamiseen lainausmerkkeihin (", tuplahipsut) kirjoitettua **merkkijonovakiota** (merkkijonoliteraali) `"Kalle Korhonen"`. Huomaa että yksittäistä kirjainta, eli `char`-tietotyyppiä vastaava vakio (kirjainliteraali) kirjoitetaan heittomerkkeihin (', yksinkertaisiin hipsuihin), esimerkiksi (`'A'`).

Jälkimmäinen tapa muistuttaa enemmän alkeistietotyyppien alustamista, mutta silti pitää muistaa, että merkkijonot ovat C#:ssa aina *olioita*. Ja tällöin merkkijonomuuttujat ovat viitteitä olioihin, eli eivät sisällä itse merkkijonoa, vaan viitteen siihen olioon, joka sisältää merkkijonon kirjaimet.

Eli meillä voi esimerkiksi olla kaksi eri merkkijonomuuttujaa (eli viitettä), jotka viittaavat samaan merkkijonoon (olioon).

Toisaalta meillä voi olla eri viitteitä, joiden “päästä” löytyy saman sisältöinen merkkijono(olio).

```
1         string jono1 = "Kissa";
2         string jono2 = jono1;
3         string jono3 = "Koira";
4         string jono4 = new String("Koira");
```

Seuraavassa animaatiossa on vielä näytetty mitä tapahtuu jos `jono1`-merkkijonoa “muutetaan”. Merkkijonohan on muuttumaton ja siksi rivi

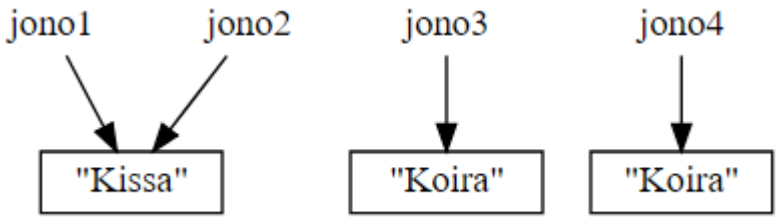
```
jono1 += " istuu";
```

on sama kuin

```
jono1 = "Kissa" + " istuu";
```

jolloin syntyy uusi merkkijono-olio, johon `jono1` käännetään viittaamaan.

Katso animaatiota liikkumalla nuolilla »10pt



Seuraavassa esimerkkiohjelmassa on pakotettu luomaan uusi jono viitemuuttujaa `jono4` varten, koska muuten kääntäjä käyttäisi hyväkseen sitä, että merkkijonot ovat muuttumattomia ja sijoittaisi `jono4`-(viite)muuttujaan saman viitteen kuin on jo sijoitettu muuttujaan `jono3`.

```
1         string jono3 = "Koira";
2         string jono4 = new String("Koira");
3         string jono5 = "Koira"; // kääntäjä käyttää jo kerran luotua Koiraa
4         Console.WriteLine(Object.ReferenceEquals(jono3,jono4)); // False
5         Console.WriteLine(Object.ReferenceEquals(jono3,jono5)); // True
```

Katso animaatiota liikkumalla nuolilla10pt

Viitemuuttuja = muuttuja joka viittaa johonkin olioon. Usein puhutaan silti vain muuttujasta.
Merkkijonoviitemuuttuja = muuttuja, joka viittaa merkkijono-olioon. Usein puhutaan silti vain viitemuuttujasta, muuttujasta tai (harhaanjohtavasti) merkkijonosta.
Merkkijono-olio = jonnekin muistiin luotu olio, joka edustaa merkkijonon sisältöä. Tästäkin saatetaan käyttää nimeä merkkijono.
Merkkijonoliteraali eli **merkkijonovakio** = lainausmerkkien (”) sisään kirjoitettu jono

Merkkijono-olion muuttumattomuuden ansiosta merkkijonojen voidaan monessa kohti ajatella käyttäytyvän kuten tavallisten muuttujien. Esimerkiksi vaikka aliohjelmakutsussa vietäisiin merkkijono parametrina, niin aliohjelmasta paluun jälkeen sillä on silti varmasti alkuperäinen

arvo. Tämän ansiosta merkkijonoja käytettäessä ei ole välttämätöntä ajatella niitä koko ajan viitteinä ja olioina. Muuttuvien merkkijonojen (`StringBuilder`) kohdalla tämä ei pidä paikkaansa.

Esimerkiksi jos kokonaislukumuuttujan on sijoitettu:

```
int luku = 5;
```

sanotaan, että muuttujan `luku` arvo on `5`. Jos merkkijonomuuttujaan on sijoitettu

```
string jono = "Kissa";
```

pitäisi tarkkaan ottaen sanoa, että muuttujan `jono` arvo on viite olioon, jossa on kirjaimet `Kissa`. Silti puhekielessä saatetaan (hieman väärin) sanoa, että merkkijonon arvo on `Kissa`.

12.1.1 Merkkijono on kuin taulukko

Koska merkkijono on kuin taulukko, voidaan siitä ottaa yksi merkki kuten taulukon alkiosta:

```
1  string henkilo = "Kalle Korhonen";
2  char eka = henkilo[0];    // K
3  char viides = henkilo[4]; // e koska indeksit alkavat 0:sta
4  int pituus = henkilo.Length;
```

Katso animaatiota liikkumalla nuolilla10pt

On varottava viittamasta indeksiin jota taulukossa, eli tässä tapauksessa merkkijonossa, ei ole. Esimerkiksi:

```
1  string henkilo = "Kalle Korhonen";
2  char huti = henkilo[40]; // poikkeus koska ei ole merkkiä paikassa 40
```

Tällaisen ohjelman ajo päättyisi poikkeukseen:

```
Unhandled Exception:
System.IndexOutOfRangeException: Array index is out of range.
  at Pohja3.Main () [0x00000] in <filename unknown>:0
[ERROR] FATAL UNHANDLED EXCEPTION:
  System.IndexOutOfRangeException: Array index is out of range.
  at Pohja3.Main () [0x00000] in <filename unknown>:0
```

12.1.2 Null-viittaus ja tyhjä merkkijono

Koska merkkijono on olio, on sitä vastaava muuttuja viite ja viite voi olla niin sanottu `null`-viite. Toisaalta merkkijono voi olla sellainen, jossa ei ole yhtään merkkiä. Huomaa myös, että kumpikin edellä mainituista on eri asia kuin merkkijono, joka sisältää näkymättömiä merkkejä (white space), esimerkiksi välilyöntejä.

Tyhjä merkkijono on usein ihan hyödyllinen, mutta sen kanssa pitää myös muistaa olla varovainen kun siinä ei ole yhtään merkkiä, ei edes ensimmäistä.

```
1  string eiViitetta = null;
2  string tyhja = "";
```

```
3         string valilyonti = " ";
4
5         int tyhjanPituus = tyhja.Length;              // 0
6         int valilyonninPituus = valilyonti.Length;    // 1
7         char vali = valilyonti[0];                    // ' '
8         // char eka = tyhja[0];       // kaatuisi Array index is out of range.
9         // int nullPituus = eiViitetta.Length; // kaatuisi NullReferenceException
```

Jonon tyhjyyttä voidaan testata vertaamalla sen pituutta. Tässä on kuitenkin se riski, että jos itse viite on `null`, niin ohjelma kaatuu `NullReferenceException`-poikkeukseen. Jos olemme muusta edeltävästä koodista varmoja siitä, että viite ei voi olla `null`, niin silloin pituuden testaaminen on ok. Muuten joudumme ensin testaamaan ettei viite ole `null`. Tämä voidaan tehdä joko `String`-luokan valmiilla staattisella funktiolla `IsNullOrEmpty` tai yhdistetyllä ehtolausella.

```
1         string eiViitetta = null;
2         string tyhja = "";
3         string nimi = "Matti";
4
5         if ( nimi.Length > 0 ) Console.WriteLine("Nimi ok");   // tulostuu
6         if ( tyhja.Length > 0 ) Console.WriteLine("Tyhja ok"); // ei tulostu
7         if ( tyhja != null ) Console.WriteLine("Tyhja ei null"); // tulostuu
8         if ( eiViitetta == null ) Console.WriteLine("on null"); // tulostuu
9         if ( String.IsNullOrEmpty(eiViitetta) ) Console.WriteLine("on null");
10        if ( !String.IsNullOrEmpty(tyhja) ) Console.WriteLine("ei tulostu");
11        if ( !String.IsNullOrEmpty(nimi) ) Console.WriteLine("Nimi ok");
12        if ( nimi != null && nimi.Length > 0 ) Console.WriteLine("Nimi ok");
```

12.2 Hyödyllisiä metodeja ja ominaisuuksia

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

12.2.1 Metodit palauttavat uuden jonon

Huomaa että ne `String`-luokan metodit, jotka palauttavat merkkijonon, luovat **uuden** merkkijonon, eli palauttavat siis viitteen tähän uuteen olioon.

Esimerkiksi `ToLower()` palauttaa viitteen uuteen merkkijonon, jossa kaikki kirjaimet on muutettu pieniksi kirjaimiksi. Tässä, kuten muissakaan vastaavissa metodeissa, alkuperäinen jono ei muutu lainkaan. Metodi luo *uuden olion*, jossa on samat merkit kuin alkuperäisessä jonossa, mutta pieninä kirjaimina. Sitten palautetaan *viite* tähän uuteen jonoon. Alkuperäinen jono säilyy muuttumattomana (*immutable*).

```
1 //
2         string k = "Kissa";
3         string kissaPienena = k.ToLower(); // syntyy uusi jono
4         Console.WriteLine(kissaPienena); // tulostaa "kissa"
```

Katso animaatiota liikkumalla nuolilla10pt

Yllä olevan esimerkin `k`-viitemuuttujan arvo voidaan toki myös korvata sijoituksen yhteydessä.

Tällöin alkuperäinen olio, joka sisältää merkkijonon “Kissa” muuttuu roskaksi (ellei siihen osoita jokin muu viitemuuttuja).

```
1 //
2         string k = "Kissa";
3         k = k.ToLower(); // tässäkin syntyy uusi jono
4         Console.WriteLine(k); // tulostaa myöskin "kissa"
```

Katso animaatiota liikkumalla nuolilla10pt

Apumuuttujaan `k` sijoittamista ei kuitenkaan välttämättä tarvita, sillä `ToLower`-metodin kutsumisen seurauksena syntynyttä oliota voi käyttää osana lauseketta, esimerkiksi `WriteLine`-metodin argumenttina. Huomaa kuitenkin, että tässä alla olevassa esimerkkitapauksessa `k`-muuttuja viittaa edelleen merkkijonoon “Kissa” (isolla alkukirjaimella).

Edelleen, olio, jolle muunnos tehdään, ei tarvitse välttämättä omaa muuttujaa; alla on tästä esimerkki merkkijonolle “Koira”.

```
1 //
2         string k = "Kissa";
3         Console.WriteLine(k.ToLower()); // tulostaa edelleen "kissa"
4         Console.WriteLine(k);           // tulostaa "Kissa"
5         Console.WriteLine("KOIRA".ToLower()); // tulostaa "koira"
```

Koska alkuperäinen jono säilyy muuttumattomana, niin pelkkä kutsu ilman sijoitusta ei ole mielekäs. Esimerkki alla.

```
1 //
2         string k = "Kissa";
3         k.ToLower(); // olio johon k viittaa säilyy muuttumattomana.
4                      // ToLower()-metodin palauttamaa tulosta (siis uutta oliota)
5                      // ei käytetä missään!
6         Console.WriteLine(k); // tulostaa "Kissa"
```

Alla olevassa animaatiossa havainnollistetaan miten esimerkiksi `ToLower`-metodissa syntyy uusi jono:

Animaatio: Tutki ToLower-toimintaa

Askella ToLower esimerkki vihreällä nuolella Tutki ToLower toimintaa

12.2.2 Merkkijonometodeja

Alla tärkeimpiä `String`-luokan metodeja. Lisää löydät sivulta:

- https://learn.microsoft.com/en-us/dotnet/api/system.string#methods

12.2.2.1 Equals

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

```
1         string etunimi = "Aku";
2         if (etunimi.Equals("Aku")) Console.WriteLine("On sama sisältö!");
3         else  Console.WriteLine("Ei ole sama sisältö!");
```

Poikkeuksellisesti yhtäsuuruutta voidaan testata `String`-olioiden tapauksessa myös vertailuoperaattorilla `==`. Pitää muistaa ettei tämä toimi muiden olioiden kanssa!

```
1 //
2         string etunimi = "Aku";
3         if (etunimi == "Aku") Console.WriteLine("On sama sisältö!");
4         else  Console.WriteLine("Ei ole sama sisältö!");
```

Kokeile edellä vaihtaa muuttujaan etunimi eri tavalla kirjoitettuja nimiä, esimerkiksi `"aku"`, `"Aku Ankka"` jne.

12.2.2.2 Compare

- Compare(String, String, Boolean) Vertaa kahden merkkijonon keskinäistä aakkosjärjestystä. Jos merkkijonot ovat samat, funktio palauttaa arvon `0`. Jos ensimmäinen merkkijono on aakkosjärjestyksessä ennen toista (esimerkiksi “kahvi” on aakkosissa ennen sanaa “kasvi”), palautetaan nollaa pienempi arvo. Vastaavasti jos ensimmäinen jono on aakkosjärjestyksessä toisen jälkeen, palautetaan nollaa suurempi arvo. Kirjainkoon merkitsevyys voidaan asettaa kolmannella parametrilla (`true` = kirjainkoolla ei merkitystä tai `false` = kirjainkoolla on merkitystä, `false` on oletus).

  Tässä voidaan ajatella että jos jonot olisivat numeroita, esim `3` ja `5` niin `compare` palauttaa niiden erotuksen (`3-5<0` , `3-3 = 0`, `5-3>0`).

```
1         string s1 = "jAnNe"; string s2 = "JANNE";
2         if (String.Compare(s1, s2, true) == 0)
3             Console.WriteLine("Samat tai melkein samat!");
4         else
5             Console.WriteLine("Erilaiset!");
6         if ( String.Compare("kahvi", "kasvi") < 0 )
7             Console.WriteLine("Kahvi on ensin!");
```

12.2.2.3 Contains

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

```
1         string henkilo = "Ville Virtanen";
2         string haettava = "irta";
3         if (henkilo.Contains(haettava))
4             Console.WriteLine(haettava + " löytyi!");
5         else
6             Console.WriteLine("Ei löydy!");
```

12.2.2.4 IndexOf

- IndexOf(char) Palauttaa annetun merkin ensimmäisen esiintymän sijainnin (indeksin) merkkijonossa. Palauttaa -1, jos merkkiä ei löydy merkkijonosta. Metodista on myös useita

muita versioita, esimerkiksi sellainen, missä etsiminen aloitetaan tietystä paikasta nollan sijaan, ks. dokumentaatio.

Huomaa että jos sinun pitää tietää onko jonossa osajonoa ja saada sen indeksi, ei kannata ensin kutsua `Contains`, koska jos osajonoa ei ole, niin `IndexOf` palauttaa -1 ja siitä tietää ettei osajonoa ollut.

```
1         string henkilo = "Ville Virtanen";
2         int epaikka = henkilo.IndexOf('e'); // etsitään missä on e-kirjain
3         Console.WriteLine(epaikka); // 4
4         int eioo = henkilo.IndexOf('x'); // etsitään x-kirjainta
5         Console.WriteLine(eioo); // -1
6         int toinenepaikka = henkilo.IndexOf('e',5); // aloitetaan paikasta 5
7         Console.WriteLine(toinenepaikka); // 12
```

12.2.2.5 Substring

- Substring(Int32) Luo uuden merkkijonon, jossa on alkuperäisen jonon merkit alkaen parametrinaan olevasta indeksistä. Palauttaa viitteen uuteen jonoon.

```
1         string henkilo = "Ville Virtanen";
2         string sukunimi = henkilo.Substring(6);
3         Console.WriteLine(sukunimi); // Virtanen
```

- Substring(Int32, Int32) Palauttaa viitteen uuteen jonoon, jossa on osa merkkijonosta parametrien mukaan. Ensimmäinen parametri on uuden merkkijonon ensimmäisen merkin indeksi ja toinen parametri palautettavien merkkien määrä. Huomaa, että Javassa toinen parametri on indeksi, jota ei enää oteta mukaan. Jos aloitetaan paikasta 0, nämä ovat sama asia, muuten ei.

```
1         string henkilo = "Ville Virtanen";
2         string etunimi = henkilo.Substring(0,5); // Huom! Luo uuden merkkijonon
3         Console.WriteLine(etunimi); // Ville
```

Katso animaatiota liikkumalla nuolilla10pt

`IndexOf` ja `Substring`-metodit yhdessä soveltuvat joskus hyvin merkkijonon pilkkomiseen ja tietyn palasen ottamiseen. Toisissa tapauksissa taas `Split`-metodi on kätevämpi tähän; tästä lisää luvussa Merkkijonojen pilkkominen ja muokkaaminen.

12.2.2.6 ToLower

- ToLower() Palauttaa viitteen uuteen merkkijonon niin, että kaikki kirjaimet on muutettu pieniksi kirjaimiksi. Huomaa että tässä, kuten ei muissakaan vastaavissa funktioissa, alkuperäinen jono muutu lainkaan.

```
1         string henkilo = "Ville Virtanen";
2         Console.WriteLine(henkilo.ToLower()); // "ville virtanen"
3         Console.WriteLine(henkilo);           // "Ville Virtanen"
```

12.2.2.7 ToUpper

- ToUpper() Luo ja palauttaa viitteen uuteen merkkijonoon, jossa kaikki kirjaimet on muutettu suuraakkosiksi.

```
1         string henkilo = "Ville Virtanen";
2         Console.WriteLine(henkilo.ToUpper()); // "VILLE VIRTANEN"
```

12.2.2.8 Replace

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

```
1 //
2         string sana = "katti";
3         sana = sana.Replace('t', 's');
4         Console.WriteLine(sana);  //tulostaa "kassi"
```

Katso animaatiota liikkumalla nuolilla10pt

- Replace(String, String) Palauttaa viitteen uuteen merkkijonoon, jossa on korvattu merkkijonon kaikki merkkijonoesiintymät toisella merkkijonolla. Ensimmäisenä parametrina korvattava merkkijono ja toisena korvaaja. Huomattavaa on, että itse jono ei muutu, vaan palautetaan viite uuteen jonoon, johon muutos on tehty. Alla olevassa esimerkissä viitteeseen `sana` sijoitetaan tämän uuden jonon viite. Alkuperäinen jono (johon kukaan ei enää viittaa), on muuttumaton.

```
1         string sana = "katti kattinen";
2         sana = sana.Replace("atti", "issa");
3         Console.WriteLine(sana);  // "kissa kissanen"
```

Katso animaatiota liikkumalla nuolilla10pt

```
1         string sana = "katti kattinen";
2         string muutettusana = sana.Replace("atti", "issa");
3         Console.WriteLine(sana);  // "katti kattinen"
4         Console.WriteLine(muutettusana);  // "kissa kissanen"
```

Katso animaatiota liikkumalla nuolilla10pt

12.2.2.9 Lisäksi

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

```
        string henkilo = "Ville";
        int henkilonNimenPituus = henkilo.Length;
        Console.WriteLine(henkilonNimenPituus); //tulostaa 5
```

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

```
        string henkilo = "Seppo Sirkuttaja";
        char kolmasKirjain;
        int i = 2;
        kolmasKirjain = henkilo[i];  // indeksit menevät 0,1,2,3 jne...
        Console.WriteLine(henkilo + " -nimen paikassa " + i +
                          " oleva merkki on " + kolmasKirjain);
```

Merkkijonojen indeksointi alkaa nollasta! Merkkijonon ensimmäinen merkki on siis indeksissä 0. Viimeinen indeksi on `Length-1`.

Kysymyksiä merkkijonoista

Mitkä seuraavista kommenteista pitää paikkaansa:

	True	False
‘string‘ ja ‘char‘ ovat oliotyyppejä.	☐	☐
string nimi = Kaija; On oikein alustettu.	☐	☐
‘string‘-olion arvoa ei voi muuttaa.	☐	☐
Merkkijonojen vertailuun on käytettävä **aina** ‘Equals‘-metodia.	☐	☐
Merkkijonon viimeinen indeksi on vähemmän kuin ‘Length‘-ominaisuuden sisältämän muuttujan arvo.	☐	☐

Tehtävä 12.1

Tässä ohjelmassa kysytään käyttäjältä syöte ja tulostetaan se sitten neljä kertaa tervehdyksen kanssa. Muuta ohjelmaa niin, että yhdessä tuloksessa syöte on kokonaan pienellä, toisessa kokonaan isolla, kolmannessa se on korjannut kaikki a-kirjaimet x-kirjaimella ja viimeisessä ensimmäinen ja viimeinen kirjain on jätetty pois. Mitä tapahtuu jos käyttäjä antaa tyhjän syötteen?10pt

```
using System;

///@author
///@version
///
/// <summary>
```

```
7  /// Harjoitellaan merkkijonoja
8  /// </summary>
9  public class NimenTulostus
10 {
11     /// <summary>
12     /// Pyydetään käyttäjältä syöte ja tulostellaan.
13     /// </summary>
14     public static void Main()
15     {
16         String nimi;
17
18         Console.Write("Anna nimi > ");
19         nimi = Console.ReadLine();
20         Console.WriteLine();
21         Console.WriteLine("Hei, " + nimi + "!"); // vaihda nimi -> nimi.ToLower()
22         Console.WriteLine("Hei, " + nimi + "!");
23         Console.WriteLine("Hei, " + nimi + "!");
24         Console.WriteLine("Hei, " + nimi + "!");
25     }
26 }
```

12.2.3 Harjoitus 12.2

Osaisitko tehdä tehtävän, jossa käyttäjältä kysytään nimi (Etunimi Sukunimi) ja sen jälkeen tulostetaan annetun nimen nimikirjaimet? Esimerkiksi jos nimi olisi Maija Mehiläinen, tulostettaisiin M.M. Katso ohjelman tekeminen luennolta ja täydennä koko ohjelma alla olevaan ohjelmakenttään. Täydennä myös testit.

Nimikirjaimet syötteestä -luento Luento 7 (1h5m1s)

Tehtävä 12.2

Täydennä ohjelma luennon mukaisesti. Muista dokumentaatio ja testit.10pt

12.3 Huomautus

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;` Tapana on, että muuttujan tyypiksi esitellään `string`. Jos viitataan luokan metodiin (staattiseen aliohjelmaan), niin sitä kutsutaan isolla kirjaimella alkavalla muodolla `String.AliohjelmanNimi`.

12.4 Muokattavat merkkijonot: StringBuilder

Tässä luvussa esitellään vain tärkeimpiä `StringBuilder`-luokan metodeja. Täydellisen luettelon metodeista löydät dokumentista:

- https://learn.microsoft.com/en-us/dotnet/api/system.text.stringbuilder#methods

Niin sanottujen muuttumattomien (*immutable*) 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.

StringBuilder vs String Luento 9a (6m56s) 12.3 Muokattavat merkkijonot:Stringbuilder

C#-kielessä (kuten Javassakin) 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.

```
1         StringBuilder nimi = new StringBuilder(); // "" (tyhjä)
2         nimi.Append("Kustaa"); // "Kustaa"
3         nimi.Append(" ");      // "Kustaa "
4         nimi.Append("Aadolf"); // "Kustaa Aadolf"
```

Katso animaatiota liikkumalla nuolilla10pt

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 kaikkia samoja tietotyyppejä kuin Append-metodillakin. Voisimme esimerkiksi lisätä edelliseen esimerkkiin järjestysnumeron `VI`. Sitä ennen tarkastellaan merkkien järjestystä ja indeksointia ja kirjoitetaan kunkin tulostuvan merkin yläpuolelle sen paikkaindeksi.

```
01234567890123456789
|----+----|----+----|
Kustaa Aadolf
```

Tästä huomaamme, että indeksi, johon haluamme `VI`:n lisätä, on `7`.

```
1         StringBuilder nimi = new StringBuilder("Kustaa Aadolf");
2         nimi.Insert(7,"VI "); // "Kustaa VI Aadolf"
```

Katso animaatiota liikkumalla nuolilla10pt

Huomaa, että `Insert`-metodi ei korvaa indeksissä 7 olevaa merkkiä, vaan lisää merkkijonoon kokonaan uuden merkin/merkkejä, jolloin merkkijonon pituus kasvaa siis lisättävän jonon pituudella. Korvaamiseen on olemassa oma metodi, `Replace`. Yksittäisen kirjaimen voi vaihtaa suoraan myös `nimi[7] = 'I';`

```
1         StringBuilder nimi = new StringBuilder("Kustaa VI Aadolf");
2         nimi[7] = 'I';  // Kustaa II Aadolf
```

Katso animaatiota liikkumalla nuolilla10pt

12.4.1 Muita StringBuilder-luokan hyödyllisiä metodeja

- Remove(Int32, Int32). Poistaa merkkijonosta merkkejä siten, että ensimmäinen parametri on aloitusindeksi, ja toinen parametri on poistettavien merkkien määrä.

```
1         StringBuilder nimi = new StringBuilder("Kustaa VI Aadolf");
2         nimi.Remove(7,3);  // Kustaa Aadolf
```

Katso animaatiota liikkumalla nuolilla10pt

- 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.

Huomaa, että `StringBuilder`-luokan olioita ei voi verrata yhtäsuuruusoperaattorilla `==`, vaan veratailu pitää tehdä `equals`-metodilla. Samoin erittäin tarkkana pitää olla verrattaessa `StringBuider` ja `String` -luokan olioita:

```
1         StringBuilder sb1 = new StringBuilder("Aku");
2         StringBuilder sb2 = new StringBuilder("Aku");
3         string s1 = "Aku";
4
5         if ( sb1 == sb2 ) Console.WriteLine("Tämä ei tulostu");
6         if ( sb1.Equals(sb2) ) Console.WriteLine("sb1.Equals(sb2)");
7         // if ( s1 == sb2 ) Console.WriteLine("Tästä käännösvirhe");
8         if ( s1.Equals(sb2) ) Console.WriteLine("Tämä ei tulostu");
9         if ( sb1.Equals(s1) ) Console.WriteLine("sb1.Equals(s1)");
10        if ( sb2.Equals(s1) ) Console.WriteLine("sb2.Equals(s1)");
11        if ( s1 == sb2.ToString() ) Console.WriteLine("s1==sb2.ToString()");
```

Katso animaatiota liikkumalla nuolilla10pt

12.4.2 StringBuildereiden testaaminen

StringBuilderiä ei voi suoraan verrata merkkijonoon, vaan se pitää ensin muuttaa merkkijonoksi.

```
1
2     /// <summary>
3     /// Lisää sanan merkkijonon alkuun tai loppuun niin, että
4     /// jos sana on aakkosissa ennen jonossa olevaa jonoa, niin alkuun, muuten ↩
      loppuun.
5     /// </summary>
```

```
6     /// <param name="jono">jono johon lisätään</param>
7     /// <param name="sana">lisättävä sana</param>
8     /// <example>
9     /// <pre name="test">
10    ///    StringBuilder jono = new StringBuilder("koti");
11    ///    Lisaa(jono, "kissa");
12    ///    jono =S= "kissakoti";
13    ///    Lisaa(jono, "korjaamo");
14    ///    jono =S= "kissakotikorjaamo";
15    ///  </pre>
16    /// </example>
17    public static void Lisaa(StringBuilder jono, string sana)
18    {
19        if ( jono.ToString().CompareTo(sana) > 0 ) jono.Insert(0,sana);
20        else jono.Append(sana);
21    }
```

Yllä ComTestin vertailuoperaattori `=S=` tekee sen, että ennen vertailua molemmat verrattavat lausekkeet muutetaan merkkijonoksi. Eli ilman tuota vertailu pitäisi tehdä:

```
jono.ToString() === "kissakotikorjaamo";
```

ja `=S=` tuottaa:

```
jono.ToString() === "kissakotikorjaamo".ToString();
```

ja koska oikea puoli on jo `string`, niin se ei haittaa vaikka se muutetaan merkkijonoksi.

Vastaavasti jos olisi funktio, joka palauttaa `StringBuilder`-tyyppisen olion, pitäisi ComTestissä muuttaa funktion palauttama tulos ensin merkkijonoksi:

```
///  LuoJono("a",4) =S= "aaaa";
tai
///  LuoJono("a",4).ToString() === "aaaa";
```

Vaikka aikaisemmin sanottiinkin, että funktion kutsumisessa ilman että sen arvoa sijoitetaan mihinkään, ei ole yleensä järkeä, voi em. esimerkin kaltaisissa tapauksissa asia olla toisin. Koska C#:issa saa kutsua funktiota sijoittamatta tulosta mihinkään, voidaan em. funktio tehdä `StringBuilder`-tyyppiseksi ilman, että olemassa olevaa koodia "rikotaan". Aliohjelman muuttaminen funktioksi antaa tässä sen mahdollisuuden, että kutsuja on lyhyempi ketjuttaa ja esimerkiksi testit lyhentyvät.

```
1     /// <summary>
2     /// Lisää sanan merkkijonon alkuun tai loppuun niin, että
3     /// jos sana on akkosissa ennen jonossa olevaa jonoa, niin alkuun, muuten ↩
      loppuun.
4     /// </summary>
5     /// <param name="jono">jono johon lisätään</param>
6     /// <param name="sana">lisättävä sana</param>
7     /// <example>
8     /// <pre name="test">
9     ///    StringBuilder jono = new StringBuilder("koti");
10    ///    Lisaa(jono, "kissa").ToString() === "kissakoti";
11    ///    Lisaa(jono, "korjaamo").ToString() === "kissakotikorjaamo";
12    ///  </pre>
13    /// </example>
```

```
14      public static StringBuilder Lisaa(StringBuilder jono, string sana)
15      {
16          if ( jono.ToString().CompareTo(sana) > 0 ) jono.Insert(0,sana);
17          else jono.Append(sana);
18          return jono;
19      }
```

Testejä, joissa joudutaan testattava muuttamaan merkkijonoksi `ToString`-metodilla, voidaan myös lyhentää tyyliin (kokeille em esimerkkeihin):

```
///     jono =S= "kissakoti";

///     Lisaa(jono, "kissa") =S= "kissakoti";
///     Lisaa(jono, "korjaamo") =S= "kissakotikorjaamo";
```

Tuo ComTestin vertailuoperaattori `=S=` tekee sen, että se muuttaa vertailun molemmat puolet ensin `ToString`-metodilla merkkijonoiksi ja suorittaa sitten vertailun.

Lue lisää ComTestin dokumentista.

12.4.3 Kutsujen ketjuttaminen

Viimeisimmän esimerkin tapa on hyvin yleinen mm. `StringBuilder`-luokan metodeissa.

> StringBuilder dokumentaatio

Vaikka metodit muuttavat itse jonoa, ne palauttavat silti viitteen muutettuun olioon (joka on siis sama viite kuin alkuperäinenkin). Tämän ansiosta kutsuja voidaan ketjuttaa tyyliin:

```
1       public static void Main()
2       {
3           StringBuilder jono = new StringBuilder("luku");
4           StringBuilder alkuluku = jono.Insert(0, "alku");
5           StringBuilder altaulukko = alkuluku.Append("taulukko");
6           System.Console.WriteLine(altaulukko);
7           // Huom!  edellä kaikki kolme jonoa viittaavat samaan olioon
8
9           // Nyt koska on sijoitus alkuluku = jono.Insert(0, "alku");
10          // voidaan seuraava rivi kirjoittaa laittamalla alkuluvun
11          // tilalle vastaava lauseke, eli
12          // altaulukko = jono.Insert(0, "alku").Append("taulukko")
13          // ja koska altaulukko on WriteLinen sisällä, voidaan
14          // tämä lauseke sijoittaa sen tilalle ja saadaan lopulta
15          // koko hommalle lyhyempi muoto:
16
17          StringBuilder jono2 = new StringBuilder("luku");
18          System.Console.WriteLine(jono2.Insert(0, "alku").Append("taulukko"));
19      }
```

12.5 string-viite vs. StringBuilder-viite

String vs. StringBuilder -viite

Kokeile ja mieti miksi toisessa muuttuu ja toisessa ei. Katso alla oleva

```
1         string jono1 = "luku";
2         StringBuilder jono2 = new StringBuilder("luku");
3
4         string viite1 = jono1;
5         StringBuilder viite2 = jono2;
6
7         viite1 = "alku" + jono1;  // tai sama tulos jos viite1 = "alku" + viite1;
8         viite2.Insert(0, "alku");
9
10        Console.WriteLine($"{jono1} {jono2}");
11        // Tulostaa luku alkuluku
```

Tutki animaation nuolella > miksi toinen jono muuttuu ja

Tehtävä: Piirrä kuva toisesta tilanteesta

Piirrä kuva matkien alla olevaa ”piirtokoodia” (ei C#) tilanteessa,

```
1 new *$1 luku
2 ref jono1 -> $1
3 new *$2 luku
4 ref jono2 -> $2
5 ref viite1 -> $1
6 ref viite2 -> $2
7 new *$3 alkuluku
8 viite1 -> $3
9 $2 = alkuluku
```

String vs. StringBuilder -viite aliohjelmakutussa

Edellinen esimerkki ei sellaisenaan ole kovin järkevä, mutta se on juuri

```
1     public static void Main()
2     {
3         string jono1 = "luku";
4         StringBuilder jono2 = new StringBuilder("luku");
5
6         LisaaAlku(jono1, jono2);
7
8         Console.WriteLine($"{jono1} {jono2}");
9         // Tulostaa luku alkuluku
10    }
11
12
13    public static void LisaaAlku(string viite1, StringBuilder viite2)
14    {
15        viite1 = "alku" + viite1;
```

```
16          viite2.Insert(0, "alku");
17      }
```

Tutki animaation nuolella > miksi toinen jono muuttuu ja

12.6 Huomautus: aritmeettinen + vs. merkkijonoja yhdistelevä +

Merkkijonoihin voidaan "+"-merkkiä käyttämällä yhdistellä myös numeeristen muuttujien arvoja. Tällöin ero siinä, toimiiko "+"-merkki aritmeettisena operaattorina vai merkkijonoja yhdistelevänä operaattorina, on todella pieni. Tutki ja kokeile alla olevalla esimerkkillä.

```
1           int luku1 = 2;
2           int luku2 = 5;
3
4           //tässä "+"-merkki toimii aritmeettisena operaattorina
5           Console.WriteLine(luku1 + luku2); //tulostaa 7
6
7           //tässä "+"-merkki toimii merkkijonoja yhdistelevänä
8           Console.WriteLine(luku1 + "" + luku2);  //tulostaa 25
9
10          //Tässä ensimmäinen "+"-merkki toimii aritmeettisena
11          //ja toinen merkkijonoja yhdistelevänä operaattorina
12          Console.WriteLine(luku1 + luku2 + "" + luku1); //tulostaa 72
```

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

Animaatio: Suorita merkkijonoja yhdistävä ohjelma

Askella ohjelmaa vihreällä nuolella. Tutki merkkijonojen yhdistämistä.

12.7 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.

```
1           int luku = 23;
2           bool totuusarvo = false;
3
4           String merkkijono1 = "" + luku;
5           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.

12.8 Reaalilukujen muotoilu String.Format-metodilla

`String`-luokan `Format`-metodi tarjoaa monipuoliset muotoilumahdollisuudet useille tietotyypeille, mutta katsotaan tässä erityisesti 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:

```
1         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.

Aaltosuluissa oleva `0` tarkoittaa että muotoilujonon jälkeisistä parametreista ensimmäinen (indeksissä 0) tulee siihen kohti. Myöhemmin on esimerkkejä missä muotoilujonon jälkeen on useita parametreja. Muotoilujonossa voi olla mitä tahansa tekstiä ja muut kuin aaltosuluissa olevat tekstit tulevat syntävään merkkijonoon normaaliin C# tapaan.

Aliohjelmasta `WriteLine` on olemassa muoto, joka käyttää samaa parametrilistaa, eli silloin ensimmäinen parametri on muotoilujono ja loput tulostettavia olioita tai lausekkeita.

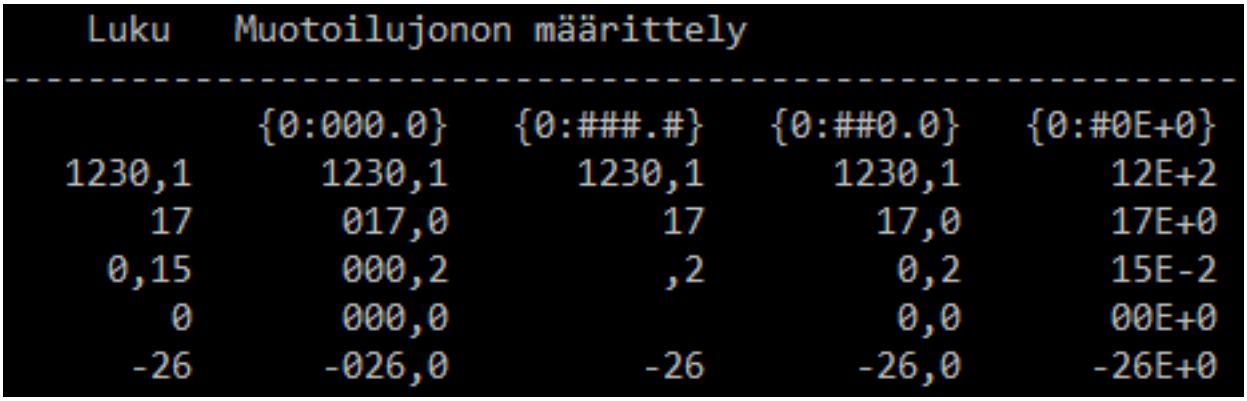
```
1         int a = 5;
2         int b = 9;
3         Console.WriteLine("Lasku {0} + {1} = {2} on aika helppo.", a, b, a+b);
4         // Tulostaa: Lasku 5 + 9 = 14 on aika helppo.
```

Uudemman C#-kääntäjän mukana on tullut ominaisuus, jossa merkkijonoliteraalista (vakiosta) voidaan `$`-merkillä tehdä muotoilujono, jossa aaltosuluissa voi olla mitä tahansa muotoiltavia lausekkeita. Microsoft käyttää tästä nimeä *String Interpolation*. Esimerkiksi edellinen esimerkki tällä muotoilujonolla olisi:

```
1         string pii = $"{Math.PI:#.##}"; // pii = "3.14"
```

Seuraavana esimerkkejä erilaisista muotoilumääreistä. Samat määreet toimivat myös dollarilla alkavan muotoilujonon määreinä.

```
1         Console.WriteLine("123456789012345");
2         Console.WriteLine(String.Format("{0, 11}", 1230.123));
3         Console.WriteLine(String.Format("{0, -11}", 1230.123)); // vasen reuna
4         Console.WriteLine(String.Format("{0, 11:000.0}", 1230.12));
5         Console.WriteLine(String.Format("{0, 11:###.##}", 1230.12));
6         Console.WriteLine(String.Format("{0, 11:##0.00}", 1230.1));
7         Console.WriteLine(String.Format("{0, 11:#0E+0}", 1230.123));
```



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://gitlab.jyu.fi/tie/ohj1/luentomonistecs/-/blob/master/esimerkit/StringFormat.cs

Esimerkkisarakkeista kolmas, `{0,11:##0.0}` (otsikkoriviltä puuttuu tuo ,11 joka määrää kentän viemän minimitilan), on esimerkki siitä, miten erilaisia lukuja saadaan järjestettyä siististi myös päällekkäin desimaalipisteen (tai -pilkun) kohdalta. Ensimmäinen `0` tarkoittaa että parametrilistan indeksissä 0 oleva arvo tulostuu tähän. `,11` tarkoittaa että ko. tulostuspaikan tulee olla vähintään 11 merkkiä leveä. Kaksoispiste (`:`) aloittaa tarkemman tulostusohjeen. Risuaita (`#`) tarkoittaa, että jos luvussa ei ole sen merkin kohdalla numeroa, se jätetään pois paitsi `E`-muotoilussa. Sarakkeissa 3 ja 4 pisteen etupuolella olevalla `#`-merkillä ei ole vaikutusta tulokseen. 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 desimaalierottimen jälkeen olisi mielekästä olla kaksi nollaa, jolloin “nollasentit” näytetään joka tapauksessa, myös rahamäärän ollessa tasasumma.

Huomaa, että ylläolevissa esimerkikuvassa on järjestelmän desimaalierottimena ollut pilkku. Tämä on järjestelmäkohtaista ja muutettavissa esimerkiksi Windows 7:ssa

```
Control panel/Region and language/Formats/Additional settings/Decimal symbol
```

Muotoilumerkkijono laitetaan lainausmerkkeihin ja jokaista muotoiltavaa parametria varten sitä vastaava indeksinumero aaltosulkujen sisään ensimmäiseksi. Muotoilujonossa voi siis olla muotoiluohjeet useille paramtereille kerralla. Tästä esimerkki alla

- paikkaan 0 tuostuu indeksissä 0 oleva parametri (esimerkissä arvoltaan 1),
- seuraavaan paikkaan tulostuu parametrilistan indeksissä 2 oleva arvo, eli 3
- seuraavaan paikkaan tulostuu parametrilistan indeksissä 1 oleva arvo, eli 2
- viimeiseen paikkaan tulostuu uudelleen parametrilistan indeksissä 0 oleva arvo, eli 1

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

Kullekin tulostettavalle kohdalla voi kirjoittaa aikaisempien esimerkkien tapaan tarkempia muotoiluohjeita. Seuraavassa esimerkissä on annettu kullekin tulostettavalle luvulle minimileveys johon se tulostuu. Lisäksi toiseen paikkaan on haluttu tulostaa indeksin 2 mukaisessa kohdassa oleva parametri kuuden merkin kokoiseen tilaan niin, että siihen tulee kolme kpl nollia jos luku ei muuten ole vähintään kolmen numeron kokoinen. Eli `{2,6:000}` tarkoittaa että arvo 3 (on siis paikassa 2 parametrilistassa) tulostuu (tyhjiä paikkoja on merkitty alleviivoilla) `"___003"`. Mikäli muotoilu olisi `{2,-6:000}`, tulostuisi `"003___"`. Vastaavasti jos paikassa 2 oleva arvo olisi

vaikkapa 2017 tulostuisi alkuperäisellä muotoiluohjeella "__2017 ja jälkimmäisellä "2017__.

```
        String luvut = String.Format("{0,4} {2,6:000} {1,2} {0,3}", 1, 2, 3);
        Console.WriteLine(luvut); // Tulostaa:    1    003  2   1
```

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

```
//
        Console.WriteLine("{0} {2} {1} {0}", 1, 2, 3);
        Console.WriteLine("{0} {2,6:000} {1} {0}", 1, 2, 3);
        Console.WriteLine("{0} {2,-6:000} {1} {0}", 1, 2, 3);
        Console.WriteLine("{0} {2,6:000} {1} {0}", 1, 2, 2017);
        Console.WriteLine("{0} {2,-6:000} {1} {0}", 1, 2, 2017);
        Console.WriteLine("{0:F5} ", 123.4567);  // reaaliluku 5:llä desimaalilla
        Console.WriteLine("{0,11:F5} ", 123.4567);
```

Toki tulostettavat arvot voivat olla mitä tahansa muuttujiakin (vastaavasti toki`String.Format`-funktiossakin) tai jopa lausekkeita:

```
//
        int a = 7;
        int b = 29;
        int c = 11;
        Console.WriteLine("{0} {2} {1} {0,3:00}", a, b, 2*c + 3); // 7 25 29  07
```

Sama esimerkki dollari-muotoilulla (*String Interpolation*)

```
//
        int a = 7;
        int b = 29;
        int c = 11;
        Console.WriteLine($"{a} {c,6:000} {b} {a}");
```

Vaikka uusi *String Interpolation* onkin mukava, joutuu kuitenkin edelleenkin käyttämään vanhempaa `String.Format` ainakin silloin, kun haluaa muotoilujonon olevan muuttujassa. Muotoilujonon sisällön pitäminen muuttujassa on järkevää esimerkiksi jos samaa muotoilua tarvitaan monta kertaa.

```
//
        int a = 7;
        int b = 29;
        int c = 11;
        string muotoilu = "Luvut ovat: {0} {2,6:000} {1} {0}";
        Console.WriteLine(String.Format(muotoilu, a, b, c));
        Console.WriteLine(String.Format(muotoilu, a, c, b));
```

Lisätietoja merkkijonojen muotoilusta löytyy MSDN-dokumentaatiosta:

- Custom Numeric Format Strings
- Interpolated Strings

12.8.1 Yhteenveto erilaista tavoista tulostaa

`WriteLine`-aliohjelmalle menee perusmuodossa parametrina yksi arvo. Jos parametri on merkkijonoviite, se tulostetaan merkkijonona. Normaalisti tulostettavia alkioita ei erotella pilkuilla, vaan "liimataan" yhteen plus-operaattotilla. `Format`-funktiolla voidaan tuottaa merkkijono, jossa tulostettavat lausekkeet ovat halutussa kohtaa. Ja tämä syntynyt jono voidaan antaa `WriteLine`-aliohjelmalle. Tästä on olemassa erikoistapaus, jossa `WriteLine`-aliohjelman ensimmäinen parametri on samanlainen muotoilujono kuin `Format`-funktiossa ja sitten sen perässä pilkulla eroteltuina tulostettavat lausekkeet.

Edellisestä seuraa, että ei ole syntaktisestä väärin kirjoittaa

```
Console.WriteLine("Kissa", ika, paino);
```

mutta tämä tulostaa vain `Kissa`, koska nyt ensimmäinen parametri tulkitaan muotoilujonoksi ja koska siinä ei ole yhtään tulostuspaikkaa aaltosuluilla merkittynä, niin jono tulostuu sellaisenaan.

Sitten

```
Console.WriteLine("Kissa" + ika + paino);
```

tulostaisi kaiken rumasti yhteen ja siksi noiden väliin on tavalla tai toisella saatava ainakin yksi välilyönti.

Seuraavassa esimerkissä on yhteenveto eri tavoista:

```
1         string elain = "Kissa";
2         int ika = 5;
3         double paino = 3.2;
4         Console.WriteLine(elain, ika, paino); // tulostaa vain Kissa, koska
5                                               // luulee 1. param muotoilujonoksi
6         Console.WriteLine(elain + ika + paino);  // Kissa53.2
7         Console.WriteLine(elain + " " + ika + " " + paino);  // Kissa 5 3.2
8         Console.WriteLine(String.Format("{0} {1} {2}", elain, ika, paino));
9         Console.WriteLine("{0} {1} {2}", elain, ika, paino); // Kissa 5 3.2
10        Console.WriteLine("{0} {1} {2:0.00}", elain, ika, paino); // Kissa 5 3.20
11        Console.WriteLine($"{elain} {ika} {paino}"); // Kissa 5 3.2
12        Console.WriteLine($"{elain} {ika} {paino:0.00}"); // Kissa 5 3.20
```

12.9 Char-luokka

Tyyppi `char` edustaa yhtä kirjainta ja sitä vastaava vakioarvo laitetaan yksinkertaisiin heittomerkkeihin. Merkkijonoluokista `string` ja `StringBuilder` tehdyt muuttujat ovat olioviitteitä ja niillä on tukku metodeja, joilla olioita voidaan käsitellä. Näistä oli edellä paljon esimerkkejä. Koska `char` on perustietotyyppi, niin sillä ei ole vastaavia metodeja.

char-tyyppi ◘ Luento 8 (2m19s)

On kuitenkin luokka `Char`, joka sisältään joukon staattisia funktioita, joilla voidaan tuottaa uusia kirjain arvoja tai kysellä kirjaimeen liittyviä asioita. Luokan funktioita kutsuttaessa pitää kertoa minkä luokan funktiota kutsutaan ja parametrina viedä "tutkittava asia", eli muoto on:

```
char uusiKirjain = Char.Funktio(kirjain);
if (Char.IsFunktio(kirjain) ...
```

Huomaa että tämä poikkeaa olioiden metodien kutsusta, jotka voivat olla muotoa (esim `String`):

```
string uusiJono = jono.Metodi();
```

Char-luokkaan liittyvät metodit Luento 8 (1m34s)

Erilaisia yhden kirjaimen muunnoksia ja vertailuja löytyy Char-luokasta:

```
1         char c = 'a';
2         char n = '5';
3         char isoC = Char.ToUpper(c);
4         Console.WriteLine("{0} isona on {1}",c,isoC);
5         if ( Char.IsDigit(n) ) Console.WriteLine("n on numero");
6         if ( Char.IsLetter(c) ) Console.WriteLine("c on kirjain");
7         if ( Char.IsLower(c) ) Console.WriteLine("c on pieni");
```

Tehtävä 12.3 Funktioita

Kirjoita kutsuja vastaavat funktiot niin että ohjelma toimii. Älä muuta aliohjelmakutsuja, kirjoita pelkät aliohjelman toteutukset. Aliohjelmia/funktioita tarvitaan yhteensä kuusi. Kannattaa valita ”Highlight” aja napin vierestä jos ei jo ole värillisenä valmiiksi. 10pt

```
1  public class Funktioita
2  {
3      public static void Main()
4      {
5        string kokoNimi = LiitaMerkkijonot("Vinssi", "Vikkelä");
6        // Liittää merkkijonot yhdeksi merkkijonoksi
7        int mjPituus = Pituus("Sonaatti");
8        // palauttaa merkkijonon pituuden
9        string siistitty = PoistaValit("    koira    ");
10       // poistaa valit alusta ja lopusta
11       char ekaMerkki = Eka(siistitty);
12       // palauttaa merkkijonon ensimmäisen merkin
13       char vikaMerkki = Vika(siistitty);
14       // palauttaa merkkijonon viimeisen merkin
15       Tulosta(kokoNimi, mjPituus, siistitty, ekaMerkki, vikaMerkki);
16       // tulostaa kaikki muuttujat jollakin tavalla, ei palauta mitään
17     }
18 }
```

Luku 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 aluksi näistä ensimmäiseen, `if`-rakenteeseen.

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

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

Kaarisuluissa oleva `ehto` on *looginen lauseke*, jota seuraa aaltosulkeissa oleva *runko-osa*. Loogisen lausekkeen arvo on tosi (true) tai epätosi (false). Looginen lauseke voi sisältää muun muassa lukuarvojen vertailua vertailuoperaattoreilla.

Mikäli looginen lauseke saa arvon `true`, suoritetaan runko-osa. Mikäli looginen lauseke saa arvon `false`, runko-osaa ei suoriteta vaan hypätään sen yli ja jatketaan ohjelman suoritusta.

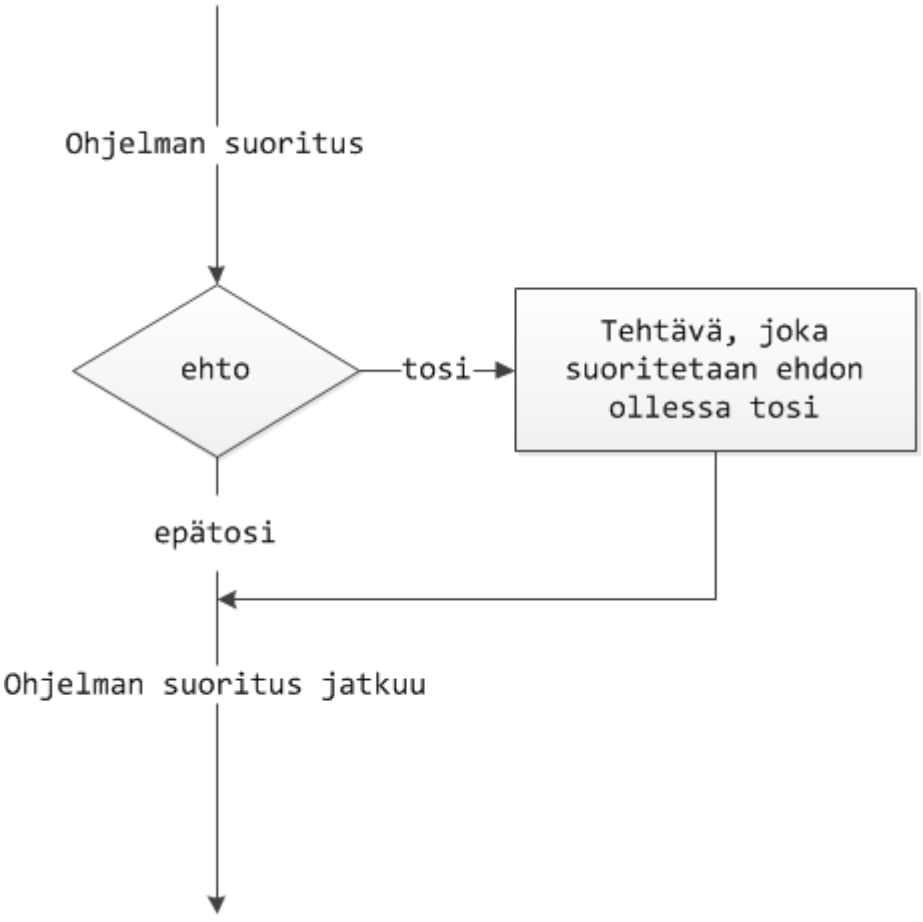
Esimerkkinä näytetty ehtolause "Jos aurinko paistaa, mene ulos" voidaan nyt esittää C#:n syntaksin mukaan seuraavasti.

```
if (aurinkoPaistaa)
{
    MeneUlos();
}
```

Jos lohkon sisällä on vain yksi lause, kuten esimerkissämme yllä, voidaan aaltosulut jättää pois ja kirjoittaa `if`-lause yhdelle riville runko-osan kanssa.

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

Vuokaaviolla `if`-rakennetta voisi kuvata seuraavasti:



Kuva 13: if-rakenne vuokaaviona.

> *Vuokaavio* = Kaavio, jolla mallinnetaan *algoritmia* tai prosessia.

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

13.3 Vertailuoperaattorit

If-lauseen suluissa olevat ehto pitää olla joku totuusarvoinen lauseke. Esimerkiksi

```
if (a > 3) ...
if (a != 2) ...
```

Jos `a` olisi vaikkapa `5`, niin ensimmäinen lauseke `a > 3` olisi totta, samoin toinen koska `a` on erisuuri kuin `2`. Jos taas `a` olisi vaikkapa `2`, niin molemmat suluissa olevat lausekkeet olisivat epätosia.

Vertailuoperaattoreilla voidaan vertailla aritmeettisia arvoja ja osin myös merkkijonoja ja muitakin olioita jos niille on operaattori määritelty.

Taulukko 6: C#:n vertailuoperaattorit.

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

Animaatio: Suorita ohjelma

Askella ohjelman läpi vihreällä nuolella Tutki ohjelman toimintaa

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

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

13.4 Esimerkki: yksinkertaisia if-lauseita

Yhtäsuuruuden vertailuoperaattorissa on kaksi `=` -merkkiä.

```
1         int henkilonIka = 20;
2         if (henkilonIka == 20) Console.WriteLine("Onneksi olkoon!");
```

Alla oleva aiheuttaa virheilmoituksen, koska on yritetty verrata käyttäen vain yhtä `=` -merkkiä.

```
1         // TÄMÄ OHJELMA EI KÄÄNNY
2         int henkilonIka = 20;
3         if (henkilonIka = 20) Console.WriteLine("Onneksi olkoon!");
```

Seuraava esimerkki havainnollistaa toisen vertailuoperaattorin käyttöä.

```
1         int luku = -7;
2         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 `true` 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, mikäli 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;
```

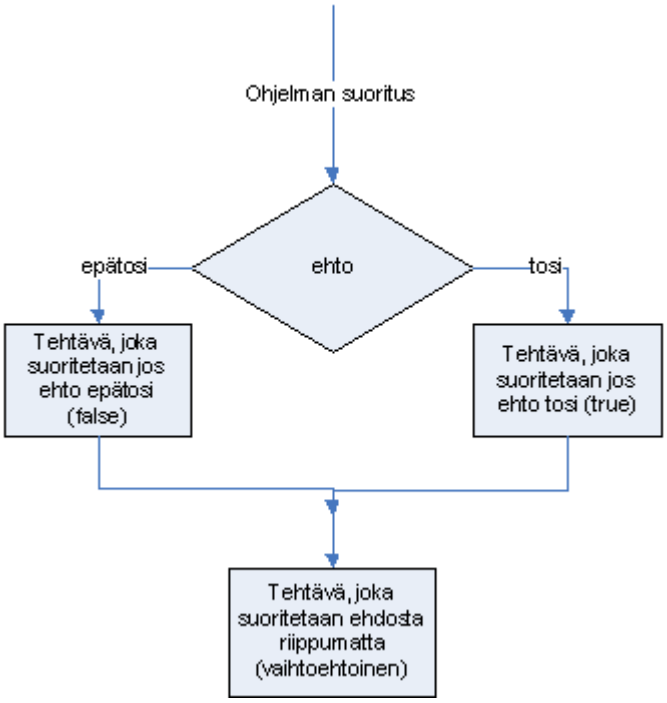
if-else -rakenteen yleinen muoto:

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

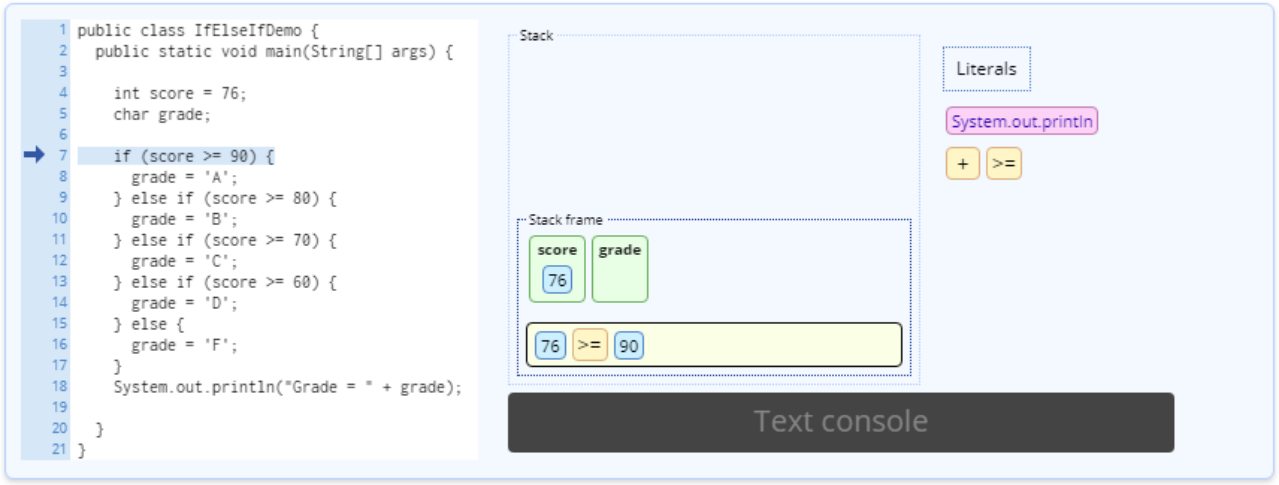
Kuten pelkässä if-rakenteessa myös if-else -rakenteessa lauseiden tilalla voi olla myös lohko.

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

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



Kuva 14: if-else-rakenne vuokaaviona.



Tutki if-else toimintaa (animaatio verkkoversiossa)

13.5.1 Esimerkki: Pariton vai parillinen

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

```
1     public static bool OnkoLukuParillinen(int luku)
2     {
3        if ((luku % 2) == 0) return true;
4        else return false;
5     }
```

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 jakolaskun jakojäännös operaattorilla %. Jos jakojäännös on 0, niin silloin luku on parillinen, eli

palautetaan true. Jos jako ei mennyt tasan (jakojäännös eri kuin 0), niin silloin luvun on pakko olla pariton, eli palautetaan false.

Itse asiassa, koska aliohjelman suoritus päättyy return-lauseeseen, voidaan 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:

```
1     public static bool OnkoLukuParillinen(int luku)
2     {
3        if ((luku % 2) == 0) return true;
4        return false;
5     }
```

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

```
1     public static bool OnkoLukuParillinen(int luku)
2     {
3        return (luku % 2) == 0;
4     }
```

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.

Milloin tarvitaan else-lausetta Luento 8 (1m54s)

Loogisia arvoja ei ole koskaan tyylikästä testata muodossa

```
if ( ( a < 5 ) == true ) ...     // vaan if ( a < 5 ) ...
if ( ( a < 5 ) == false ) ...    // vaan if ( a >= 5 ) ...
                                 // tai if ( 5 <= a ) ...
if ( OnkoLukuParillinen(3) == false ) ...
   // vaan if ( !OnkoLukuParillinen(3) ) ...
```

13.6 Loogiset operaattorit

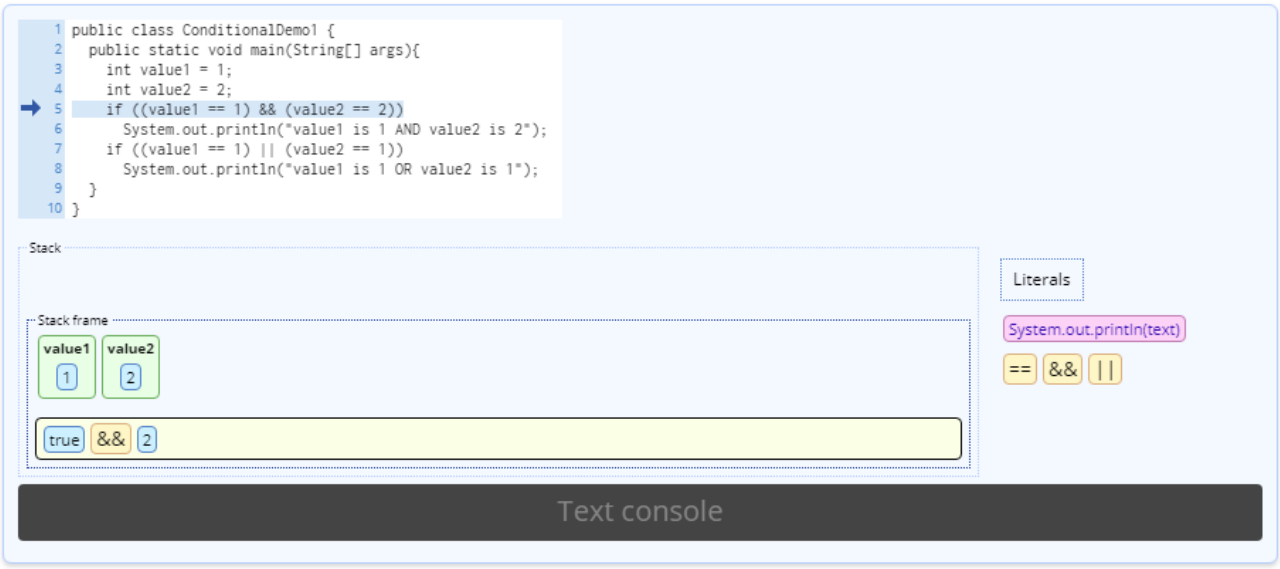
Loogisia lausekkeita voidaan myös yhdistellä *loogisilla operaattoreilla*.

Taulukko 7: Loogiset operaattorit.

C#-koodi	Operaattori	Toiminta
`!`	looginen **ei**	Tosi, jos lauseke epätosi.
`&&`	looginen ehdollinen **ja**	Tosi, jos molemmat lausekkeet tosia. Eroaa seuraavasta 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 **ja**	Tosi, jos molemmat lausekkeet tosia. Suorittaa aina molemmat ehdot (turhaan).

C#-koodi	Operaattori	Toiminta
`||`	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.
`|`	looginen **tai**	Tosi, jos toinen lausekkeista on tosi. Suorittaa aina molemmat ehdot (turhaan).
`^`	eksklusiivinen tai (XOR)	Tosi, *jos* toinen, *mutta* eivät molemmat, on tosi.

Kannattaa yleensä käyttää nimenomaan noita kahden merkin operaattoreita `&&` ja `||`, koska ne lopettavat ehdon laskemisen heti kun totuusarvo on selvinnyt.



Tutki loogisia operaattoreita (animaatio verkkoversiossa)

13.6.1 Operaattoreiden totuustaulut

Taulukko 8: Seuraavassa 0=epätosi, 1=tosi. Totuustaulu eri operaattoreille. Voit kuvitella että `||` on kuten + ja `&&` kuten kertolasku. Ja 1 on mikä tahansa positiivinen luku, eli `1+1=1`.

p	q	p && q	p || q	p ^ q	!p
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

13.6.2 Operaattoreiden käyttö

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. Alla olevat lauseet ovat siis samoja:

```
1         int luku = 7;
2         if ( !(luku <= 0) ) Console.WriteLine("Luku on suurempi kuin nolla");
3         if ( luku > 0 ) Console.WriteLine("Luku on suurempi kuin nolla");
```

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

```
1         int luku = 7;
2         if ((1 <= luku) && (luku <= 99)) Console.WriteLine("Luku on välillä 1-99"↩
      );
```

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

```
1         int luku = 7;
2         if (1 <= luku)
3             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, että käytetään aina pienempi kuin -merkkiä (nuolen kärki vasemmalle). Tällöin epäyhtälön operandit ovat samassa järjestyksessä, jossa ihmiset mieltävät lukujen suuruusjärjestyksen.

Loogisia operaattoreita voi olla samassa ehdossa enemmänkin kuin yksi. Niiden suorituksessa käytetään järjestystä `&&`-operaattorit ensin (vertaa kertolasku) ja sitten `||`-operaattorit (vrt. yhteenlasku). Jos ei ole varma suoritusjärjestyksestä, kannattaa käyttää sulkuja selventämään asiaa.

```
1         int a=1, b=2, c=3;
2         if ( a == 1 && b < 3 && 0 < c )
3            Console.WriteLine("Kaikki oikein :-)");
```

```
1         int a=1, b=2, c=-33, d=12;
2         if ( a == 1 && b < 3 && 0 < c || 10 < d )
3            Console.WriteLine("Riitti että d > 10");
```

Usein funktiossa voidaan palauttaa heti arvo kun jonkin asian tiedetään olevat totta. Seuraavana sama funktio kirjoitettuna 3:lla eri tavalla. Yhdistetyllä ehtolauseella, ilman if-lausetta sekä monena if-lauseena, jossa palautetaan aina tieto jota pidetään “varmana”.

```
1     public static bool OnkoLuku11tai13a(int luku)
2     {
3        if (luku == 11 || luku == 13) return true;
4        return false;
5     }
```

```
6
7
8       public static bool OnkoLuku11tai13b(int luku)
9       {
10         return (luku == 11 || luku == 13);
11      }
12
13
14      public static bool OnkoLuku11tai13c(int luku)
15      {
16         if (luku == 11) return true; // jos tässä poistuttiin, niin luku ei ole 11
17         if (luku == 13) return true; // jos tässä poistuttiin, niin luku ei ole 11↩
        eikä 13
18         return false;
19      }
```

Tehtävä 13.1

Kirjoita aliohjelma, jolle viedään kolme lukua ja se palauttaa tiedon onko niistä mitkään kaksi samoja. Eli luvuista 1,2,3 palauttaa false ja luvuista 1,2,2 palauttaa true. Ennen funktion toteutuksen kirjoittamista, kirjoita funktiolle lisää ComTestejä10pt

```
1  //
2       /// <summary>
3       /// Palautetaan tosi, jos vähintään kaksi luvuista on samoja
4       /// </summary>
5       /// <param name="luku1">Ensimmäinen luku</param>
6       /// <param name="luku2">Toinen luku</param>
7       /// <param name="luku3">Kolmas luku</param>
8       /// <returns>true jos vähintään kaksi lukua ovat samoja</returns>
9       /// <example>
10      /// <pre name="test">
11      ///   OnkoSamoja(1, 2, 2) === true;
12      /// </pre>
13      /// </example>
14      public static bool OnkoSamoja(int luku, int luku2, int luku3)
15      {
16          return false;
17      }
```

13.6.3 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.

Kokeile ohjelmaa kaikilla mahdollisilla p:n ja q:n arvoilla.10pt

```
1           bool p = true;
2           bool q = true;
3           Console.WriteLine(!(p || q) == (!p && !q));
4           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, voidaan ehdot saattaa siistimpään muotoon.

Olkoon esimerkiksi

```
p = a < 5
q = b < 3
```

eli vaiheittain saadaan muutettua

```
if ( !(a < 5  && b < 3) ) ...
if ( !(a < 5) || !(b < 3) ) ... // de Morgan
if ( a >= 5 || b >= 3 ) ...     // sovelletaan not operaattoria
```

jolloin ei-operaattorin siirto voikin olla mielekästä.

Toinen tällainen laki on osittelulaki.

13.6.4 Osittelulaki

Koulusta tuttu osittelulaki sanoo, että kertolasku voidaan ottaa yhteiseksi tekijäksi ja päinvastoin:

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

esimerkiksi

```
2*(3+4) = 2*7 = 14
2*3 + 2*4 = 6 + 8 = 14
```

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

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

eli esimerkiksi:

```
(a > 5) && ((b < 3) || (c==2)) on sama kuin
(a > 5) && (b < 3) || (a > 5) && (c==2)
```

Tämä voidaan todistaa esimerkiksi totuustaululla. Nimetään ehdot:

```
a > 5   : A
b < 3   : B
c == 2  : C
```

Kukin näistä osaehdoista voi olla epätosi (0) tai tosi (1). Kirjoitetaan taulukkoon (=totuustaulu) kaikki ehtojen `A`, `B` ja `C` kombinaatiot (käytännössä peräkkäiset 3 bitin luvut). Loogisen totuuden kannalta `&&` ja `&` toimivat samalla tavalla, joten kirjoitetaan laiskuuksissaan vain yhdellä merkillä (mutta C#:issa ÄLÄ sekoita `&&` ja `&`). Seuraaviin sarakkeisiin otsikoksi kaikki lauseesta löytyvät osalausekkeet (kuten `B|C` jne) ja seuraaville riveille kunkin osalausekkeen totuusarvo (esimerkiksi kun `A=1, B=1, C=0` niin `B|C=1`).

A	B	C	B|C	A&(B|C)	A&B	A&C	A&B | A&C
0	0	0	0	**0**	0	0	**0**
0	0	1	1	**0**	0	0	**0**
0	1	0	1	**0**	0	0	**0**

A	B	C	B|C	A&(B|C)	A&B	A&C	A&B | A&C
0	1	1	1	**0**	0	0	**0**
1	0	0	0	**0**	0	0	**0**
1	0	1	1	**1**	0	1	**1**
1	1	0	1	**1**	1	0	**1**
1	1	1	1	**1**	1	1	**1**

Lopuksi huomataan, että väittämän oikea ja vasen puoli (tummennettu sarake) on kaikissa tilanteissa sama. Silloin lause on todistettu oikeaksi.

Päinvastoin kuin normaalissa aritmetiikassa, loogisille operaatioille osittelulaista on myös toinen versio (joka johtuu siitä, että `1||1 == 1`):

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

Toimiiko luvuilla?

Kokeile muuten luvuilla, että tämä ei toimi jos `&` tilalla on `*` ja `|` tilalla `+`. Tässähän ei päde että `1+1 = 1`.
Lauseen kumoamiseksi riittää yksikin vastaesimerkki, jossa vasen ja oikea puoli lauseesta eroavat toisistaan. Todeksi näyttämiseksi pitää käydä kaikki mahdolliset vaihtoehdot läpi (mikä luvuilla olisi mahdotonta, totuusarvoille tehtävissä) tai sitten käyttää matemaattisesti yleispätevää todistusta.

Kokeile toista osituslakia

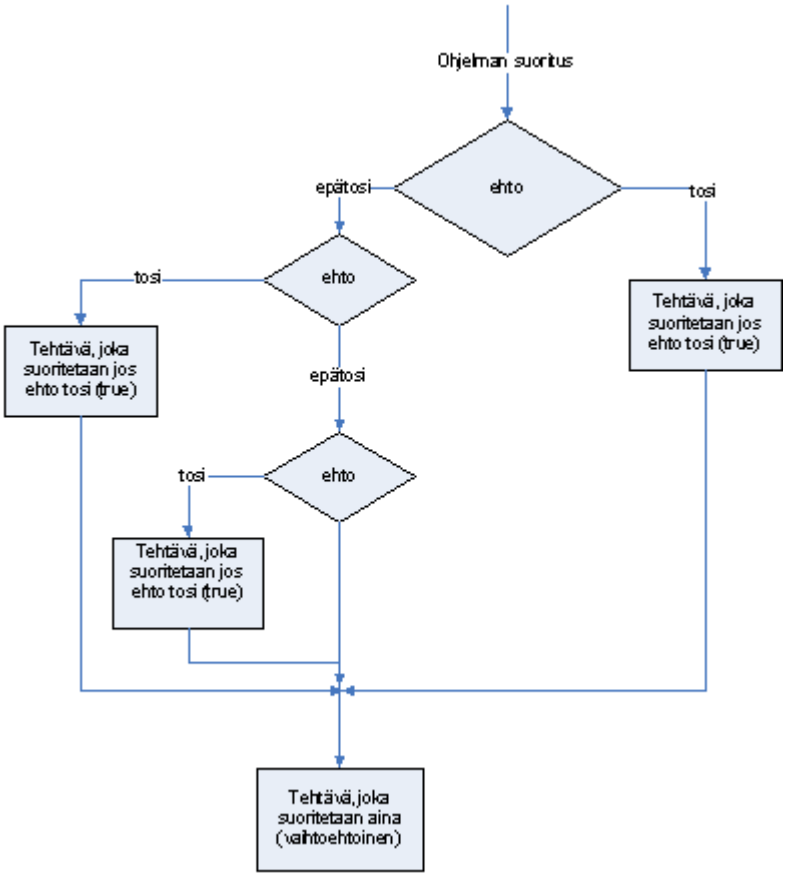
Tässä on uudelleen ”todistus” osittelulaille, vertaa sitä yllä olevaan

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.



Kuva 15: else-if-rakenne vuokaaviona.

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

13.7.1 Esimerkki: Tenttiarvosanan laskeminen

Tehdään laitoksen henkilökunnalle aliohjelma, joka laskee opiskelijan tenttiarvosanan. Parametreina 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.

```
1  using System;
2  /// <summary>
3  /// Laskee opiskelijan tenttiarvosanan.
4  /// </summary>
5  public class LaskeTenttiArvosana
6  {
7      /// <summary>
8      /// Laskee tenttiarvosanan pistevälien mukaan.
9      /// </summary>
10     /// <param name="maksimipisteet">Tentin pisteet jolla saa 5</param>
11     /// <param name="lapaisyraja">Tentin läpipääsyraja</param>
12     /// <param name="pisteet">Opiskelijan saamat tenttipisteet</param>
13     /// <returns>tenttiarvosana välillä 0-5.</returns>
14     /// <example>
15     /// <pre name="test">
16     ///    LaskeArvosana(5, 1, 0) === 0;
17     ///    LaskeArvosana(5, 1, 1) === 1;
18     ///    LaskeArvosana(5, 1, 2) === 2;
```

```
19      ///     LaskeArvosana(5, 1, 3) === 3;
20      ///     LaskeArvosana(5, 1, 4) === 4;
21      ///     LaskeArvosana(5, 1, 5) === 5;
22      ///     LaskeArvosana(5, 1, 6) === 5;
23      /// </pre>
24      /// </example>
25
26      public static int LaskeArvosana(int maksimipisteet,
27                               int lapaisyraja, int pisteet)
28      {
29          //Lasketaan eri arvosanoille tasaiset pistevälit
30          int pisteErot = (maksimipisteet - lapaisyraja) / (5-1);
31          int arvosana = 0;
32
33          if      (lapaisyraja + 4 * pisteErot <= pisteet) arvosana = 5;
34          else if (lapaisyraja + 3 * pisteErot <= pisteet) arvosana = 4;
35          else if (lapaisyraja + 2 * pisteErot <= pisteet) arvosana = 3;
36          else if (lapaisyraja + 1 * pisteErot <= pisteet) arvosana = 2;
37          else if (lapaisyraja + 0 * pisteErot <= pisteet) arvosana = 1;
38          return arvosana;
39      }
40
41      /// <summary>
42      /// Pääohjelmassa tehdään testitulostuksia
43      /// </summary>
44      public static void Main()
45      {
46          //Tehdään muutama testitulostus
47          Console.WriteLine(LaskeArvosana(100, 50, 75));
48          Console.WriteLine(LaskeArvosana(24, 12, 12));
49      }
50  }
```

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 *silmukalla* ja/tai *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;` ? Kokeile.

Tehtävä 13.2

Täydennä funktio Samat, joka palauttaa true, jos sille viedään kaksi samaa merkkijonoa. Toisin kuin equals, se ei ota huomioon isoja ja pieniä kirjaimia. Merkkijono ”Kukka” ja ”KUKKA” siis palauttaisi arvon true. Lisää testejä.10pt

```
using System;
///@author
///@version
///
/// <summary>
/// Testaillaan merkkojonojen vastaavuuksia
/// </summary>
public class SamatMerkkijonot
{
        /// <example>
        /// <pre name="test">
        /// Samat("Kukka","KUKKA") === true;
        /// </pre>
        /// </example>
        public static bool Samat(string a, string b)
        {

            return false;
        }

    /// <summary>
    /// Pääohjelmassa tehdään testitulostustuksia
    /// </summary>
    public static void Main()
    {
        //Tehdään muutama testitulostus
        Console.WriteLine(Samat("Kissa", "kissa"));  // pitäisi tulla true
    }
}
```

13.8 switch-rakenne

`switch`-rakennetta voidaan käyttää silloin, kun valinta halutaan tehdä lausekkeen perusteella. Lausekkeen kullekin odotetulle arvolle merkitään switch-rakenteessa oma case-osa. Yleinen muoto switch-rakenteelle on seuraava.

```
switch (valitsin) // valitsin on lauseke jonka arvo ei ole null
{
   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 esimerkiksi C++:ssa, ei C#:ssa sallita suorituksen siirtymistä tapauksesta (case) toiseen, mikäli tapauksessa on yksikin lause. Esimerkiksi seuraava koodi aiheuttaisi virheen.

```
1         int valitsin = 1;
2         switch (valitsin)
3         {
4             // Seuraava koodi aiheuttaa virheen
5             case 1:
6                 Console.WriteLine("Tapaus 1...");
7                 // Tähän kuuluisi break-lause tai muu hyppylause!
8                 // kokeile esim:
9                 // goto case 2;
10            case 2:
11                Console.WriteLine("... ja/tai tapaus 2");
12                break;
13        }
```

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

```
1         int luku = 1;
2         switch (luku)
3         {
4             case 0:
5             case 1:
6                 Console.WriteLine("Luku on 0 tai 1");
7                 break;
8             case 2:
9                 Console.WriteLine("Luku on 2");
10                break;
11            default:
12                Console.WriteLine("Oletustapaus");
13                break;
14        }
```

Animaatio: Suorita ohjelma

Askella switch-rakenne vihreällä nuolella Tutki switch-rakenne

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 on jätettävä 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";
       }
```

```
13      }
```

`break`-lauseen voi siis pitää 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.

```
1     public static string KirjallinenArvosana(int numero)
2     {
3         if ( numero == 0 ) return "Hylätty";
4         if ( numero == 1 ) return "Välttävä";
5         if ( numero == 2 ) return "Tyydyttävä";
6         if ( numero == 3 ) return "Hyvä";
7         if ( numero == 4 ) return "Kiitettävä";
8         if ( numero == 5 ) return "Erinomainen";
9         return "Virheellinen arvosana";
10    }
```

Miksi edellä ei tarvittu `else`-lauseita? Mitä edellä tapahtuisi mikäli viimeinen `return` jätettäisiin pois? Kokeile! Miksi?

```

```

Tehtävä 13.3

Täydennä luokka. Lisää siihen pääohjelma sekä aliohjelma, jolle viedään parametrina yksi kokonaisluku. Aliohjelma palauttaa sitä vastaavan viikonpäivän. Esim. jos viedään luku 1, palautetaan merkkijono ”maanantai”. Ota huomioon myös, mitä tapahtuu jos luku ei ole välillä 1-7. Voit myös lisätä testit funktiolle. 10pt

```
1  using System;
2
3  ///@author
4  ///@version
5  ///
6  /// <summary>
7  /// Tulostetaan viikonpäiviä
8  /// </summary>
9  public class Viikonpaiva
10 {
11
12 }
```

Luku 14

Olioiden ja alkeistietotyyppien erot

Tässä luvussa tutustutaan siihen, miten oliotietotyypit tallennetaan muistiin ja mikä niiden ero on alkeistietotyyppeihin kuten esimerkiksi `int`. Tähän lukuun kannattaa palata useammankin kerran, jotta lopulta ymmärtäisi mistä on kyse. Ensimmäisellä lukukerralla ei luonnollisesti kaikki tule ymmärretyksi. Kuitenkin ymmärtämällä tämän luvun asiat, on monet muut tämän kurssin asiat helpompi sisäistää. Eli tähän kannattaa panostaa!

Lukemisen rinnalla voi kannattaa ajaa esimerkkiohjelmaa oman IDE:n debuggerilla kuten on tehty alla olevassa videossa.

Miksi esimerkissä on käytetty StringBuilder?

Esimerkissä käytetään oliotietotyyppinä `StringBuilder`-luokan ja taulukkoluokan olioita. Kaikki muutkin oliotyypit kuten Jypelin fysiikkaoliot yms. toimivat vastaavasti. Eli ei saa takertua siihen että esimerkissä on `StringBuilder`, vaan yleistää ajatus kaikille oliotietotyypeille. Muuttumatonta merkkijonoluokkaa `String` ei ole käytetty sen takia, että C#-kielessä `String`-tyypin olioita voidaan verrata sisällön puolesta `==`-operaattorilla, mutta juuri muita oliotietotyyppejä ei voi, vaan `==` vertaa olioviitteiden arvoja. Eli sikäli `StringBuiler` on hyvä yksinkertainen luokka edustamaan muitakin olioluokkia.

Alla oleva video pysähtyy siihen kun tämä asia on käsitelty, mutta jos jatkat sen katsomista, on lopussa esimerkkejä, joissa on muunkin tyyppisiä olioita.

Tehdään ohjelma, jolla demonstroidaan olioiden ja alkeistietotyyppien eroja.
 Katso video Luento 11, syksy 2024 (30m59s)

```
1  using System;
2  using System.Text;
3
4  /// <summary>
5  /// Tutkitaan olioviitteiden käyttöä ja käyttäytymistä.
6  /// </summary>
7  public class Olioviitteet
8  {
9    /// <summary>
10   /// Alustetaan muuttujia ja tulostetaan.
11   /// Testaillaan olioiden ja alkeismuuttujien eroja.
```

```
12    /// </summary>
13    public static void Main()
14    {
15      StringBuilder s1 = new StringBuilder("eka");
16      StringBuilder s2 = new StringBuilder("eka");
17
18      Console.WriteLine(s1 == s2);       // false
19      Console.WriteLine(s1.Equals(s2));  // true
20      Console.WriteLine(s1.Equals("eka"));  // true
21      Console.WriteLine("eka".Equals(s2));  // false
22
23      int i1 = 11;
24      int i2 = 10 + 1;
25
26      Console.WriteLine(i1 == i2);       // true
27
28      int[] it1 = new int[1]; it1[0] = 3;
29      int[] it2 = new int[1]; it2[0] = 3;
30
31      Console.WriteLine(it1 == it2);       // false
32      Console.WriteLine(it1.Equals(it2));  // false
33      Console.WriteLine(it1[0] == it2[0]); // true
34
35      s2 = s1;
36      Console.WriteLine(s1 == s2);         // true
37    }
38  }
```

Tarkastellaan ohjelmaa hieman tarkemmin:

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

Yllä luodaan kaksi StringBuilder-luokan ilmentymää eli oliota. Muuttujat `s1` ja `s2` ovat viitteitä noihin olioihin. Eli teksti `eka` ei missään tapauksessa sijaitse muistipaikassa `s1`.

```
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
```

Olioiden sisältöjä, 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 arvojen 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 alkeistietotyyppiä), niin *kokonaislukutaulukko* on olio. Jälleen verrataan taulukkomuuttujien viitteitä, eikä arvoja joihin muuttujat

viittaavat.

Ohjelman kaikki muuttujat ovat lokaaleja (paikallisia) muuttujia, eli ne on esitelty lokaalisti `Main`-funktion sisällä eivätkä "näy" näin ollen `Main`-funktion ulkopuolelle. Tällaisille muuttujille kääntäjä varaa tilaa yleensä kutsupinosta. Kutsupino on dynaaminen tietorakenne, johon tallennetaan kunkin aliohjelman lokaalit muuttujat ja tieto siitä, mihin pitää palata kun aliohjelma on suoritettu loppuun. Kutsupinosta käytetään usein myös pelkästään nimeä pino. Kutsupinosta puhutaan lisää kurssilla ITKA203 Käyttöjärjestelmät.

Kääntäjä on ohjelman käännösvaiheessa etsinyt mistä muuttujille `s1, s2, i1, i2, it1, it2` löydetään tilaa (allokoinut, varannut). Tässä alempana olevan leikkiesimerkin kuvassa tilaa on löytynyt muistipaikoista 100-120. Oikeasti osoitteet voivat olla ihan muuta ja prosessoriarkkitehtuurista riippuen jopa muuallakin kuin pinomuistissa, esimerkiksi rekistereissä. Oleellista on, että muistin kannalta ja prosessorin kannalta muisti on samanlaista "lukumössöä" eli bittijonoja riippumatta mitä tietotyyppiä siinä on. Tietotyyppi vaikuttaa vaan siihen, montako tavua muistia tarvitaan kutakin muuttujaa varten ja miten sisältöä tulkitaan. Jos haluat tarkempaa tietoa, katso lukua 26. Lukujen esitys tietokoneessa Esimerkin kuvassa on oletettu, että kukin muuttuja tarvitsee int-tyypin verran tilaa, eli 32 bittiä, eli 4 tavua.

Ennen riviä `s2 = s1;` pino voisi hieman yksinkertaistettuna olla lokaalien muuttujien ja kekomuistin olioiden kohdalta suurin piirtein seuraavan näköinen:

Kuva 16: Olioviitteet

Operaatio `new StringBuilder("eka")` varaa kekomuistista (tämäkin voi vaihdella eri arkkitehtuureissa) tilaa sen verran, mitä tarvitaan `StringBuilder`-olion tallentamiseksi (C#:issa yleensä aluksi tilaa 16 kirjaimelle ja sen hallinnalle). Esimerkissä tila on varattu muistipaikasta `8010`. Tällä on esimerkissä yritetty korostaa sitä, että se on hyvin eri paikassa kuin itse lokaalit muuttujat (joiden sijainti alkaa esimerkissä paikasta `100`).
Kun tila on varattu, kutsutaan `StringBuilder`-luokan muodostajaa (tästä enemmän myöhemmillä kursseilla), joka alustaa varatun tilan kirjaimilla `E k a`. Tämän jälkeen `new` palauttaa sen muistipaikan osoitteen, josta varasi tilan. Siksi tässä esimerkissä tuo on tavallaan sijoitus

```
s1 = 8010;
```

Vastaavasti `new int[1];` luo yksipaikkaisen kokonaislukutaulukon, jonka sisältö on alunperin täynnä 0-arvoja. Ja kun tila on varattu `new` palauttaa tiedon mistä muistipaikasta se sai tilan varattua (esimerkissä `8070`) ja tämä sijoitetaan muuttujaan `it1`, joka on muistipaikassa `116`. Sijoitus `it1[0] = 3` vastaavasti tarkoittaa että mennään muistipaikan `it1` osoittaman taulukon (paikassa `8070`) luokse ja sen ensimmäiseen paikkaan (ja ainoaan tässä tapauksessa) sijoitetaan arvo `3`.

Mitä on yksinkertaistettu

Kuvassa on yksinkertaistettu sitä, miltä `StringBuilder` oikeasti näyttää. StringBuilderissä on oikeasti varattu enemmän tilaa kuin tarvitaan ja se tila on vielä "omassa laatikossaan". Tällöin nuolen päässä olevassa oliossa (esim muistipaikassa 8010) olisi hallinta käytettyjen merkkien määrästä ja viite sinne, missä merkit oikeasti ovat. Samoin on yksinkertaistettu sitä, miltä muistipaikoissa 8070 ja 8080 olevat yksipaikkaiset taulukot näyttävät. Lisäksi viitenuolet, tai oikeastaan mitä on nuolen alun ja lopun välillä, on C#:issä monimutkaisemmin automaattiset muistinhallinnan takia. C-kielessä kuva olisi kohtuullisen oikean näköinen, erona lähinnä että merkkijonon viimeisen merkin jälkeen olisi `nul`-merkki.

Lisäksi on piirtämättä vakiomerkkijono `"eka"`, joka aluksi syntyy, jotta siitä voidaan kirjaimet kopioida `StringBuilder`-olioihin.

Olioviitteiden ja olioiden ymmärtämiseksi kuva on kuitenkin riittävä muodostamaan mielikuvamallin siitä, miten perustietotyyppien muuttujat eroavat olioviite-tyyppisistä muuttujista ja olioista.

Luvussa Listan sisäinen rakenne on rakenne piirretty tarkemmin kokonaislukulistan tapauksessa. `StringBuilder` toimii vastaavasti.

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.

Sijoitus `s2 = s1;` toimii aivan vastaavasti kuin toimisi sijoitus `i2 = i1`, eli haetaan muistipaikassa `s1` oleva arvo (esimerkissä `8010`) ja se sijoitetaan muistipaikkaan `s2`. Tietokoneessa ei ohjelman ajon aikana ole noita muuttujien nimiä, vaan esimerkin kuvassa tuo sijoitus `s2 = s1` tarkoittaa prosessorille:

- hae muistipaikassa `100` oleva arvo (esimerkissä `8010`) ja sijoita se muistipaikkaan `104`.

Sijoituksen jälkeen kuvassa muistipaikkaan 8040 ei enää osoita (viittaa) kukaan, jolloin 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 olioviitettä 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.

Muista vielä, että esimerkin muistipaikkojen osoitteet (`100-120` ja `8010-8090`) ovat keksittyjä. Ne vaihtuvat täysin eri käyttökerroilla ja ovat erilaisia eri käyttöjärjestelmissä ja prosessoreissa. Siksi nykykielissä ei oikeasti enää mietitä tuota, mihin muistipaikkoihin mikäkin muuttuja menee. Jatkossa piirretään kuvat niin, että unohdetaan muistipaikat ja viitettä olioon kuvataan nuolella.
Sisäisesti viitteet toteutetaan karkeasti kuten edellisissä kuvissa. Samoin muistipaikat yleensä piirretään erilleen, koska mikään ei luotettavasti takaa, että tietyt muuttujat menisivät peräkkäisiin osoitteisiin.

Pitää kuitenkin muistaa, ettei tietokoneen muistissa ole "nuolia", vaan ainkostaan niitä lukuarvoja (bittijonoja), jotka loogisesti voidaan tulkita viittaamaan (osoittamaan) toiseen paikkaan.

Muistipaikan (muuttujan) tyypin perusteella päätetään miten sen sisältönä olevaa lukuarvoa tulkitaan.

Ilman muistipaikkojen osoitteita piirrettynä edellinen kuva olisi silloin:

Kuva 16b: Piirretään ilman muistipaikkojen osoitteita ja ja paikat erilleen

Voit vielä alla animoida [>]-nuolilla esimerkin toimintaa askel kerrallaan. Animaatiossa ensin syntyy kaikki lokaalit muuttujat (pinossa), koska karkeasti näin kääntäjäkin tekee, eli varaa heti lohkoon (esimerkiksi aliohjelmaan) tultaessa sen tarvitseman muuttujien tilan ja päättää mitä muistipaikkaa käytetään mihinkäkin tarkoitukseen. Esimerkiksi että `s1` on olioviite `StringBuilderiin` ja `i1` tavallinen primitiivitietotyypin (esimerkissä `int`) arvomuuttuja.

Animoi painikkeilla muistin toimintaa

Tutki erityisesti askeleita 7 ja 810pt

Mitä tekee automaattinen muistinhallinta ja muistin siivous (*garpage collection*)?

Kun pääset animaation loppuun, huomaat että muistipaikat `8040` ja `8090` olisivat vapaana. Jos joku nyt tarvitsisi tilaa esimerkiksi 40 tavua, niin tuo `8040` osoitteesta alkava 30 paikkaa ei siihen riittäisi. Jos muualla ei olisi enää tilaa, niin siirtämällä `8070` ja `8080` sisällöt ylemmäksi tulisi niiden alle yhtenäinen 40 tavun tila. Jos uutta tilaa olisi haluttu 30 tavua tai vähemmän, olisi se voitu allokoida suoraan paikasta `8040` alkaen ilman, että mitään olisi tarvinnut siirrellä. Muistin saa korruptoitumaan sillä, että varailee paljon pieniä alueita ja sitten vapauttaa niistä vaikka joka toisen. Sen jälkeen ei löytyisi yhtään isoa yhtenäistä lohkoa ja silloin muistisiivouksella olisi paljon siirreltävää.

Tämä on juuri sitä mitä muistinsiivous (*garbage collection*) tekisi. Esimerkissä jos tuo 40 tavua tarvittaisiin, niin siirtelyn lisäksi pitäisi samalla viitemuuttujien `it1` ja `it2` arvoja muuttaa. Tämän takia oikeasti tuossa "nuolen" kohdalla on monimutkaisempi toteutus, jotta kaikkia tarvittavia viitteitä saataisiin muutettua. Hyvin toimivan muistinhallinnan toteutus on ollut merkittävä alan tutkimuskohde. Toteutukset eivät ole kovin yksinkertaisia ja siksi emme edes tässä vaiheessa yritä piirtää kuvia aivan täsmällisesti "oikein".

Alla olevassa Aalto-yliopiston animaatiossa `id1` on sama kuin `8010` edellä olevissa kuvissa. Vastaavasti `id3` on vastaa paikkaa `8040`. Huomaa että tuossa animaatio on tehty vanhemman C# aikana, jolloin vertailu, jossa on `StringBuilder`-olio ja `String`-olio palautti aina `false`, koska tulkittiin eri eri tyyppiset oliot eivät koskaan voi olla samanlaisia. Tämä muuttui joskus 2020 ja `StringBuilder`-olion vertaaminen `String`-olioon katsotaan sen mukaan miten niiden sisältämät merkkijonot vertautuvat.

Animaatio: Suorita ohjelma

Askella vertailuesimerkkiä vihreällä nuolella.
Huom! .net 5:ssa ja uudemmissa `s1.Equals("eka")` on `true`
Tutki oliotyypin ja perustietotyypin arvojen vertailua

Tarkista tietosi

Mitkä seuraavista ovat totta ja mitkä väärin?

	True	False
int-tyyppinen muuttuja on alkeistietotyyppi	☐	☐
double-tyyppisistä muttujista koostuva taulukko on olio	☐	☐
Kaikki taulukot ovat olioita	☐	☐
int-taulukon tiettyjen alkioiden arvoja voi verrata käyttäen ==	☐	☐
Jos halutaan verrata, onko kaksi taulukkoa samanlaisia, voidaan käyttää vertailussa ==	☐	☐
alkeistietotyypin sijoituksessa sijoitetaan viite	☐	☐
oliotietotyypin sijoituksessa sijoitetaan viite	☐	☐
oliotietotyypeistä tallennetaan viite muistipaikkaan, ja muistipaikassa on olion tiedot	☐	☐
alkeistietotyypeistä ei ole viitettä olioon	☐	☐
alustamattoman lokaalin olio-viitteen arvona on null-viite	☐	☐

Luku 15

Taulukot

Muuttujaan pystytään tallentamaan yksi arvo kerrallaan. Jos haluaisimme tallettaa esimerkiksi kaikkien kuukausien päivien lukumäärän, 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 arvoja, on usein syytä ottaa käyttöön tietorakenne. C#:ssa on useita valmiita tietorakenteita, joista *taulukko* (engl. array) on ehkäpä kaikkein yksinkertaisin. Taulukkoon voi tallentaa useita samantyyppisiä muuttujia. Yksittäistä taulukon muuttujaa sanotaan *alkioksi* (element). Jokaisella alkiolla on taulukossa paikka, jota sanotaan *indeksiksi* (index). Taulukon indeksointi alkaa C#:ssa 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. Tosin on olemassa `Array.Resize`-metodi, joka ei muuta alkuperäisen taulukon kokoa, vaan luo uuden taulukon, kopioi alkuperäisen taulukon kaikki alkiot uuteen taulukkoon, ja sen jälkeen korvaa alkuperäisen taulukon (viitteen) uudella taulukolla (viitteellä). Ks. dokumentti.

Taulukot ja silmukat ovat kuin reppu ja reissumies. Toki kumpikin voi tulla toimeen ilman toista. Mutta usein nimenomaan taulukot ovat tyypillisin silmukoiden käyttökohde. Molemmat pitäisi käsitellä kerralla, mutta toisella on aloitettava. Kun aloitamme taulukoilla, niin joissakin esimerkeissä on pakko viitata silmukoihin, joten palaa uudestaan sellaisiin kohtiin sitten kun olet käynyt läpi silmukka-luvun.

15.1 Taulukon luominen

Taulukon alustaminen ja sen alkioon viittaaminen Luento 9 (7m25s)

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];  //kaikki alkiot null-viitteitä
```

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[] kkPaivat = new int[12]; // kaikki alkiot 0
```

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

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

Esimerkiksi kuukausien päivien lukumäärille voisimme määritellä ja alustaa taulukon seuraavasti:

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

Taulukko voitaisiin nyt kuvata 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: kkPaivat-taulukko

tai:

Kuva 18b: kkPaivat-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. Oletusarvot ovat sellaisia, missä kaikki bitit ovat 0. Tällöin numeeriset arvot alustetaan nollaksi, bool-tyyppi saa arvon false ja oliotyypit (esim. `String`) `null`-viitteen. `char`-tyypillä arvo on merkki, jonka ASCII-koodi on 0. Tulostettuna se ei näy, mutta on eri asia kuin välilyönti, jonka ASCII-koodi on 32 (ks. ASCII-koodi).

Alusta kokonaislukutaulukko t, jossa on arvot 1-7.10pt

Taulukon alkioiden lukumäärä saadaan selville `Length`-ominaisuuden avulla. Vastaavasti siis kuten merkkijonon pituus. Huomaa, että Visual studio voi helposti tässä tarjota laajennusmetodia `Count` joka ei ole oikea vaihtoehto mikäli ei lisätä `using`-lauseita koodin alkuun.

```
1         int[] kkPaivat = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
2         int kuukausia = kkPaivat.Count;
3         System.Console.WriteLine(kuukausia);
```

Ohjelmassa alustetaan taulukot oletusarvoilla. Mitä tulostuu?10pt

```
1         int[] luvut = new int[1];
2         double[] pituudet = new double[1];
3         char[] merkit = new char[1];
4         StringBuilder[] a = new StringBuilder[1];
5         bool[] arvot = new bool[1];
6
7         Console.WriteLine("pituudet[0]: "+ pituudet[0]);
8         Console.WriteLine("merkit[0]: " + merkit[0]);
9         Console.WriteLine("a[0]: " + a[0]);
10        Console.WriteLine("luvut[0]: " + luvut[0]);
11        Console.WriteLine("arvot[0]: " + arvot[0]);
12        string b = null;
13        Console.WriteLine("/"+b+"/");
14        if ( a[0] == null ) Console.WriteLine("on null");
```

15.2 Taulukon alkioon viittaaminen

Taulukon alkioihin pääsee käsiksi taulukon **nimellä** ja **indeksillä**. Molemmat on pakko sanoa, jos halutaan tietystä taulukosta tietyssä paikassa oleva alkio. Vastaavastihan pitää sanoa jonkun osoitteessa että osoite on esimerkiksi

```
Paratiisitie 13
```

Pelkkä `Paratiisitie` ei riittäisi kertomaan henkilön tarkkaa osoitetta ja vastaavasti pelkkä `13` ei yhtään kertoisi missä päin henkilö asuu.

Tähän, että sanotaan “katu” ja “talon numero”, on monessa ohjelmointikielessä valittu syntaksi, jossa ensiksi kirjoitetaan taulukon nimi, jonka jälkeen hakasulkeiden sisään lauseke, joka tuottaa halutun alkion indeksin. Yleisessä muodossa taulukon alkioihin viitataan seuraavasti:

```
taulukonNimi[lausekeJokaTuottaaHalutunIndeksin]

// esimerkkejä tiettyn alkioon viittaamiseksi jos taulukon nimi on `t`
t[0]
int i = 2;
t[i]
t[i-1]
```

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

```
1         Console.WriteLine(kkPaivat[0]); // 31
```

Tai tallentaa tammikuun pituuden edelleen muuttujaan.

```
1         int tammikuu = kkPaivat[0];
2         Console.WriteLine(tammikuu); //tulostuu 31
```

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

```
1         int indeksi = 0;
2         Console.WriteLine(kkPaivat[indeksi]); // 31
3         Console.WriteLine(kkPaivat[indeksi + 3]); // 30
```

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.

```
1         kkPaivat[1] = 29;
2         Console.WriteLine(String.Join(" ",kkPaivat));
```

Kuva 18c: kkPaivat-taulukko muutoksen jälkeen

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

```
1         kkPaivat[12] = 29;
2         Console.WriteLine(String.Join(" ",kkPaivat));
```

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

Myöhemmin opitaan kuinka poikkeuksista voidaan toipua ja ohjelman suoritusta jatkaa.

15.2.1 Funktioita jotka käsittelevät taulukkoa ja niiden testaaminen

Miten voin keksiä silmukan? Lue tästä ideaa.

Ajatellaanpa aluksi, että meillä olisi aluksi taulukko `t`:

```
//          0  1   2   3    indeksit
int[] t = {5, 9, 14, 16};
```

Sitten pitäisi laskea yhteen taulukon luvut muuttujaan `s`. Tämähän onnistuisi alkeellisesti koodilla

```
int s = 0;
s += t[0];
s += t[1];
s += t[2];
s += t[3];
```

Toki tuo koodi olisi aika hyödytön muun kokoisille taulukoille. Aloitetaan järkevämmäksi muuttaminen lisäämällä apumuuttuja `i`:

```
int s = 0;
int i = 0;
s += t[i];
s += t[1];
s += t[2];
s += t[3];
```

Nythän koodi tekee saman asian, koska `t[0]` on sama kuin `t[i]`, koska `i == 0`. Mutta

```
int s = 0;
int i = 0;
s += t[i];
s += t[i];  // i == 0?
s += t[2];
s += t[3];
```

ei enää toimisikaan (paljonko tulisi summaksi?). Siksi i:tä pitää kasvattaa ennen seuraavaa riviä jotta siellä `i` olisi 1, eli:

```
int s = 0;
int i = 0;
s += t[i];
i++;
s += t[i]; // i == 1, hyvä
s += t[2];
s += t[3];
```

Ja kun ajatusta jatketaan eteenpäin saadaan täysin alkuperäisen kanssa saman asian tekevä koodi:

```
int s = 0;
int i = 0;
s += t[i];  // i == 0
i++;
s += t[i];  // i == 1
i++;
s += t[i];  // i == 2
i++;
s += t[i];  // i == 3
i++;        // viimeinen kasvatus on toki turha, mutta ei haittaa => i == 4
```

Nyt kun saimme koodiin täysin identtisiä rivejä, voi olla järkevä käyttää silmukkaa hoitamaan saman asian, samalla näkee että siinä ylimääräisessä kasvatuksessa olikin järkeä:

```
int s = 0;
int i = 0;
while (i < 4)
{
    s += t[i];
    i++;
}
```

Vielä kun 4 korvataan sillä, että otetaan huomioon mahdollinen eri kokoinen taulukko, saadaan:

```
int s = 0;
int i = 0;
while (i < t.Length)
{
    s += t[i];
    i++;
}
```

ja meillä on koodi, joka laskee minkä tahansa (jopa 0) kokoisen taulukon alkioiden summan. Useinhan tällaisessa tapauksessa käytetään mieluummin saman asian tekevää for-silmukkaa (ks. tarkemmin Toistorakenteet (silmukat)-luvusta):

```
int s = 0;
for (int i = 0; i < t.Length; i++)
{
    s += t[i];
}
```

Nyt voimme soveltaa tätä ajatusta seuraavassa `Summa`-funktiossa.

Tehdään esimerkiksi funktio joka laskee yhteen taulukossa olevat luvut. Esimerkissä on myös malleja miten taulukon saa mukaan testiin joko apumuuttujan avulla tai luomalla se suoraan kutsussa.

```
1     public static void Main()
2     {
3         int[] kkPaivat = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4         int paivia = Summa(kkPaivat);
5         System.Console.WriteLine("Vuodessa päiviä: " + paivia);
6
7     }
8
9
10    /// <summary>
11    /// Aliohjelma palauttaa annetun
12    /// kokonaislukutaulukon alkioiden
13    /// summan.
14    /// </summary>
15    /// <param name="taulukko">Kokonaislukutaulukko</param>
16    /// <returns>Taulukon alkioiden summa</returns>
17    /// <example>
18    /// <pre name="test">
19    ///    int[] luvut = {1,2,3};
20    ///    Summa(luvut) === 6;
21    ///    luvut = new int[0];
22    ///    Summa(luvut) === 0;
23    ///    Summa(new int[]{3,4}) === 7; // ilman apumuuttujia
24    ///    Summa([3, 4]) === 7;  // nykyisin voi kirjoittaa näinkin
25    /// </pre>
26    /// </example>
27    public static int Summa(int[] taulukko)
28    {
29        int summa = 0;
30        for (int i = 0; i < taulukko.Length; i++)
31            summa += taulukko[i];
32        //  summa = summa + taulukko[i]; // sama asia kuin yllä oleva
```

```
33          return summa;
34      }
```

Tutki animaatiolla miten taulukon viite viedään parametrina ja näin voidaan käyttää pääohjelman taulukkoa:10pt

Seuraavaksi esimerkki, jossa funktio muuttaa taulukkoa. Kaikki taulukon arvot, jotka ovat yli parametrina olevan rajan muutetaan rajaksi. Funktio palauttaa muutettujen lukujen lukumäärän. Testeissä on esimerkkejä miten taulukon sisältöä voidaan testata `String.Join` -funktion avulla.

```
1       public static void Main()
2       {
3           int[] luvut = { 2, 30, 15, 24, 5 };
4           int lkm = MuutaYli(luvut,20);
5           Console.WriteLine("Muutettu taulukko: " + String.Join(" ", luvut));
6           Console.WriteLine($"Muutettuja lukuja {lkm} kpl.");
7       }
8
9
10      /// <summary>
11      /// Aliohjelma muuttaa kaikki taulukun yli rajan olevat luvut raja-arvoon
12      /// </summary>
13      /// <param name="taulukko">taulukko jota muutetaan</param>
14      /// <param name="raja">raja-arvo jonka yli olevat muutetaan</param>
15      /// <returns>montako alkiota muutettiin</returns>
16      /// <example>
17      /// <pre name="test">
18      ///     int[] luvut = {1,2,4,2,5};
19      ///     MuutaYli(luvut,3) === 2;
20      ///     String.Join(" ", luvut) === "1 2 3 2 3";
21      ///     MuutaYli(luvut,3) === 0;
22      ///     MuutaYli(luvut,0) === 5;
23      ///     String.Join(" ", luvut) === "0 0 0 0 0";
24      ///     luvut = new int[0];
25      ///     MuutaYli(luvut,1) === 0;
26      /// </pre>
27      /// </example>
28      public static int MuutaYli(int[] taulukko, int raja)
29      {
30          int lkm = 0;
31          for (int i = 0; i < taulukko.Length; i++)
32          {
33              if ( taulukko[i] > raja )
34              {
35                  lkm++;
36                  taulukko[i] = raja;
37              }
38          }
39          return lkm;
40      }
```

Tutki ohjelman toimintaa alla olevalla animaatiolla, mutta kokeile myös askeltaa askel kerrallaan oman IDEsi debuggerissa ja mieti tarkkaan mitä tapahtuu.

Tutki animaation nuolella > miten taulukon viite viedään

- lisää taulukkoesimerkkejä

15.2.2 Muita taulukkoesimerkkejä

Tehtävä Tauno: muuta taulukon alkioiden arvoja

Lisää taulukon alkioihin juokseva luku. Ensimmäiseen alkioon lisätään 0, toiseen 1, seuraavaan 2 jne. Voit raahata arvoja vasemmalla alhaalla olevaan ”laskukoneeseen” ja sitten sieltä halumaasi paikkaan. Taulukosta voit raahata alkion vain paikasta, johon indeksimuuttuja i ”osoittaa”. Indeksimuuttujaa voit siirtää joko raahamalla sen uuteen kohtaan tai viemällä sen päälle +1 tai -1 operaattorit. Esim: taulukko 23 45 12 9 3 7 muuttuu taulukoksi 23 46 14 12 7 1210pt

```
1         int i = 0;
```

Animaatio: Suorita taulukko-ohjelmaa

Askella taulukkoesimerkkiä nuolella Tutki taulukkoa

Muokkaa ohjelma toimivaksi. 10pt

```
1  using System;
2  public class Taulukot
3  {
4     public static void Main()
5     {
6         int[] a = { 1, 1, 3, 4, 7, 9, 0 };
7         if (a[0].Equals(a[a.Length-1]))
8         {
9             Console.WriteLine("Nehän täsmää!");
10        }
11        else Console.WriteLine("Höh, ei ole samat!");
12    }
13 }
```

Alusta ensin kymmenpaikkainen taulukko t, jonka kaikki arvot ovat 0. Muuta sitten ensimmäisen ja viimeisen alkion arvo samaksi kuin taulukon pituus.10pt

Voit alla olevasta videosta katsoa näytelmän, jossa koetetaan havainnollistaa miten parametrin välitys mm. taulukoille toimii:

- Näytelmä parametrien välittämisestä ja taulukoista (36m9s) Esimerkin koodi: Kutsuminen.cs
- `Kutsuminen.cs` - voit katsoa tässäkin videolla käytettyä koodia

Sivulla luento12/syksy 2024 on ohjeet miten voit oman IDEsi debuggerin asetukset laittaa jos haluat debugata koodia samaan tahtiin tai jälkeenpäin omaan tahtiin.

Kutsuminen.cs

```
1  using System;
2
3  /// @author  Vesa Lappalainen
4  /// @version 12.10.2012
5  /// <summary>
6  /// Esimerkkjä aliohjelmien kutsumisesta
7  /// </summary>
8  public class Kutsuminen
9  {
10     /// <summary>
11     /// Kutsutaan aliohjelmia
12     /// </summary>
13     /// <param name="args">ei käytössä</param>
14     public static void Main(string[] args)
15     {
16         int a = 4, b = 5;
17         int iso;
18
19         iso = Suurempi(a, b);
20         Console.WriteLine(iso);
21
22         iso = Suurempi(iso+1, 7 / 3 + 5 / 2);
23         Console.WriteLine(iso);
24
25         int[] luvut = new int[4];
26         int[] m = luvut;
27
28         luvut[2] = 4;
29         luvut[1] = 6;
30         m[3] = iso+3;
31
32         VaihdaSuurin(luvut, 5);
33
34         Console.WriteLine(String.Join(" ",luvut));
35     }
36
37
38     /// <summary>
39     /// Palautetaan kahdesta luvusta suurempi
40     /// </summary>
41     /// <param name="luku1">eka luku</param>
42     /// <param name="luku2">toinen luku</param>
43     /// <returns>suurempi luvuista</returns>
44     /// <example>
45     /// <pre name="test">
46     ///   Suurempi(3,4) === 4;
47     ///   Suurempi(-3,4) === 4;
48     ///   Suurempi(-3,-4) === -3;
49     ///   Suurempi(2,2) === 2;
50     /// </pre>
51     /// </example>
52     public static int Suurempi(int luku1, int luku2)
53     {
54         if (luku1 >= luku2) return luku1;
55         return luku2;
56     }
```

```
57
58
59      /// <summary>
60      /// Vaihdetaan taulukon suurin alkio halutuksi.
61      /// </summary>
62      /// <param name="t">taulukko josta suurinta etsitään ja missä suurin ↩
        vaihdetaan</param>
63      /// <param name="korvaavaArvo">suurimman alkion uusi arvo</param>
64      /// <example>
65      /// <pre name="test">
66      ///   int[] luvut = {2,5,4,3};
67      ///   VaihdaSuurin(luvut,1);
68      ///   String.Join(" ",luvut) === "2 1 4 3";
69      ///   VaihdaSuurin(luvut,1);
70      ///   String.Join(" ",luvut) === "2 1 1 3";
71      ///   VaihdaSuurin(luvut,9);
72      ///   String.Join(" ",luvut) === "2 1 1 9";
73      ///   VaihdaSuurin(luvut,1);
74      ///   String.Join(" ",luvut) === "2 1 1 1";
75      ///   VaihdaSuurin(luvut,-1);
76      ///   String.Join(" ",luvut) === "-1 1 1 1";
77      ///   luvut = new int[]{5};
78      ///   VaihdaSuurin(luvut,2);
79      ///   String.Join(" ",luvut) === "2";
80      ///   luvut = new int[0];
81      ///   VaihdaSuurin(luvut,2);
82      ///   String.Join(" ",luvut) === "";
83      /// </pre>
84      /// </example>
85      public static void VaihdaSuurin(int[] t, int korvaavaArvo)
86      {
87          int paikka = -1;
88          int suurin = int.MinValue;
89          for (int i = 0; i < t.Length; i++)
90          {
91              int luku = t[i];
92              if (luku > suurin)
93              {
94                  suurin = luku;
95                  paikka = i;
96              }
97          }
98
99          if (paikka < 0) return;
100         t[paikka] = korvaavaArvo;
101     }
102
103 }
```

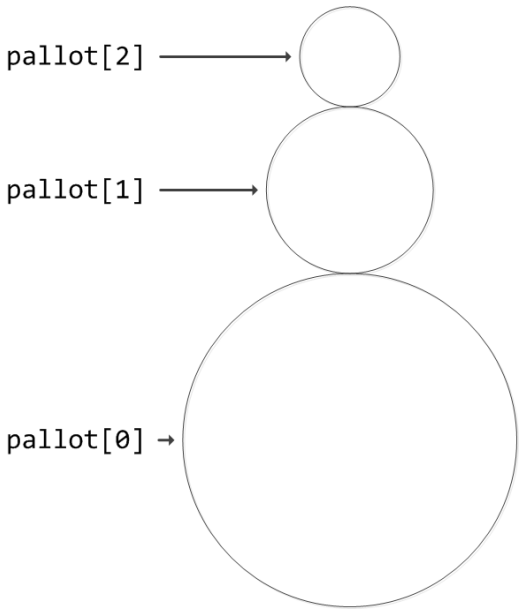
15.3 Esimerkki: lumiukon pallot taulukkoon

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

```
        // 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;
```

Tehtävä 15.1

Kokeile muidenkin pallojen värien vaihtamista edellä.10pt

```
        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.

Piirretään vielä kuvan miltä `arvosanat`-taulukko "oikeasti" näyttää. Arvosanat on siis viite taulukkoon, jossa on viitteitä merkkijonoihin. Toisaalta kukin merkkijono on oikeastaan merkkitaulukko, eli tämä on vastaava kuin myöhemmin esille tuleva *jagged-array*. Esimerkiksi sanan `hyvä` `y`-kirjain voitaisiin hakea lauseella:

```
char k = arvosanat[3][1];  // k == 'y';
```

Kuva: Arvosanat taulukko oikeasti

Toisaalta usein "laiskuuksissaan" nimenomaan `string`-taulukko piirretään ikäänkuin taulukon alkiot olisivat muuttujia. Tämä on osin puolusteltavissa sillä, että koska merkkijono on muuttumaton, niin tämä mielikuva ei kuitenkaan johda väärään tulkintaan koodin toiminnasta.

Kuva: Arvosanat taulukko hieman oikaisten.

On kuitenkin oltava varovainen sikäli, että jos taulukon alkiot olisivat esimerkiksi `StringBuilder`-luokan olioita (tai oikeastaan mitä tahansa muita olioita kuin `string`), niin kuva on ehdottomasti piirrettävä kuten ensimmäisessä kuvassa.

String vs. StrngBuilder -taulukkko

Kokeile ja mieti miksi toisessa taulukko muuttuu. Katso em. 1. kuva.10pt

```
using System;
using System.Text;

public class Arvosanat2
{
```

```
6       public static void Main()
7       {
8           string[] arvosanat = {"Hylätty", "Välttävä", "Tyydyttävä",
9                                 "Hyvä", "Kiitettävä", "Erinomainen"};
10          StringBuilder[] arvosanat2 = {new StringBuilder("Hylätty"),
11                                        new StringBuilder("Välttävä"),
12                                        new StringBuilder("Tyydyttävä"),
13                                        new StringBuilder("Hyvä"),
14                                        new StringBuilder("Kiitettävä"),
15                                        new StringBuilder("Erinomainen")};
16          string hyl = arvosanat[0];
17          StringBuilder hyl2 = arvosanat2[0];
18
19          hyl = hyl.Remove(3, 4);
20          hyl2.Remove(3, 4);
21
22          Console.WriteLine($"{arvosanat[0]} {arvosanat2[0]}");
23          // Tulostaa Hylätty Hyl
24      }
25
26  }
```

Muuta viikonpäivät-ohjelma käyttämään taulukkoa. (ks Tehtävä 13.3)10pt

15.5 Moniulotteiset taulukot

Taulukot voivat olla myös moniulotteisia. Kaksiulotteinen taulukko (eli matriisi) on esimerkki moniulotteisesta taulukosta, joka koostuu vähintään kahdesta samanpituisesta taulukosta.

Esimerkiksi kirja voitaisiin kuvitella 3-ulotteiseksi taulukoksi, koska tietyn sanan paikan ilmoittamiseksi pitäisi antaa kolme “indeksiä”:

```
- sivunumero
- rivinumero
- monesko rivin sana
```

Tai jopa 4-ulotteiseksi jos haluttaisiin ilmaista tietyn kirjaimen paikka:

```
- sivunumero
- rivinumero
- monesko rivin sana
- monesko sanan kirjain
```

Toki voitaisiin riviltä jo ilmoittaa monesko kirjain ja silloin kirjain olisi tällä tavalla alkio 3-ulotteisessa taulukossa. Tietokoneessa ei ole kuin peräkkäisiä muistipaikkoja, eli tietokoneen muisti on tässä mielessä yksiulotteinen taulukko. Mutta tuolla “kirjaimen paikka rivillä” vaiko “sanan paikka rivillä + kirjaimen paikka sanassa” -idealla voidaan yksiulotteisia taulukoita tulkita useampiulotteisiksi, ja näin toteuttaa tietokoneessa useampiulotteisia taulukoita.

Sitten jos ajateltaisiin yhtä kirjahyllyn hyllyä, niin silloin kirjaimen ilmoittamiseen tarvittaisiin 5 indeksiä, edellisten luettelon alkuun vielä kirjan paikka hyllyllä. Jos otetaan koko kirjahylly, niin kirjaimen ilmoittamiseen vielä pitäisi lisätä hyllyn numero. Eli meillä olisi 6-ulotteinen taulukko. Jos kirjahyllyjä olisi monta, se olisi 7-ulotteinen taulukko. Ja tätä voidaan jatkaa eteenpäin niin paljon kuin halutaan. Kadun jokaisessa talossa voi myös olla kirjahyllyjä. Toki

tämän esimerkin huono puoli on se että esim kirjan rivit eivät ole yhtäpitkiä keskenään, tosin siitäkin selvitään myöhemmin esiteltävällä jagged-array idealla.

Tee sijoituksia taulukkoon

Alla olevan kuvan taulukkoon `t1` olisi voitu laittaa tuo luku

```
t[0,3] = 9
```

Huomaa tässä ero tyypillisiin geometrisiin koordinaatteihin, joissa ensin sanotaan x-suunta ja siten y-suunta. Kirjavertauksella tai kerrostalovertauksella kuitenkin rivi tai kerros on tärkeämpi kuin "sarake", siksi **taulukoissa "y"-suunta tulee ensin**. Lisäksi vielä "y-koordinaatti" kasvaa "alaspäin", toisin kuin geometrian y-koordinaatti. Muutenkin on aina oltava tarkkana miten eri `koordinaatistot` toimivat. Maantieteellisissä koordinaateissa y-suunta tulee ensin, esimerkiksi Agora on koordinaatissa: 62.232, 25.737 (pohjoinen kulma päiväntasaajalta, itäinen kulma Greenwichin nollameridiaanista). Jypelissä käytetään geometrista koordinaatistoa oletuksena.

Samalla idealla esimerkiksi yksi kirjain voitaisiin pyytää meidän 7-ulotteisesta kirjahyllyjen joukosta lauseella:

```
char k = kirjaimet[2,3,7,123,17,6,4];
```

jos olisi:

- 2 = kirjahyllyn numero
- 3 = hyllyrivin numero
- 7 = kirjan numero hyllyllä
- 123 = sivunumero
- 17 = rivinumero
- 6 = monesko rivin sana
- 4 = monesko sanan kirjain

Taulukon ulottuvuus on siis se, montako "indeksiä" tarvitaan, jotta päästään haluttuun alkioon käsiksi. Indeksi ovat vasemmalta alkaen eniten määräävässä järjestyksessä. Eli kun aletaan etsimään tiettyä kirjainta meidän "kirjastosta", niin toki pitää ensin mennä oikean kirjahyllyn luokse, sitten siinä oikealle hyllyriville jne.

- 2-ulotteisen taulukon indeksit Luento 11, syksy 2024 ndash; 1h17m59s (3m11s)

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[,] opiskelijat = new String[256, 2];
```

Taulukkoon voisi nyt asettaa kurssilaisten nimiä seuraavasti:

```
//ensimmäinen kurssilainen
opiskelijat[0, 0] = "Virtanen";
opiskelijat[0, 1] = "Ville";
//toinen kurssilainen
opiskelijat[1, 0] = "Korhonen";
opiskelijat[1, 1] = "Kalle";
```

Taulukko näyttäisi nyt seuraavalta:

Kuva 20: `opiskelijat`-taulukko hieman oikaisten.

Pitää taas kuitenkin muistaa, että edellinen kuva on "riittävä" vain jos alkiot ovat `string`-olioita. Jos alkiot ovat perustietotyyppisiä, niin kuva on nimenomaan kuten edellä. Mutta olioille siis kuten alla:

Kuva 20: `opiskelijat`-taulukko oikeasti.

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

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

Edellinen tulostus voitaisiin $-merkkijonolla tehdä myös:

```
Console.WriteLine($"{opiskelijat[0,1]} {opiskelijat[0,0]}");
```

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

```
// tulostaa Virtanen, Ville
Console.WriteLine(opiskelijat[0,0] + ", " + opiskelijat[0,1]);
// tai
Console.WriteLine($"{opiskelijat[0,0]}, ${opiskelijat[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.

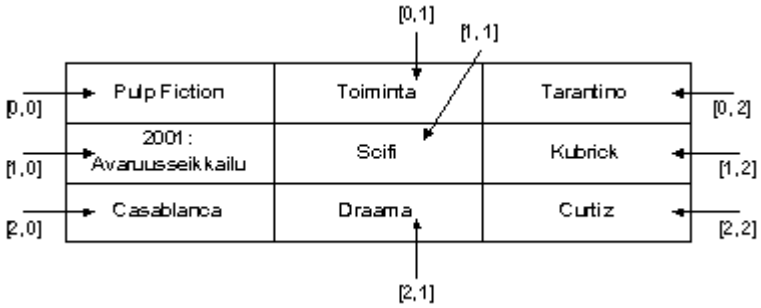
Animaatio: Suorita ohjelma

Askella moniulotteiseen taulukkoon viittaaminen vihreällä nuolella. Huom tässä Java-esimerkissä 2-ulotteinen taulukko on tehty taulukkona taulukoista (jagged array). Tutki moniulotteista taulukkoa

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

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

Yllä oleva määrittely luo 3 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.

```
1         string[,] elokuvat =  {
2           {"Pulp Fiction", "Toiminta", "Tarantino"     },
3           {"2001: Avaruusseikkailu", "Scifi", "Kubrick"},
4           {"Casablanca", "Draama", "Curtiz"            }
5         };
6
7         Console.WriteLine(elokuvat[0, 0]);  // "Pulp Fiction"
8         Console.WriteLine("Tyyppi: " + elokuvat[0, 1]); // "Tyyppi:  Toiminta"
9         Console.WriteLine("Ohjaaja: " + elokuvat[0, 2]); // "Ohjaaja: Tarantino"
```

Kuten edellä opiskelija-taulukon kohdalla, tämäkin esimerkki on sikäli hieman väärin, että taulukon eri alkiot eivät tarkoita samaa asiaa. Kun seuraavilla kursseilla perehdytään tietueisiin ja olioihin, ne olisivat oikeampi tapa ratkaista opiskelijoiden ja elokuvien tallentaminen taulukkoon.

Tällä tavalla jokaiselle riville tulee yhtä monta saraketta eli alkiota. Jos eri riveille halutaan eri määrä alkioita, voidaan käyttää niin sanottuja *jagged array* -taulukkoja. Eli moniulotteinen taulukko tehdään taulukkona taulukoista. Lue lisää *jagged arrayn* MSDN-dokumentaatiosta osoitteesta https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/jagged-arrays.

Jagged-array: Taulukoita taulukoista.

Erona tavalliseen 2-ulotteiseen taulukkoon on se, että ensin ilmoitetaan

```
t[0][3] = 9
```

15.5.1 Harjoitus

Miten tulostat taulukosta Casablanca? Entä Kubrick?10pt

```
        Console.WriteLine(elokuvat[0, 0]);
        Console.WriteLine("Tyyppi: " + elokuvat[0, 1]);
        Console.WriteLine("Ohjaaja: " + elokuvat[0, 2]);
```

Etsi moniulotteisesta taulukosta sanoja ja tulosta ne. Mitä tapahtuu jos käytät tulostuslausetta System.Console.WriteLine(ristikko[0,0] + ristikko[1,1]); Yksi sana on mallina. 10pt

```
//
        char[,] ristikko = {
           {'M', 'V', 'O', 'I', 'D', 'S', 'T', 'I'},
           {'U', 'C', 'K', 'O', 'O', 'D', 'I', 'A'},
           {'U', 'H', 'K', 'N', 'L', 'N', 'M', 'E'},
           {'T', 'A', 'U', 'N', 'O', 'I', 'I', 'L'},
           {'T', 'R', 'L', 'I', 'N', 'T', 'O', 'B'},
           {'U', 'M', 'U', 'Y', 'S', 'L', 'K', 'U'},
           {'J', 'E', 'A', 'E', 'H', 'T', 'O', 'O'},
           {'A', 'S', 'T', 'R', 'I', 'N', 'G', 'D'},
        };
        System.Console.WriteLine("" + ristikko[0,1] + ristikko[0,2] +
                                       ristikko[0,3] + ristikko[0,4]);
```

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
        Console.WriteLine(taulukko2[0]); //tulostaa 10
```

Tutki animaatiolla miten samaan taulukkoon on kaksi viitettä. Tämä on idea miten taulukoita viedään parametrina.10pt

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

Taulukon aito kopioiminen onnistuu muun muassa Clone-metodilla.

```
        int[] taulukko1 = {1, 2, 3, 4, 5};
        // Clone-metodi luo identtisen kopion taulukosta
        int[] taulukko2 = (int[])taulukko1.Clone();

        taulukko2[0] = 10;
        Console.WriteLine(taulukko1[0]); // tulostaa 1
        Console.WriteLine(taulukko2[0]); // tulostaa 10
```

Huomaa, että sijoituksessa vaaditaan ns. tyyppimuunnos: ennen `taulukko1.Clone`-lausetta kirjoitetaan `(int[])`, sulkujen kanssa, joka muuttaa `Clone`-metodin palauttaman "yleisen" Object-olion kokonaislukutaulukoksi.

Tutki animaatiolla miten nyt taulukot ovat eri viitteitä.10pt

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

Tässä muutetaan taulukon alkion arvoa aliohjelmasta. Aja ohjelma ja katso koko koodia. Koska meillä on viite yhteiseen taulukkoon, taulukkoa ei tarvitse mitenkään palauttaa tai sijoittaa, jotta muutos näkyisi. 10pt

```
1  using System;
2
3  public class Kasvata
4  {
5      public static void Main()
6      {
7          int[] arvosanat = {1, 2, 3, 4, 5};
8          System.Console.WriteLine(String.Join(" ",arvosanat));
9          KasvataAlkiota(arvosanat, 0);
10         System.Console.WriteLine(String.Join(" ",arvosanat));
11     }
12
13
14     public static void KasvataAlkiota(int[] taulukko, int alkio)
15     {
16        taulukko[alkio] +=1;
17     }
18 }
```

Tutki animaation nuolella > miten taulukon viite viedään

Seuraavassa kappaleessa opetellaan tekemään sama kaikenkokoisille taulukoille ja tulostamaan taulukko järkevämmin.

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.

```
1  using System;
2
3  /// @author  Antti-Jussi Lakanen
4  /// @version 22.8.2012
5  ///
6  /// <summary>
7  /// Moniulotteiset taulukot käytännössä.
8  /// </summary>
9  public class Laivanupotus
10 {
```

```
11    /// <summary>
12    /// Taulukon alustus ja tulostus.
13    /// </summary>
14    public static void Main()
15    {
16      int[,] ruudut = { { 1, 0, 2 }, { 0, 0, 3 } };
17      Console.WriteLine(ruudut[0, 0] + " " + ruudut[0, 1] + " " + ruudut[0, 2]);
18      Console.WriteLine(ruudut[1, 0] + " " + ruudut[1, 1] + " " + ruudut[1, 2]);
19      // Tulostaa:
20      // 1 0 2
21      // 0 0 3
22    }
23  }
```

Vaikka ruudukko oli vielä aika pieni (2 riviä x 3 saraketta), on alkioiden tulostaminen melko työlästä. Esimerkiksi 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, jotka esitellään seuraavassa luvussa.

Katso seuraava video ja tehtävät sen jälkeen, kun olet tutustunut silmukka-lukuun.

- Matriisia käsittelevä funktio Luento 12, syksy 2024 ndash; 1h23m52s (10m16s)

MiltaRiviltaLoytyy

Animoi etsimistä >-nuolella. Katso huolella miten iy ja

Täydennä funktio `MiltaRiviltaLoytyy` samalla kun katsot videota.

```
1  using System;
2  using System.Text;
3
4  /// @author  Vesa Lappalainen
5  /// @version 10.10.2011
6  ///
7  /// <summary>
8  /// Esimerkkejä matriisista
9  /// </summary>
10 public class Matriisit
11 {
12     /// <summary>
13     /// Esimerkkejä matriiseista
14     /// </summary>
15     /// <param name="args">ei käytössä</param>
16     public static void Main(string[] args)
17     {
18         // int[] luvut = int[3];
19         int[] luvut = { 3, 5, 1, 9 };
20
21         int[,] matriisi1 = new int[23, 5];
```

```
22
23          int[,] matriisi = {
24                                      {1,2,3},
25                                      {4,5,6},
26                                      {7,8,9},
27                                      {10,11,12}
28                                  };
29
30          // Tulosta(matriisi);
31
32          Console.WriteLine("Dimensio: " + matriisi.Rank);
33          Console.WriteLine("Rivejä: " + matriisi.GetLength(0));
34          Console.WriteLine("Sarakkeita: " + matriisi.GetLength(1));
35          Console.WriteLine("Paikassa 2,1: " + matriisi[2, 1]);
36
37          // int rivi = MiltaRiviltaLoytyy(matriisi, 8);
38          // Console.WriteLine("rivi = " + rivi);
39
40          matriisi1[6, 2] = 99;
41
42          // Tulosta(matriisi1);
43      }
44
45   // TÄYDENNÄ
46
47 }
```

Pelikentän esittäminen kirjaintaulukolla, käytännön esimerkki 2-ulotteisen taulukon hyödyistä.

- Esimerkki pelikentän määrittelystä kirjaintaulukolla Luento 11, syksy 2024 ndash; 1h28m30s (2m32s)

Kuva: Olioiden sijoittelu

```
1 ref kentta
2 array $1 c4x3 P.P|.On|n.P|O P
3 kentta -> $1
```

Esimerkki jossa pelikenttä luodaan 2-ulotteisen kirjaintaulukon pohjalta. Jos taulukossa on monta pelaajan merkkiä (.), niin vain viimeistä voi liikutella.10pt

```
1 using System;
2 using Jypeli;
3
4 /// @author Vesa Lappalainen
5 /// @version 10.10.2020
6 ///
7 /// <summary>
8 /// Ohjelmassa harjoitellaan 2-ulotteisen
9 /// merkkitaulukon (char[,]) hyodyntamista kentan
10 /// luomisessa.
11 /// </summary>
12 public class KentanMuokkaus : PhysicsGame
13 {
14     private PhysicsObject pelaaja; // Liikuteltava pallo, jota pystytään
15                                    // liikuttamaan kaikista tämän luokan ↩
      aliohjelmista
```

```
16      const int KOKO = 20; // paljonko on yksittäisen palikan koko.
17                           // huom const eli vakio! (ei voi muuttaa)
18
19      /// <summary>
20      /// Asetetaan painovoima ja kutsutaan alustuksiin
21      /// liittyvia aliohjelmia. Aloitetaan peli.
22      /// Zoomataan kamera.
23      /// </summary>
24      public override void Begin()
25      {
26          // SetWindowSize(1000, 700);
27          Level.Width = 100;
28          Level.Height = 100;
29          char[,] kentta = {
30                  {'P', '.', 'P'},
31                  {'.', 'O', 'n'},
32                  {'n', '.', 'P'},
33                  {'O', ' ', 'P'},
34                           };
35          // Gravity = new Vector(0, -1000);
36          TeeKentta(kentta);
37          AsetaOhjaimet();
38          Level.CreateBorders();
39          Camera.ZoomToLevel();
40      }
41
42
43      /// <summary>
44      /// Luodaan pelikentta 2-ulotteisen merkkitaulukon (char[,])
45      /// avulla. Kutsutaan aliohjelmia eri merkeille luotujen merkitysten
46      /// perusteella.
47      /// </summary>
48      private void TeeKentta(char[,] kentta)
49      {
50          double alkuX = Level.Left + KOKO / 2 + 20;
51          double alkuY = Level.Top - KOKO / 2;
52
53
54          double y = alkuY;
55
56          // Kaydaan rivit ja sarakkeet lapi.
57          // "Muutetaan" merkkitaulukko kentaksi siten
58          // etta tietyt merkit vastaavat tietyn kentalle lisattavan olion
59          // aliohjelmakutsua. Myos koordinaatit saadaan matriisista:
60          // Rivi toimii x-koordinaattina, ja sarake y-koordinaattina.
61          for (int iy = 0; iy < kentta.GetLength(0); iy++)
62          {
63              double x = alkuX;
64              for (int ix = 0; ix < kentta.GetLength(1); ix++)
65              {
66                  char t = kentta[iy, ix];
67
68                  switch (t)
69                  {
70                      case 'P':
71                          LuoPalikka(x, y);
72                          break;
73                      case 'n':
74                          LuoNelio(x, y);
```

```
75                          break;
76                      case 'O':
77                          LuoYmpyra(x, y);
78                          break;
79                      case '.':
80                          LuoPelaaja(x, y);
81                          break;
82                  }
83                  x += KOKO;
84              }
85              y -= KOKO;
86          }
87      }
88
89
90      /// <summary>
91      /// Lisataan pallo kentalle ja annetaan pallolle massa.
92      /// </summary>
93      /// <param name="x">Pallon kp x-koordinaatti</param>
94      /// <param name="y">Pallon kp y-koordinaatti</param>
95      private void LuoPelaaja(double x, double y)
96      {
97          pelaaja = new PhysicsObject(KOKO / 2, KOKO / 2, Shape.Circle);
98          pelaaja.Position = new Vector(x, y);
99          pelaaja.Color = Color.Red;
100         Add(pelaaja);
101     }
102
103
104     /// <summary>
105     /// Lisataan staattinen suorakulmio (nelio) kentalle
106     /// annettuun paikkaan.
107     /// </summary>
108     /// <param name="x">Nelion kp x-koordinaatti</param>
109     /// <param name="y">Nelion kp y-koordinaatti</param>
110     private void LuoPalikka(double x, double y)
111     {
112         PhysicsObject palikka = PhysicsObject.CreateStaticObject(KOKO, KOKO);
113         Add(palikka);
114         palikka.Image = LoadImage("tiili2");
115         palikka.Position = new Vector(x, y);
116     }
117
118
119     /// <summary>
120     /// Lisataan pieni staattinen suorakulmio (nelio) kentalle
121     /// annettuun paikkaan.
122     /// </summary>
123     /// <param name="x">Nelion kp x-koordinaatti</param>
124     /// <param name="y">Nelion kp y-koordinaatti</param>
125     private void LuoNelio(double x, double y)
126     {
127         PhysicsObject palikka = PhysicsObject.CreateStaticObject(KOKO / 2, KOKO /↩
        2);
128         Add(palikka);
129         palikka.Color = Color.Blue;
130         palikka.Position = new Vector(x, y);
131     }
132
```

```
133
134     /// <summary>
135     /// Tehdaan staattinen ympyra
136     /// ja lisataan se pelikentalle
137     /// annettuun paikkaan.
138     /// </summary>
139     /// <param name="x">Ympyran kp x-koordinaatti</param>
140     /// <param name="y">Ympyran kp y-koordinaatti</param>
141     private void LuoYmpyra(double x, double y)
142     {
143         PhysicsObject pallo = PhysicsObject.CreateStaticObject(KOKO, KOKO, Shape.↩
        Circle);
144         this.Add(pallo);
145         pallo.Color = Color.Yellow;
146         pallo.Position = new Vector(x, y);
147     }
148
149
150     /// <summary>
151     /// Asetetaan nappaintenkuuntelijat.
152     /// </summary>
153     private void AsetaOhjaimet()
154     {
155         const int voima = 100;
156         Keyboard.Listen(Key.Up, ButtonState.Pressed, LiikutaPalloa, "Ylös", new ↩
        Vector(0, voima));
157         Keyboard.Listen(Key.Down, ButtonState.Pressed, LiikutaPalloa, "Alas", new↩
         Vector(0, -voima));
158         Keyboard.Listen(Key.Left, ButtonState.Pressed, LiikutaPalloa, "Vasemmalle↩
        ", new Vector(-voima, 0));
159         Keyboard.Listen(Key.Right, ButtonState.Pressed, LiikutaPalloa, "Oikealle"↩
        , new Vector(voima, 0));
160
161         // Ohjeiden näyttämistä varten. Paina F1 pelin aikana.
162         Keyboard.Listen(Key.F1, ButtonState.Pressed, ShowControlHelp, "Ohjeet");
163
164         Keyboard.Listen(Key.Escape, ButtonState.Pressed, Exit, "Poistu");
165     }
166
167     /// <summary>
168     /// Aliohjelma liikuttaa palloa.
169     /// </summary>
170     /// <param name="suunta">Suunta, johon palloa liikutetaan.</param>
171     private void LiikutaPalloa(Vector suunta)
172     {
173         pelaaja.Hit(suunta);
174     }
175
176 }
```

Huomaa että Jyplissä on valmiina rakenteita pelikentän tekemiseksi tämän idean pohjalta, eli katso:

- Kentän luominen merkkijonotaulukosta (yleensä suositeltavin)
- Kentän tekeminen tekstitiedostosta
- Kentän tekeminen kuvatiedostosta

15.8 Taulukoiden täydennykset lisätietosivuilla

Lisätietoja 1- ja 2-ulotteisista taulukoista löydät kurssin lisätietosivulta. Lue myös nuo materiaalit läpi (kuuluvat tenttialueeseen).

- 1-ulotteiset taulukot
- 2-ulotteiset taulukot

Tehtävä 15.2

Selitä seuraaavat termit: a) Taulukko, b) Merkkijono, c) Alkeistietotyyppi, d) Lokaali muuttuja, e) Olio f) Matriisi 10pt

Luku 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 `kkPaivat`-taulukon luvut, onnistuisi se tietenkin seuraavasti:

```
1         Console.WriteLine(kkPaivat[0]);
2         Console.WriteLine(kkPaivat[1]);
3         Console.WriteLine(kkPaivat[2]);
4         Console.WriteLine(kkPaivat[3]);
5         Console.WriteLine(kkPaivat[4]);
6         Console.WriteLine(kkPaivat[5]);
7         Console.WriteLine(kkPaivat[6]);
8         Console.WriteLine(kkPaivat[7]);
9         Console.WriteLine(kkPaivat[8]);
10        Console.WriteLine(kkPaivat[9]);
11        Console.WriteLine(kkPaivat[10]);
12        Console.WriteLine(kkPaivat[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 `kkPaivat`-taulukon kaikki luvut.
```

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

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

C#:ssa on neljä erilaista silmukkarakennetta:

- 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.

> Kun jatkossa käsitellään silmukoita, niin niissä kaikissa on jossakin kohti ehtolauseke. Silmukassa tulee olla harvoja poikkeuksia lukuunottamatta lause/lauseita, joka muuttaa muuttujia sillä tavalla, että ehto tulee joskus epätodeksi, jotta silmukka saadaan päättymään. Yleensä tämä lause on tämän kurssin esimerkeissä tyyliin `i++`, mikäli ehto on muotoa `(i < ylaraja)`.

16.2 while-silmukka

while-silmukka on yleisessä muodossa seuraava:

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

Kuten ehtolauseissa, täytyy ehdon taas olla jokin 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.

Huomaa että vähintään yhden suoritettavan lauseen on syytä olla sellainen, että se muuttaa ehdon arvoa niin, että siitä voi joskus tulla `false`.

16.2.1 Kohti silmukkaa

Otetaan esimerkki, missä meillä olisi lukuja jotka pitää laskea yhteen. Tässä vaiheessa emme vielä ota kantaa siitä, mistä näitä lukuja saadaan. Oikeasti niitä saadaan joltakin mittalaitteelta tai luetaan esimerkiksi tiedostosta. Ohjelman ensimmäinen versio voisi olla:

```
//
        int t0=10, t1=7, t2=5, t3=6;
        int summa = 0;
        summa += t0;  // sama kuin summa = summa + t0;
        summa += t1;
        summa += t2;
        summa += t3;
        System.Console.WriteLine("summa on " + summa);
```

Oikeasti aina kun on joukko muuttujia, jotka ovat tietyssä mielessä samanarvoisia, ne kannattaa laittaa johonkin tietorakenteeseen, esimerkiksi taulukkoon tai listaan. Taulukolla tehtynä edellinen ohjelma olisi:

```
//
        int[] t={10, 7, 5, 6};
        int summa = 0;
        summa += t[0];  // sama kuin summa = summa + t[0];
        summa += t[1];
        summa += t[2];
        summa += t[3];
        System.Console.WriteLine("summa on " + summa);
```

Tässä on vielä se vika, että jos taulukkoon laitetaan lisää lukuja, lasketaan edelleen noiden 4 ensimmäisen summa. Tai mikäli lukuja vähennetään, viitataan alkioon, jota ei olemassa.

Aloitetaan kohti silmukkaa meneminen niin, että yritetään aluksi saada kaikista riveistä keskenään täsmälleen samanlaisia.

Aloitetaan esittelemällä indeksimuuttuja `i`, joka kirjoitetaan sitten aluksi indeksivakion `0` tilalle.

```
//
        int[] t={10, 7, 5, 6};
        int summa = 0;
        int i = 0;

        summa += t[i];   // sama kuin summa = summa + t[i]
        summa += t[1];
        summa += t[2];
        summa += t[3];
        System.Console.WriteLine("summa on " + summa);
```

Eli kun `i=0`, niin on sama kirjoitetaanko `t[0]` vaiko `t[i]`. Mutta mikäli seuraava rivi korvattaisiin myös `t[i]`, niin silloin se ei olisikaan sama kuin `t[1]`. Ellei sitten `i`:tä kasvateta ennen rivin suorittamista. Ja sama pätee seuraavaankin riviin. Eli seuraava muutettu ohjelma tekee saman kuin alkuperäinen ohjelma:

```
//
        int[] t={10, 7, 5, 6};
        int summa = 0;
        int i = 0;

        summa += t[i]; i++;   // jotta seuraavalla rivillä i == 1
        summa += t[i]; i++;   // jotta seuraavalla rivillä i == 2
        summa += t[i]; i++;   // jotta seuraavalla rivillä i == 3
        summa += t[i]; i++;   // i++ ei tarpeen, mutta nyt rivit samanlaisia
```

```
10          System.Console.WriteLine("summa on " + summa);
```

Nyt olemme saneet kaikista riveistä täsmälleen samanlaisia. Tämä on yksi tapa lähestyä silmukoita. Ensin tehdään asia niin yksinkertaisesti kuin se osataan, sitten yritetään saada melkein samanlaisina toistuvista riveistä täsmälleen samanlaisia.

Tästä on nyt helpompi päästä silmukkaan:

```
1  //
2          int[] t={10, 7, 5, 6};
3          int summa = 0;
4          int i = 0;
5
6          while ( i < 4 )
7          {
8              summa += t[i]; i++;
9          }
10         System.Console.WriteLine("summa on " + summa);
```

Tosin yleensä on tapana kirjoittaa jokainen rivi omalle rivilleen ja mieluummin kuin käyttää vakiota `4`, käytetään sen tilalla taulukoiden alkioiden lukumäärää:

```
1  //
2          int[] t={10, 7, 5, 6};
3          int summa = 0;
4          int i = 0;
5
6          while ( i < t.Length )
7          {
8              summa += t[i];
9              i++;
10         }
11         System.Console.WriteLine("summa on " + summa);
```

Nyt ohjelma toimii vaikka taulukkoa muutettaisiin lisäämällä tai poistamalla siitä alkioita. Kokeile!

Silmukan ansiosta meidän ei enää tarvitse välittää siitä, montako alkiota taulukossa on. Ja `while`-silmukka toimii, vaikka taulukko olisi tyhjäkin, koska silloin heti alussa ehto `i < t.Length` on siis sama kuin `i < 0` eli on epätosi.

16.2.2 Esimerkkejä while-silmukoista

Animaatio: Suorita while-silmukkaa

Askella silmukan suoritusta vihreällä nuolella Tutki while-silmukan toimintaa

While-silmukan malliohjelma Luento 8 (11m11s)

While-silmukan malliohjelman debuggaus Luento 8 (4m11s)

Muokkaa ohjelma toimivaksi. Laita pääohjelma ennen muita aliohjelmia. 10pt

```
1  using System;
2  public class Whileloop
3  {
```

```
4     public static void Main()
5     {
6         string lause = "Poukkoileva siili ja kaunis orava kävelivät tien yli";
7         while (lause.Length > 1)
8         {
9             Console.WriteLine(lause = lause.Substring(0, lause.Length-1));
10        }
11    }
12 }
```

16.2.3 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.

Esimerkkinä seuraava tulostaisi ikuisesti merkkiä 0. Ei tule edes rivinvaihtoja koska käytetään `Write`-aliohjelmaa, vaan 0:t tulostuvat yhteen pötköön.

```
while (true)
{
    System.Console.Write("0"); // virhe
}
```

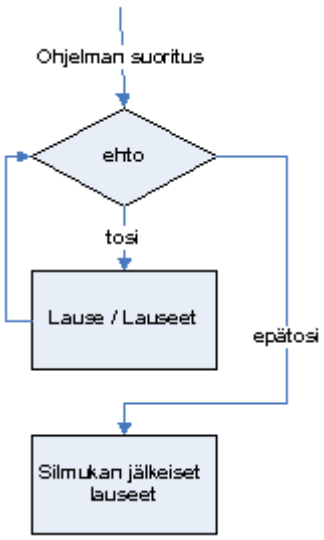
Silmukka voidaan katkaista `break`-lauseella. Useimmiten `break` on alla olevasta esimerkistä poiketen jonkin ehtolauseen takana.

```
while (true)
{
    System.Console.Write("0");
    break;
}
```

Silmukka voidaan lopettaa muuttamalla ehto epätodeksi. Vakioehtoa `true` ei voi muuttaa epätodeksi, joten tehdään uusi muuttuja jossa ehto on. Tosin useimmiten tällaiset lippumuuttujat ovat huono ratkaisu, koska pidemmästä koodista on vaikea huomata missä ne muuttuvat.

```
bool ehto = true;
while (ehto)
{
    System.Console.Write("0");
    ehto = false;
}
```

16.2.4 while-silmukka vuokaaviona



Kuva 22: while-silmukka vuokaaviona.

16.2.5 Esimerkki: Taulukon tulostaminen

Tehdään aliohjelma, joka tulostaa int-tyyppisen yksiulotteisen taulukon sisällön. Luento 10 (6m16s)

```
1  using System;
2
3  public class Silmukat
4  {
5    /// <summary>
6    /// Tulostaa int-tyyppisen taulukon sisällön.
7    /// </summary>
8    /// <param name="taulukko">Tulostettava taulukko</param>
9    public static void TulostaTaulukko(int[] taulukko)
10   {
11     int i = 0;
12     while (i < taulukko.Length){
13       Console.Write(taulukko[i] + " ");
14       i++;
15     }
16     Console.WriteLine();
17   }
18
19   /// <summary>
20   /// Pääohjelma.
21   /// </summary>
22   public static void Main()
23   {
24     int[] kkPaivat = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
25     TulostaTaulukko(kkPaivat);
26   }
27 }
```

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.

Tässä tapauksessa silmukan jälkeen on syytä käyttää `WriteLine()`-kutsua jotta mahdollinen seuraava tulostus jatkuisi omalta riviltään. Kokeile mitä tapahtuu jos vaihdat `Write` tilalle `WriteLine`.

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 tehdä for-silmukalla, missä vaara ikuisen silmukan syntymiseen on pienempi. Myöhemmin löytyy järkevämpää käyttöä while-silmukalle.

Seuraavassa animaatio, joka tulostaa luvun numeroiden summan:

Animaatio: Suorita while-silmukkaa

Askella silmukan suoritusta vihreällä nuolella Tutki while-silmukan toimintaa

Tehtävä 16.1

Tee kokonainen ohjelma, jossa tulostetaan luvut 0-50 käyttäen while-silmukkaa. Varmista ettei silmukka ole ikuinen silmukka. 10pt

Tehtava 16.2

Valitse `Näytä koko koodi`. Tee aliohjelma

```
1
2         /// <summary>
3         /// Jakaa annettua lukua kunnes se on
4         /// pienempi tai yhtäsuuri kuin raja
5         /// </summary>
6         /// <param name="aloitusluku">luku jota aletaan jakamaan</param>
7         /// <param name="raja">luku johon tai alle pitäisi päästä</param>
8         /// <returns>tarvittavien jakojen lukumäärä</returns>
9         /// <example>
10        /// <pre name="test">
11        ///     Jakokerrat(100000, 1) === 16;
12        /// </pre>
13        /// </example>
14        ??? ??? ??? Jakokerrat(???, ???)
15        {
16        }
```

16.3

Täydennä aliohjelma toimimaan ohjeiden mukaisesti.10pt

```
1    /// <summary>
2    /// Palauttaa merkkijonotaulukon, jonka alkiot on lyhennetty maxpituuteen
3    /// </summary>
4    /// <param name="merkkijonot">taulukko</param>
5    /// <param name="maxpituus">maxpituus</param>
6    /// <returns>Merkkijono-taulukko, jonka alkiot on <= maxpituus</returns>
7    public static string[] LyhennaLiianPitkat(string[] merkkijonot, int maxpituus)
8    {
9        string[] kopiomj = (string[])merkkijonot.Clone();
10       return kopiomj;
11   }
```

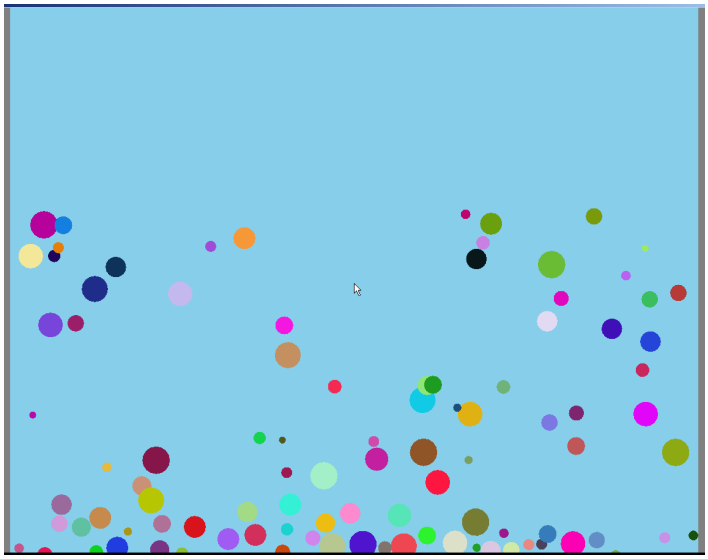
16.2.6 Esimerkki: Monta palloa

Tehdään aliohjelma, joka luo halutun kokoisen pallon haluttuun paikkaan ja vielä halutulla värillä. Main-metodi on jätetty listauksesta pois. Luento 8 (9m44s)

```
1  using System;
2  using Jypeli;
3
4  /// <summary>
5  /// Paljon palloja tippuu alaspäin.
6  /// </summary>
7  public class Peli : PhysicsGame
8  {
9      /// <summary>
10     /// Ruudulla näkyvä sisältö.
11     /// </summary>
12     public override void Begin()
13     {
14         Level.CreateBorders();
```

```
15          Gravity = new Vector(0, -500);
16          Camera.ZoomToLevel();
17
18
19          int i = 0;
20          while (i < 100)
21          {
22              int sade = RandomGen.NextInt(5, 20);
23              double x = RandomGen.NextDouble(Level.Left + sade, Level.Right - sade↩
        );
24              double y = RandomGen.NextDouble(Level.Bottom + sade, Level.Top - sade↩
        );
25              Color vari = RandomGen.NextColor();
26              PhysicsObject pallo = LuoPallo(x, y, vari, sade);
27              Add(pallo);
28              i++;
29          }
30      }
31
32      /// <summary>
33      /// Luo yksittäisen pallon ja palauttaa sen.
34      /// </summary>
35      /// <param name="x">Pallon kp x-koordinaatti</param>
36      /// <param name="y">Pallon kp y-koordinaatti</param>
37      /// <param name="vari">Pallon väri</param>
38      /// <param name="sade">Pallon säde</param>
39      public static PhysicsObject LuoPallo(double x, double y, Color vari, double ↩
        sade)
40      {
41          PhysicsObject pallo = new PhysicsObject(2 * sade, 2 * sade, Shape.Circle)↩
        ;
42          pallo.Color = vari;
43          pallo.X = x;
44          pallo.Y = y;
45          return pallo;
46      }
47  }
```

Ajettaessa koodin tulisi piirtää ruudulle sata palloa, jotka putoavat alaspäin kohti kentän reunaa (kun ajetaan tietokoneessa, TIMissä tulee vain 2 sek videopätkä). Katso seuraava kuva.



Kuva 23: Pallot tippuu.

Tutkitaan tarkemmin LuoPallo-aliohjelmaa. Aliohjelma palauttaa PhysicsObject-olion, siis paluuarvon tyyppinä on luonnnollisesti 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 parametreina juuri tekemämme muuttujat LuoPallo-aliohjelmalle, joka sitten palauttaa haluamamme pallon. Pallo lisätään kentälle Add-metodin avulla.

```
        i++;
```

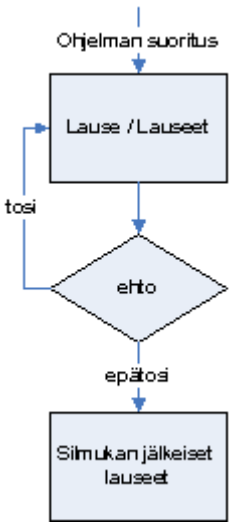
Silmukan jokaisen “kierroksen” jälkeen indeksin arvoa on lisättävä yhdellä, ettemme joutuisi ikuiseen silmukkaan.

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 aina *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.

Animaatio: Suorita ohjelma

Askella do-while rakenne vihreällä nuolella Tutki do-while-rakennetta

> Tulosta luvut 0-50 käyttäen do-while -silmukkaa.10pt

Katso luentovideo do-while -silmukasta Luento 9 (8m0s)

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.

```
1  using System;
2
3  /// <summary>
4  /// Harjoitellaan do-while-silmukan käyttöä.
5  /// </summary>
6  public class NimenTulostus
7  {
8      /// <summary>
9      /// Pyydetään käyttäjältä syöte ja tulostellaan.
10     /// </summary>
11     public static void Main()
12     {
13         String nimi;
14         do
15         {
16             Console.Write("Anna nimi > ");
17             nimi = Console.ReadLine();
18         } while (nimi != null && nimi.Length == 0);
19         Console.WriteLine();
20         Console.WriteLine("Hei, " + nimi + "!");
21     }
22 }
```

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

Kuinka ylempi monta palloa -esimerkki tehtäisiin for-silmukalla ◘ Luento 8 (3m13s)

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:

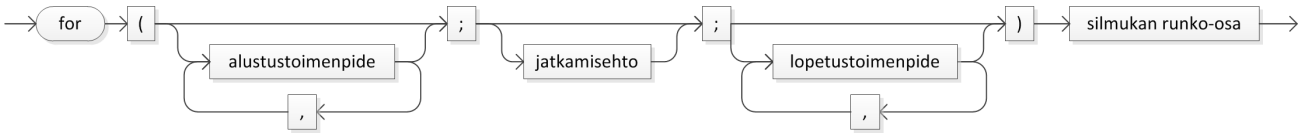
```
for (muuttujien alustukset; ehto; kasvatuslausekkeet)
{
    lauseet; // silmukan runko-osa
}
```

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.
- Kasvatuslausekkeet: silmukan lopussa tehtävät toimenpiteet: Useimmiten muuttujan tai muuttujien arvoa kasvatetaan yhdellä, mutta myös suuremmalla määrällä kasvattaminen/-vähentäminen on mahdollista. Voi olla pilkulla eroteltuina useita lausekkeita, esimerkiksi tyyliin `i++, j--`

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



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

> Älä **muuta** for-silmukan indeksiä muualla kuin `for`-rivillä.

Animaatio: Suorita ohjelma

Askella for-silmukka vihreällä nuolella Tutki for-silmukkaa

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

```
1         for (int i = 0; i < 10; i++)
2         {
3             Console.WriteLine("Hello World " + i);
4         }
```

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

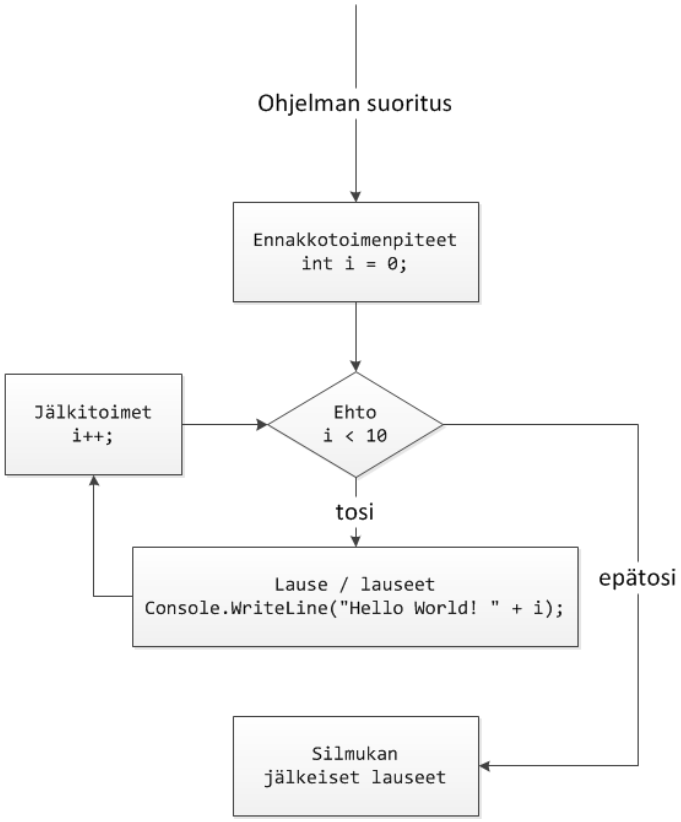
Vuokaaviona yllä olevan `for`-silmukan voisi kuvata alla olevalla tavalla.

Vuokaavio alla olevalle koodille...

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

... joka tulostaa seuraavaa:

```
Hello World! 0
Hello World! 1
Hello World! 2
Hello World! 3
Hello World! 4
Hello World! 5
Hello World! 6
Hello World! 7
Hello World! 8
Hello World! 9
```



Kuva 26: Vuokaavio for-silmukalle.

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

Silmukan runko-osassa ei useimmiten tulosteta mitään. Otetaan esimerkki, jossa luodun taulukon alkioihin sijoitetaan aina kahden edellisen alkion sisältämien lukujen summa (*Fibonaccin luku*). Ensimmäisiksi alkioiden arvoiksi asetetaan "manuaalisesti" luku 1.

```
1         int[] luvut = new int[10];
2         luvut[0] = 1;
3         luvut[1] = 1;
4
5         for (int i = 2; i < luvut.Length; i++)
6         {
7             luvut[i] = luvut[i - 1] + luvut[i - 2];
8         }
```

Huomaa, että silmukka on aloitettu indeksistä 2, jotta `i-2 >= 0` ja näin jokainen indeksi on laillinen lauseessa

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

Silmukan jälkeen taulukon sisältö näyttää seuraavalta.

```
```

Myös for-silmukalla voidaan tehdä "ikuinen" silmukka:

```
for (;;)
{
```

```
        // "ikuisesti" suoritettavat asiat
    }
```

Tämä tulostaisi ikuisesti i:n arvoa

```
    for (int i = 0; true; i++)
    {
        System.Console.WriteLine(i);
    }
```

Animaatio: Suorita ohjelma

Askella for-silmukka vihreällä nuolella Tutki for-silmukkaa

Muokkaa ohjelma toimivaksi. Laita pääohjelma ennen muita aliohjelmia. 10pt

```
using System;
public class KirjaimiaIsoksi
{
   public static void Main()
   {
      string[] nimet = {"keijo", "kaisa", "teppo", "jukka", "aku"};
      System.Console.WriteLine(String.Join(" ", EkaKirjainIsoksi(nimet)));
   }
   public static string[] EkaKirjainIsoksi(string[] merkkijonot)
   {
       for (int i = 0; i < merkkijonot.Length; i++)
       {
           if ( String.IsNullOrEmpty(merkkijonot[i]) ) continue;
           char a = merkkijonot[i][0];
           a = Char.ToUpper(a);
           merkkijonot[i] = "" + a + merkkijonot[i].Remove(0, 1);
       }
       return merkkijonot;
   }
}
```

Aikaisemmin `while`-lauseen kodalla tehtiin silmukka joka laskee taulukoiden alkioiden summan. Tämä silmukka on oikeastaan luonnollisempi tehdä `for`-silmukalla.

Laske summa

Laske taulukon kaikkien alkioiden summa. Käytä for-silmukkaa.10pt

```
//
      double summa = 0;
      double[] luvut = {1.7,2.456,36,-4.8,-5.67,60,-17,8.0,9,10.2};
```

Laske itseisarvojen summa

Muuta edellinen ohjelma toimimaan siten, että se laskee itseisarvojen summan. Käytä silmukkaa. 10pt

```
//
      double summa = 0;
      double[] luvut = {1.7,2.456,36,-4.8,5.67,60,-17,8.0,9,10.2};
```

Tehtävä käyttäjän syötteestä Itseisarvo

Muuta tehtävää Laske Itseisarvo niin, että luvut kysytään käyttäjältä. Käytä lukujen kysymiseen silmukkaa niin, että käyttäjä voi syöttää kuinka monta lukua vain. Jos käyttäjä syöttää jotain muuta kuin luvun, sitä ei oteta mukaan tulokseen. Summan laskeminen tapahtuu omassa aliohjelmassa, johon kirjoitetaan testit. Syötteet tallennetaan taulukkoon, joka viedään parametrina aliohjelmalle. Lisää myös dokumentaatio. Ohjelma voi tulostaa käyttäjälle jonkun tulostuksen. Tässä tehtävässä voi olla hyötyä kappaleista 17 ja 18. 10pt

```
using System;

///@author
///@version
///
/// <summary>
///
/// </summary>
public class ItseisarvoSyotteesta
{
    /// <summary>
    ///
    /// </summary>
    public static void Main()
    {

    }
}
```

Animaatio: Summaa positiiviset

Askella vihreällä nuolella. Avaa animaatio tästä.

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; kasvatuslauseet)
{
    lauseet; // silmukan runko-osa
}
```

voisi tehdä `while`-rakenteella seuraavasti.

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

Eli `while`-silmukka on helppo muuttaa for-silmukaksi kun kirjoittaa alustuslauseen eteen

```
for (
```

Sitten pyyhkii pois rivinvaihdon ja `while(`. Sitten laittaa ehdon jälkeen puolipisteen `;` ja siirtää

kasvatuslauseet sen perään ja poistaa niiden jälkeisen puolipisteen. Näin lauseet tulevat `for`-silmukkaan oikeaan järjestykseen.

Mihin `for`-silmukkaa sitten tarvitaan?

Oikeastaan ero on vain looginen viesti koodin lukijalle. Tyypillisesti `for`-silmukkaa käytetään kun tiedetään etukäteen kierrosten lukumäärä. Perusesimerkkinä kun pitää käydä jokaisessa taulukon alkiossa. `while` taas viestii lukijalle, että nyt tehdään monta kertaa, mutta ei vielä sitä kuinka monta.

`for`-silmukalla taulukon tulostava aliohjelma olisi seuraavanlainen:

```
1     public static void TulostaTaulukko(int[] taulukko)
2     {
3         for (int i=0; i<taulukko.Length; i++)
4         {
5             int luku = taulukko[i];
6             Console.Write(luku + " ");
7         }
8         Console.WriteLine();
9     }
```

Nuo kolme osaa - muuttujien alustus, ehto, lopussa tehtävät toimenpiteet - ovat osia, jotka jokaisessa silmukkarakenteessa tarvitaan. `for`-silmukassa ne kirjoitetaan selvästi peräkkäin 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).

Silmukoiden suurin syntaktinen ero on, että `continue`-lauseen osalta `for`- ja `while`-silmukat toimivat eri tavalla. `while`-silmukassa lisäykset voivat jäädä tekemättä `continue`-lauseen kanssa, koska `continue` “hyppää” silmukan loppuun ohittaen kasvatuslauseet. Toisaalta `for`-silmukassa kasvatuslausekkeet suoritetaan nimenomaan silmukan lopuksi ja ne tehdään myös `continue`-lauseen tapauksessa.

Toinen pieni ero on, että `for`-silmukassa kaikkien alustettavien muuttujien tulee olla samaa tyyppiä. Toki muuttujia voi esitellä ennen kuin silmukkaa kuten `while`-silmukassa, jos tarvitaan eri tietotyyppejä.

Edellinen for-silmukka while-silmukkana ja ohje miten sen voit muuttaa for-silmukaksi10pt

```
1     public static void TulostaTaulukko(int[] taulukko)
2     {
3         int i = 0;                // Kirjoita tämän eteen for (
4         while (i<taulukko.Length) // poista tästä while (. lisää ; perään
5         {
6             int luku = taulukko[i];
7             Console.Write(luku + " ");
8             i++;                  // siirrä tämä for ehdon ; jälkeen ja poista ;
9         }
10        Console.WriteLine();
11    }
```

16.4.2 Esimerkki: lumiukon pallot keltaisiksi

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;
```

```
1         // Lisätään pallot taulukkoon, ja sitten lisätään kentälle
2         PhysicsObject[] pallot = new PhysicsObject[3];
3         pallot[0] = LuoPallo(0, Level.Bottom + 200, Color.White, 100);
4         pallot[1] = LuoPallo(0, pallot[0].Y + 100 + 50, Color.White, 50);
5         pallot[2] = LuoPallo(0, pallot[1].Y + 50 + 30, Color.White, 30);
6         Add(pallot[0]); Add(pallot[1]); Add(pallot[2]);
7
8
9         // Muutetaan pallojen väri
10        for (int i = 0; i < pallot.Length; i++)
11        {
12            pallot[i].Color = Color.Yellow;
13        }
```

16.4.3 Harjoitus

Tee aliohjelma MuutaPallojenVari(pallot,vari), joka muuttaa PhysicsObject-taulukossa olevien olioiden värin halutuksi.10pt

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 niille omat aliohjelmat. Sen sijaan jos annamme luvut taulukossa, pärjäämme yhdellä aliohjelmalla. Tehdään siis nyt funktio `Keskiarvo`, joka palauttaa taulukossa olevien kokonaislukujen keskiarvon. Kirjoitetaan sille myös ComTest-testit.

```
1     /// <summary>
2     /// Palauttaa parametrina saamansa int-taulukon
3     /// alkoiden keskiarvon.
4     /// </summary>
5     /// <param name="luvut">summattavat luvut</param>
6     /// <returns>summa</returns>
7     /// <example>
8     /// <pre name="test">
9     ///   int[] luvut1 = {0};
10    ///   Keskiarvo(luvut1) ~~~ 0;
11    ///   int[] luvut2 = {3, 3, 3};
12    ///   Keskiarvo(luvut2) ~~~ 3;
13    ///   int[] luvut3 = {3, -3, 3};
14    ///   Keskiarvo(luvut3) ~~~ 1;
15    ///   int[] luvut4 = {-3, -6};
16    ///   Keskiarvo(luvut4) ~~~ -4.5;
17    /// </pre>
18    /// </example>
19    public static double Keskiarvo(int[] luvut)
20    {
21       double summa = 0;
22       for (int i = 0; i < luvut.Length; i++)
```

```
23          {
24              summa += luvut[i];
25          }
26          return summa / luvut.Length;
27      }
```

Ohjelmassa lasketaan ensiksi kaikkien taulukon 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 *pienempi tai yhtä suuri kuin* -operaattori (`<=`), 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 "`<=`"-operaattori, 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.

Edellisen esimerkin testejä erilaisille taulukoille voitaisiin nyky C#:issa kirjoittaa lyhyemmin muodossa (kokeile useammalla alkiolla):

```
///   Keskiarvo([-3, -6]) ~~~ -4.5;
```

16.4.5 Harjoitus

Pohdi, mikä **erittäin tärkeä** tapaus jää huomiotta taulukon keskiarvon laskemisessa edellisessä esimerkissä.

Korjaa edellistä esimerkkiä niin, että tapaus huomioidaan ja lisää sitä varten oma testi.

Mallivastaus

Ongelmana on nollalla jakaminen siinä tilanteessa kun taulukko on tyhjä! `for`-silmukka pitää huolen siitä, että jos taulukon pituus on heti lähtötilanteessa 0, niin silmukkaa ei tehdä yhtään kertaa. Mutta sitten `return`-lauseessa voi tapahtua nollalla jakaminen. Tyhjän taulukon testi voitaisiin lisätä ennenkin `for-silmukkaa`, mutta jos kyseessä olisi tilanne, jossa kaikkia alkioita ei otettaisi huomioon ja lukumäärä laskettaisiin silmukan aikana, ei lukumäärä ole tiedosssa ennen silmukan suorittamista. Siksi nollalla jaon estäminen on turvallisinta suorittaa juuri ennen jakamista, eli lisätään ennen `return`-lausetta:

```
if ( luvut.Length == 0 ) return 0;
```

16.4.6 Esimerkki: Taulukon kääntäminen käänteiseen järjestykseen

Kontrollirakenteen ensimmäisessä osassa voidaan 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>
/// <example>
/// <pre name="test">
///   int[] luvut = { 12, 3, 5, 9, 7, 1, 4, 9 };
///   KaannaTaulukko(luvut);
///   String.Join(" ",luvut) === "9 4 1 7 9 5 3 12";
///   int[] luvut2 = { 12, 3 };
///   KaannaTaulukko(luvut2);
///   String.Join(" ",luvut2) === "3 12";
///   int[] luvut1 = { 5 };
///   KaannaTaulukko(luvut1);
///   String.Join(" ",luvut1) === "5";
///   int[] luvut0 = { };
///   KaannaTaulukko(luvut0);
///   String.Join(" ",luvut0) === "";
/// </pre>
/// </example>
public static void KaannaTaulukko(int[] taulukko)
{
   int loppu = taulukko.Length-1;
   for (int vasen = 0, oikea = loppu; vasen < oikea; vasen++, oikea--)
   {
      int temp = taulukko[vasen];
      taulukko[vasen] = taulukko[oikea];
      taulukko[oikea] = temp;
   }
}
```

Huomaa että uudemmissa C#-toteutuksissa edellä voidaan taulukkoa testata suoraan taulukkona:

```
/// luvut === (int[])[9, 4, 1, 7, 9, 5, 3, 12];
```

Ainoa pieni harmi on, että että tuohon hakasulkujen eteen on kirjoitettava tuo tyypin muunnos `(int[])`, koska muuten kääntäjä ei tiedä tuon oikean puolen tyyppiä.

Toinen tapa lyhentää tuota `Join`-käyttöä olisi käyttää `=J=`-vertailua, joka vertailun vasemmalla puolella olevan taulukon `Join`-funktiolla merkkijonoksi:

```
///   luvut =J= "9, 4, 1, 7, 9, 5, 3, 12";
```

Kokeile noita edelliseen esimerkkiin ja valitse tilanteen mukaan sopivin.

Lue lisää taulukoiden testaamisesta ComTestin dokumentista.

Ideana yllä olevassa aliohjelmassa on, että meillä on kaksi muuttujaa. Muuttujia voisi kuvata kuvaannollisesti osoittimiksi. Osoittimista toinen osoittaa aluksi taulukon alkuun ja toinen taulukon loppuun. Oikeasti osoittimet ovat int-tyyppisiä muuttujia, jotka saavat arvoikseen 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 al-

kioiden paikat keskenään, joihin osoittimet osoittavat. Silmukan suoritus lopetetaan juuri ennen kuin osoittimet kohtaavat toisensa.

Tarkastellaan aliohjelmaa nyt hieman tarkemmin.

```
int loppu = taulukko.Length-1;
```

Ensimmäiseksi on alustettu taulukon viimeisen paikan indeksi muuttujaan `loppu`.

```
for (int vasen = 0, oikea = loppu; 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 pienempi 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 sijaan vähennetään.

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

Silmukan runko-osassa ensimmäisenä sijoitetaan 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: 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. Eli funktio palauttaa uuden taulukon eikä muuta alkuperäistä.10pt

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.

```
1 using System;
2 using System.Collections.Generic;
3
4 /// @author  Antti-Jussi Lakanen, Martti Hyvönen
5 /// @version 22.12.2011
```

```
6  ///
7  /// <summary>
8  /// Harjoitellaan vielä taulukoiden käyttöä.
9  /// </summary>
10 public class Arvosana
11 {
12     /// <summary>
13     /// Laskee opiskelijan tenttiarvosanan asteikoilla 0-5.
14     /// </summary>
15     /// <param name="maksimipisteet">Tentin pisteet jolla saa 5.</param>
16     /// <param name="lapaisyraja">Tentin läpipääsyraja.</param>
17     /// <param name="tenttipisteet">Opiskelijan tenttipisteet.</param>
18     /// <returns>Opiskelijan tenttiarvosana.</returns>
19     /// <example>
20     /// <pre name="test">
21     /// LaskeArvosana(24, 12, 11) === 0;
22     /// LaskeArvosana(24, 12, 12) === 1;
23     /// LaskeArvosana(24, 12, 13) === 1;
24     /// LaskeArvosana(24, 12, 14) === 1;
25     /// LaskeArvosana(24, 12, 15) === 2;
26     /// LaskeArvosana(24, 12, 19) === 3;
27     /// LaskeArvosana(24, 12, 20) === 3;
28     /// LaskeArvosana(24, 12, 22) === 4;
29     /// LaskeArvosana(24, 12, 24) === 5;
30     /// LaskeArvosana(24, 12, 28) === 5;
31     /// </pre>
32     /// </example>
33     public static int LaskeArvosana(int maksimipisteet, int lapaisyraja,
34                                      int tenttipisteet)
35     {
36         double[] arvosanaRajat = new double[6];
37         double arvosanojenPisteErot = (maksimipisteet - lapaisyraja) / (5.0-1-0);
38
39
40         //Arvosanan 1 rajaksi tentin läpipääsyraja
41         arvosanaRajat[1] = lapaisyraja;
42
43         //Asetetaan taulukkoon jokaisen arvosanan raja
44         for (int i = 2; i <= 5; i++)
45         {
46             arvosanaRajat[i] = arvosanaRajat[i - 1] + arvosanojenPisteErot;
47         }
48
49         //Katsotaan mihin arvosanaan tenttipisteet riittävät
50         for (int i = 5; 1 <= i; i--)
51         {
52             if (arvosanaRajat[i] <= tenttipisteet) return i;
53         }
54         return 0;
55     }
56
57     /// <summary>
58     /// Pääohjelma
59     /// </summary>
60     public static void Main()
61     {
62         Console.WriteLine(LaskeArvosana(24, 12, 19)); // tulostaa 3
63         Console.WriteLine(LaskeArvosana(24, 12, 11)); // tulostaa 0
64     }
```

```
65 }
```

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) / (5.0-1.0);
```

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 parametreikseen arvosanojen rajat taulukossa ja opiskelijan tenttipisteet. Molempien aliohjelmien ComTest-testit ovat myös näkyvillä.

```
1  using System;
2
3  /// @author  Antti-Jussi Lakanen, Martti Hyvönen
4  /// @version 22.12.2011
5  ///
6  /// <summary>
7  /// Harjoitellaan taulukoiden käyttöä ja lasketaan tenttiarvosanoja.
8  /// </summary>
9  public class Arvosanat
10 {
11     /// <summary>
12     /// Laskee tenttiarvosanojen pisterajat taulukkoon.
13     /// </summary>
14     /// <param name="maksimipisteet">tentin pisteet jolla saa 5</param>
15     /// <param name="lapaisyraja">tentin läpipääsyraja</param>
16     /// <returns>arvosanojen pisterajat taulukossa</returns>
17     /// <example>
18     /// <pre name="test">
19     /// double[] rajat1 = LaskeRajat(24, 12);
20     /// rajat1 =J= "0, 12, 15, 18, 21, 24";
21     /// LaskeRajat(27, 15) =J= "0, 15, 18, 21, 24, 27";
22     /// </pre>
23     /// </example>
24     public static double[] LaskeRajat(int maksimipisteet, int lapaisyraja)
25     {
26         double[] arvosanaRajat = new double[6];
27         double arvosanojenPisteErot =
28            Math.Round((maksimipisteet - lapaisyraja) / (5.0-1.0), 1);
29
30         arvosanaRajat[1] = lapaisyraja;
31
32         // Asetetaan taulukkoon jokaisen arvosanan raja
33         for (int i = 2; i <= 5; i++)
34             arvosanaRajat[i] = arvosanaRajat[i - 1] + arvosanojenPisteErot;
35         return arvosanaRajat;
36     }
37
38
39     /// <summary>
40     /// Laskee opiskelijan tenttiarvosanan asteikoilla 0-5.
41     /// </summary>
42     /// <param name="pisterajat">Arvosanojen rajat taulukossa.
43     /// Arvosanan 1 raja taulukon indeksissä 1 jne. </param>
44     /// <param name="tenttiPisteet">Saadut tenttipisteet.</param>
45     /// <returns>Tenttiarvosana välillä [0..5].</returns>
46     /// <example>
47     /// <pre name="test">
48     /// double[] rajat = {0, 12, 14, 16, 18, 20};
49     /// LaskeArvosana(rajat, 11) === 0;
50     /// LaskeArvosana(rajat, 12) === 1;
51     /// LaskeArvosana(rajat, 14) === 2;
52     /// LaskeArvosana(rajat, 22) === 5;
53     /// LaskeArvosana(rajat, 28) === 5;
54     /// </pre>
55     /// </example>
56     public static int LaskeArvosana(double[] pisterajat, int tenttiPisteet)
57     {
58         for (int i = pisterajat.Length - 1; 1 <= i; i--)
59         {
```

```
60              if (pisterajat[i] <= tenttiPisteet) return i;
61          }
62          return 0;
63      }
64
65      /// <summary>
66      /// Pääohjelmassa pari esimerkkiä.
67      /// </summary>
68      public static void Main()
69      {
70          double[] pisterajat = LaskeRajat(24, 12);
71          Console.WriteLine(LaskeArvosana(pisterajat, 12)); // tulostaa 1
72          Console.WriteLine(LaskeArvosana(pisterajat, 20)); // tulostaa 3
73          Console.WriteLine(LaskeArvosana(pisterajat, 11)); // tulostaa 0
74      }
75  }
```

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.

16.4.9 Harjoitus 16.4

Osaisitko tehdä ohjelman, jossa taulukon arvot muutetaan merkkijonoksi? Katso luennolta mallia ja täydennä ohjelma tehtävään 16.4.

Kuinka taulukko muutetaan merkkijonoksi käyttäen funktiota ◘ Luento (23m21s)

Tehtävä 16.4

Täydennä ohjelma luennon mukaisesti. Muista dokumentaatio ja testit.10pt

16.5 foreach-silmukka

Taulukoita ja monia muita tietorakenteita käsiteltäessä voidaan käyttää myös foreach-silmukkaa. Nimensä mukaisesti se käy läpi kaikki taulukon alkiot. Se on syntaksiltaan selkeämpi silloin, kun haluamme tehdä jotain jokaiselle taulukon alkiolle jättämättä yhtään alkiota väliin. Sen syntaksi on yleisessä muodossa seuraava.

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

Tämä vastaa for-silmukkaa:

```
for (int i=0; i<taulukko.Length; i++)
{
```

```
        taulukonAlkionTyyppi alkio = taulukko[i];
        lauseet;
    }
```

Nyt foreach-silmukan kontrollilausekkeessa ilmoitetaan vain kaksi asiaa. Ensiksi annetaan tyyppi ja nimi muuttujalle, johon tulee yksi kerrallaan kukin taulukon alkio. Tyypin täytyy olla sama kuin käsiteltävän taulukon alkiotyyppi, mutta nimen saa itse keksiä. Tälle muuttujalle tehdään ne toimenpiteet, jotka jokaiselle taulukon alkiolle halutaan tehdä. Toisena tietona foreach-silmukalle pitää antaa sen taulukon nimi, jota halutaan käsitellä. Esimerkiksi `TulostaTaulukko` voitaisiin nyt tehdä seuraavasti.

```
1     public static void TulostaTaulukko(int[] taulukko)
2     {
3         foreach (int luku in taulukko)
4         {
5             Console.Write(luku + " ");
6         }
7         Console.WriteLine();
8     }
```

Vapaasti suomennettuna: “Jokaiselle luvulle taulukossa (tee)…”.

Huomaa että edellinen `foreach` on täsmälleen sama kuin seuraava koodi. Siitä on ehkä helpompi ymmärtää miksi muuttujan `luku` muuttaminen edellisessä `foreach`-silmukassa ei muuttaisi itse taulukon vastaavassa paikassa olevaa arvoa.

```
1     public static void TulostaTaulukko(int[] taulukko)
2     {
3         for (int i = 0; i < taulukko.Length; i++)
4         {
5             int luku = taulukko[i];
6             Console.Write(luku + " ");
7             // jos olisi vaikka luku = 0, ei taulukko[i] muuttuisi
8         }
9         Console.WriteLine();
10    }
```

Tehtävä 16.4.5

Täydennä aliohjelma, joka palauttaa arvon true jos taulukosta löytyy etsittävä merkki.10pt

```
1     /// <summary>
2     /// Etsii moniulotteisesta taulukosta tiettyä merkkiä
3     /// </summary>
4     /// <param name="taulukko">taulukko josta etsitään</param>
5     /// <param name="etsittava">etsittävä merkki</param>
6     /// <returns>true jos löytyi</returns>
7     public static bool etsiTaulukosta(char[,] taulukko, char etsittava)
8     {
9         return false;
10    }
```

Mallivastaus

```
public static bool etsiTaulukosta(char[,] taulukko, string etsittava)
{
```

```
        if ( etsittava.Length < 1 ) return false;
        char etsittavaKirjain = etsittava[0];
        foreach (char kirjain in taulukko)
            if ( kirjain == etsittavaKirjain ) return true;
        return false;
    }
```

Huom! `foreach`-silmukalla **ei** voi muuttaa taulukon alkioiden arvoja! Toki jos taulukossa on olioviitteitä, voidaan olioiden sisältöjä muuttaa kuten myöhemmin oleva `taulukon pallot keltaisiksi` esimerkki osoittaa.

Animaatio: Suorita foreach-silmukka

Askella for-silmukka vihreällä nuolella. Tässä Java-esimerkissä syntaksi on hieman erilainen. Tutki for-silmukkaa

Tehtävä 16.4.6

Kirjoita edellistä animaatiota vastaava ohjelma C#:illa10pt

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.

```
1         foreach (PhysicsObject pallo in pallot)
2         {
3             pallo.Color = Color.Yellow;
4         }
```

Huomaa, että `foreach`-silmukalla ei voi muuttaa alkioiden arvoja. Edellisessä esimerkissä kuitenkin alkiona on `pallo`, joka on viitemuuttuja. Sitä ei voi muuttaa, mutta sen viitteen avulla voi muuttaa itse olioita, jonne `pallo` viittaa.

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.

```
1         for (int r = 0; r < 3; r++) // rivit
2         {
3            for (int s = 0; s < 3; s++) // sarakkeet
4            {
5               Console.Write(elokuvat[r, s] + " | ");
6            }
7            Console.WriteLine();
8         }
```

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 ulommassa silmukassa indeksejä käydään läpi eri muuttujalla kuin sisemmässä silmukassa. 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`. Indeksien niminä voisi käyttää myös `iy` ja `ix`.

Animaatio: Suorita ohjelma

Askella sisäkkäiset for-silmukat vihreällä nuolella Tutki sisäkkäisiä for-silmukoita

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).

```
1     /// <summary>
2     /// Aliohjelma palauttaa sen rivin (indeksin),
3     /// jolla on eniten vapaata (eli rivi, jolla
4     /// eniten 0-alkioita). Jos näitä rivejä on useita, niin
5     /// palautetaan niistä ensimmäinen.  Jos millään rivillä
6     /// ei ole vapaata, palautetaan -1.
7     /// </summary>
8     /// <param name="ruudut">taulukko ruuduista, kussakin
9     /// ruudussa 0 = vapaa, muu numero = laivan numero.</param>
10    /// <returns>Rivi, jolla eniten vapaata tai -1.</returns>
11    /// <example>
12    /// <pre name="test">
13    /// int[,] ruudut1 = { {1, 0, 2}, {0, 0, 2}, {3, 3, 3} };
14    /// RiviJollaEnitenVapaata(ruudut1) === 1;
15    /// int[,] ruudut2 = { {1, 1, 1}, {2, 2, 2}, {3, 3, 3} };
16    /// RiviJollaEnitenVapaata(ruudut2) === -1;
17    /// int[,] ruudut3 = { {1, 1, 0}, {0, 0, 2}, {0, 3, 0} };
18    /// RiviJollaEnitenVapaata(ruudut3) === 1;
19    /// RiviJollaEnitenVapaata(ruudut3) === 1;
20    /// </pre>
21    /// </example>
22    public static int RiviJollaEnitenVapaata(int[,] ruudut)
23    {
24      int riviJollaEnitenVapaata = -1;
25      int enitenVapaitaMaara = 0;
26
27      for (int r = 0; r < ruudut.GetLength(0); r++)
28      {
29        int vapaata = 0;
30        for (int s = 0; s < ruudut.GetLength(1); s++)
31        {
32          if (ruudut[r, s] == 0) vapaata++;
```

```
33          }
34          if (vapaata > enitenVapaitaMaara)
35          {
36            riviJollaEnitenVapaata = r;
37            enitenVapaitaMaara = vapaata;
38          }
39        }
40        return riviJollaEnitenVapaata;
41      }
```

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. Joskus, kuitenkin voi olla selkeämpää katkaista silmukka tai hypätä uudelle kierrokselle, kun lisätä suuri joukko sisäkkäisiä `if`-lauseita.

16.8.1 break

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

```
1         int laskuri = 0;
2         while (true)
3         {
4            if (laskuri >= 10) break;
5            Console.WriteLine("Hello world!");
6            laskuri++;
7         }
```

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 samalla tavalla. Useimmiten break-lauseen käytön voikin välttää.

```
1         int laskuri = 0;
2         while (laskuri < 10)
3         {
4            Console.WriteLine("Hello world!");
5            laskuri++;
6         }
```

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

Animaatio: Suorita for-silmukka joka katkaistaan break-lauseella

Askella for-silmukka vihreällä nuolella. Tutki for-silmukkaa

16.8.2 continue

`continue`-lause hyppää yli silmukan sen hetkisen kierroksen suorittamisen. Seuraavaksi suoritetaan silmukan jatkamisehto, jolloin silmukan runko-osa toteutetaan jälleen ehdon ollessa tosi, tai mikäli ehto palauttaa `false`, silmukan suorittaminen päättyy. Arkisesti voisi sanoa, että `continue`-lauseella voi hypätä silmukan kierroksen runko-osan lopun yli.

```
1         // Tulostetaan luku vain jos se on pariton
2         for (int i = 0; i < 20; i++)
3         {
4             if (i % 2 == 0) continue;
5             Console.WriteLine(i);
6         }
```

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.

```
1         // Tulostetaan luku vain jos se on pariton
2         for (int i = 0; i < 20; i++)
3         {
4             if (i % 2 != 0)
5                Console.WriteLine(i);
6         }
```

Tai vielä yksinkertaisemmin seuraavasti.

```
1         // Tulostetaan luku vain jos se on pariton
2         for (int i = 1; i < 20; i += 2)
3         {
4             Console.WriteLine(i);
5         }
```

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

Oikeastaan `continue` on ainoa mikä toimii eri tavalla `for` ja `while`-silmukoissa. `for`-silmukassa `continue` "hyppää" ensimmäiseen kasvatuslausekkeeseen ja `while`-silmukassa ehtolauseeseen.

Lauseiden `break` ja `continue` voit kuvitella olevan hyppylauseita silmukan lopettavan aaltosulun luokse. `continue` sen vasemmalle puolelle ja `break` sen oikealle puolelle. Tällöin on helpompi ymmärtää miksi `while`-lauseessa ei suoriteta kasvatuslauseita `continue` tapauksessa, mutta `for`-lauseessa ne suoritetaan.

Plugin imagex error:

Plugin does not support printing yet. Please refer to TIM help pages if you want to learn how you can manually define what to print here.

16.8.3 return

Usein erityisesti aliohjelmissa `break`-tilalla voidaan käyttää `return`-lausetta. Oletetaan että meidän pitäisi laskea taulukossa olevien lukujen summaa kunnes taulukko loppuu tai tulee vastaan sovittu luku.

Esimerkki `break-lauseella`10pt

```
    /// <summary>
    /// Funktio palauttaa niiden lukujen summa, jotka ovat ennen lopetus-arvoa
    /// taulukossa.
    /// </summary>
    /// <param name="t">taulukko josta summa lasketaan</param>
    /// <param name="lopetus">lopetetaanjos luku >= tämä</param>
    /// <returns>Ennen lopetus-arvoa olevien lukujen summa</returns>
    /// <example>
    /// <pre name="test">
    /// Summa(new int[]{1,2,3,4}, 99) === 10;
    /// Summa(new int[]{1,2,3,4}, 3) === 3;
    /// </pre>
    /// </example>
    public static int Summa(int[] t, int lopetus)
    {
        int summa = 0;
        foreach (int luku in t)
        {
            if ( luku >= lopetus ) break;
            summa += luku;
        }
        return summa;
    }
```

Esimerkki `return-lauseella`10pt

```
    public static int Summa(int[] t, int lopetus)
    {
        int summa = 0;
        foreach (int luku in t)
        {
            if ( luku >= lopetus ) return summa;
            summa += luku;
        }
        return summa;
    }
```

Usein tämä johtaa kuitenkin siihen, että silmukan sisällä olevan `return` lauseen takia joudutaan toistamaan samoja laskuja, mitä tehdään funktion lopussa. Edellä tuo ei ollut kovin hankalaa, mutta esimerkiksi keskiarvoa laskettaessa nollalla jakamisen välttämiseksi tulisi jo enemmän koodia.

Tämän takia moni suositteleekin että aliohjelmissa olisi vain yksi poistumiskohta.

16.9 Poistuminen ennen silmukkaa

Edellä todettiin, että usein olisi hyvä jos aliohjelmasta poistumiskohtia ei olisi enempää kuin yksi. Tästä säännöstä voi selkeästi poiketa, mikäli aliohjelman alussa tutkitaan, että onko aliohjelman suorittaminen järkevää. Esimerkiksi jos edellisen esimerkin funktiota muutettaisiin niin, että

summan laskemista ei pidettäisi mielekkääna jos alkoita (esim. havaintoja) on vähemmän kuin 3.

Ennen silmukkaa poistuminen10pt

```
    /// <summary>
    /// Funktio palauttaa niiden lukujen summa, jotka ovat ennen lopetus-arvoa
    /// taulukossa. Jos alle minLkm lukua, palautetaan aina 0.
    /// </summary>
    /// <param name="t">taulukko josta summa lasketaan</param>
    /// <param name="lopetus">lopetetaanjos luku >= tämä</param>
    /// <param name="minLkm">pitää olla vähintään näin monta alkiota</param>
    /// <returns>Ennen lopetus arvoa olevien lukujen summa tai 0 jos lukuja vähän↩
    </returns>
    /// <example>
    /// <pre name="test">
    /// Summa(new int[]{1,2,3,4}, 99, 3) === 10;
    /// Summa(new int[]{1,2}, 99, 3) === 0;
    /// </pre>
    /// </example>
    public static int Summa(int[] t, int lopetus, int minLkm)
    {
        if ( t.Length < minLkm ) return 0;
        int summa = 0;

        foreach (int luku in t)
        {
            if ( luku >= lopetus ) break;
            summa += luku;
        }
        return summa;
    }
```

Toki edellä oleva voitaisiin tehdä myös käyttämällä päinvastaista ehtoa ja sitten sulkea suoritus lohkoon. Tämä tapa kuitenkin lisää ohjelmassa tarvittavien sisennystasojen määrää ja siksi sitä voidaan pitää tässä tapauksessa huonompana ratkaisuna.

Sama lohkolla10pt

```
    public static int Summa(int[] t, int lopetus, int minLkm)
    {
        int summa = 0;
        if ( t.Length >= minLkm )
        {
            foreach (int luku in t)
            {
                if ( luku >= lopetus ) break;
                summa += luku;
            }
        }
        return summa;
    }
```

16.10 Älä tee silmukassa testejä, jotka voi tehdä sen ulkopuolella.

Edellisessä esimerkissä oli testi:

```
if ( t.Length < minLkm ) return 0;
```

silmukan ulkopuolella ja siellä sen pitääkin olla. Jos testi olisi silmukan sisällä, tehtäisiin se jokaisella silmukan kierroksella turhaan, koska jos ehto on kerran totta, on se jokaisella kierroksella. Ja jos ehto on kerran epätosi, on se sitä jokaisella kierroksella. Ja silloin ehdon testaaminen jokaisella kierroksella vaan turhaan hidastaa silmukkaa.

Eli jos ehtolauseessa ei ole yhtään silmukan suorituksen aikana muuttuvaa tekijää, pitää ehtolause olla silmukan ulkopuolella. Edellähän taulukon pituus on aliohjelman aikana vakio, samoin parametrina tullut `minLkm`, joten kumpikaan ei niistä muutu silmukan aikana.

16.11 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
}
```

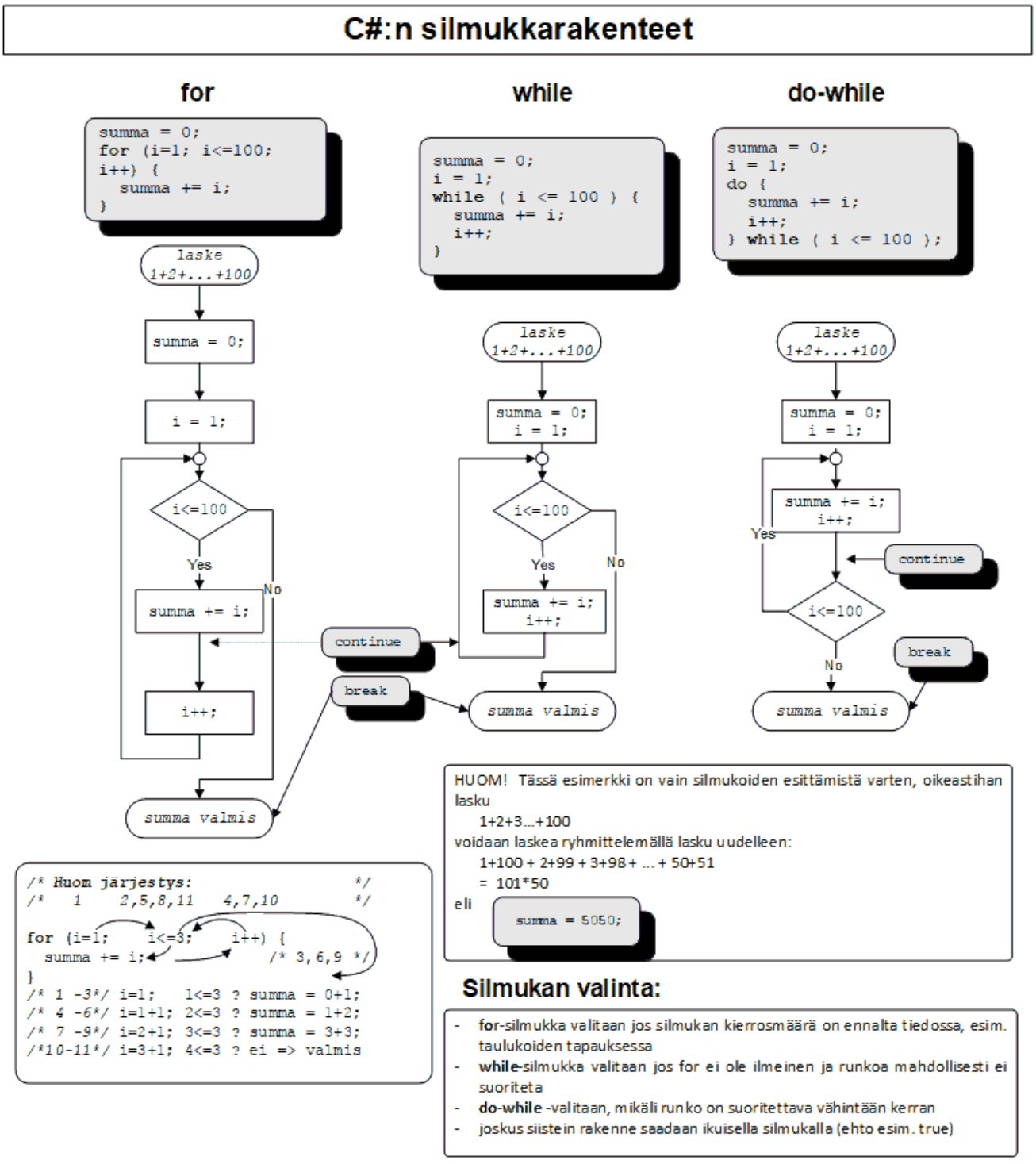
Jos silmukan ehdoksi asetetaan true, täytyy jossain kohtaa ohjelmassa olla break-lause, ettei silmukasta tulisi ikuista. Tällainen rakenne on näppärä juuri silloin, kun haluamme tarkastella silmukan lopettamista keskellä silmukkaa.

16.12 Yhteenveto

Silmukan valinta:

- `for`: Jos silmukan kierrosten määrä on ennalta tiedossa. Perustietotyyppien taulukoissa ja listoissa myös silloin jos alkioita pitää muuttaa.
- `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.
- ikuinen silmukka: Jos joutuu kirjoittamaan ehtoja useita kertoja tai väkisin alustamaan ehdon niin, että se on totta ensimmäisellä kierroksella.

Seuraava kuva kertaa C#-kielen valmiita silmukoita (`foreach` ei ole kuvassa):



Kuva 27: C#:n silmukat.

Tehtava 16.5

Tee funktio `NimetIsolla`, joka palauttaa taulukon merkkijonot

```
1 using System;
2
3 /// @author
4 /// @version
5 /// <summary>
6 ///
```

```
/// </summary>
public class Nimet
{
    /// <summary>
    /// Taulukon vienti parametrina
    /// </summary>
    public static void Main()
    {
        string[] nimet = { "Henna", "Matti", "Kaisa", "Keijo", "Matilda", "Seppo" };
        string[] nimet2 = NimetIsolla(nimet);
        System.Console.WriteLine(String.Join(" ", nimet2));
    }
}
```

Tehtävä 16.6

Valitse näytä koko koodi. Täydennä aliohjelma, jolle viedään mariisi, sekä kahden merkin mittainen merkkijono. Aliohjelma palauttaa arvon true jos taulukosta löytyy 2-merkin merkkijono. Taulukkoa tarvitsee käydä läpi vain vasemmalta oikealle. 10pt

```
    /// <summary>
    /// Etsitään onko taulukossa 2-merkin merkkijonoa
    /// </summary>
    /// <param name="taulukko">taulukko</param>
    /// <param name="etsittava">merkkijono, enintään kaksi merkkiä</param>
    /// <returns>true jos merkkijono löytyi</returns>
    public static bool EtsiTaulukosta(char[,] taulukko, string etsittava)
    {


        return false;
    }
```

Luku 17

Merkkijonojen pilkkominen ja muokkaaminen

17.1 String.Split()

Luentovideo Luento 15 (10m24s)

Merkkijonoja voidaan toki pilkkoa `IndexOf` ja `Substring`-metodien yhdistelmällä, mutta monissa tapauksissa tämä käy vielä kätevämmin `string`-olion `Split`-metodilla. Metodi palauttaa palaset merkkijono-tyyppisessä taulukossa `string[]`. `Split()`-metodille annetaan parametrina taulukko niistä merkeistä (`char`), joiden halutaan toimivan erotinmerkkeinä. Oletetaan syöte, ja lisäksi oletetaan, että haluamme välilyönnin, puolipisteen ja pilkun toimivan erotinmerkkeinä.

```
1         char[] erottimet = new char[] { ' ', ';', ',' };
2         string jono = "Kissa istuu puussa, naukuu";
3         string[] pilkottu = jono.Split(erottimet);
4         for (int i=0; i<pilkottu.Length; i++)
5         {
6             string pala = pilkottu[i];
7             Console.WriteLine("{0,2}: '{1}'", i, pala);
8         }
```

Koska `Split`-metodin esittely on muotoa:

```
public string[] Split(params char[] separator)
```

voidaan edellinen kutsu tehdä myös niin, että kutsuun luetellaan taulukon alkiot toisistaan pilkulla eroteltuina. Eli `params` tyyppiselle taulukolle voidaan viedä taulukko tai lueteltu lista taulukon alkioista. `params`-määreellä olevan parametrin pitää olla aina kutsun viimeinen parametri.

```
1         string jono = "Kissa istuu puussa, naukuu";
2         string[] pilkottu = jono.Split(' ', ';', ',' );
```

Vaikka dokumentaatiosta ei selvästi käykään ilmi, voidaan `Split`-metodia kutsua myös ilman parametreja, ja silloin pilkkominen tapahtuu välilyönnin kohdalta:

```
1         string jono = "Kissa istuu puussa ja  naukuu";
2         string[] pilkottu = jono.Split();
```

Katso animaatiota liikkumalla nuolilla10pt

17.1.1 Tyhjien alkioiden huomiotta jättäminen

Jos esimerkiksi käyttäjä antaa useamman erotinmerkin peräkkäin (vaikkapa kaksi välilyöntiä kuten edellä), niin joskus voi olla toivottavaa, ettei kuitenkaan taulukkoon luoda tyhjää alkiota. Edellisessä esimerkeissä tulee yksi tyhjä alkio. Tämä voidaan hoitaa antamalla Split()-metodille lisäparametri `StringSplitOptions.RemoveEmptyEntries`. Huomattakoon, että tämän muodon kutsussa ei ole `params`-määrettä, joten merkkijonotaulukko on luotava itse.

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

```
1         char[] erottimet = new char[] { ' ', ';', ',' };
2         string jono = "kissa,,,; koira";
3         string[] pilkottu = jono.Split(erottimet,
4                     StringSplitOptions.RemoveEmptyEntries);
```

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

Erotinmerkkien taulukon voi toki luoda "lennosta" parilla eri tavalla. Yksi vaihtoehto on muuttaa merkkijono kirjaintaulukoksi:

```
1         string jono = "kissa,,,;   koira";
2         string[] pilkottu = jono.Split(" ,;".ToCharArray(),
3                     StringSplitOptions.RemoveEmptyEntries);
```

Toinen tapa olisi luoda taulukko suoraan kutsussa:

```
1         string jono = "kissa,,,;   koira";
2         string[] pilkottu = jono.Split(new char[] { ' ', ';', ',' },
3                     StringSplitOptions.RemoveEmptyEntries);
```

`Split`-metodista on vielä joskus hyödyllinen muoto, jolla voidaan rajata palasten määrää:

```
1         char[] erottimet = new char[] { ' ', ';', ',' };
2         string jono = "Kissa istuu puussa, naukuu";
3         string[] pilkottu = jono.Split(erottimet,2);
```

On kuitenkin huomattava, ettei edellinenkään kutsu takaa, että saadaan kaksi palasta. Siksi saatujen palojen määrä on aina tarkistettava tulostaulukon pituudesta, jos siitä halutaan käsitellä tietty määrä paloja. Toisaalta joskus `if`-lauseiden välttämiseksi voi olla "nätimpää" pitää huoli, että saadaan varmasti riittävä määrä paloja:

```
1         string jono = "eka,toka";
2         string[] pilkottu = (jono+",,").Split(',');
3         Console.WriteLine(pilkottu[0]);
4         Console.WriteLine(pilkottu[1]);
```

```
5         Console.WriteLine(pilkottu[2]);
```

Edellä on haluttu, että palasia saadaan aina vähintään 3. Jonoon on ennen pilkkomista lisätty riitävä määrä erotinmerkkejä (tässä tapauksessa pilkkuja), ja pilkkominen on tehty vasta tästä syntyvälle uudelle merkkijonolle. Näin kolmas jono on varmasti olemassa tässäkin tapauksessa (nyt toki tyhjä). Kun lisättiin kaksi pilkkua, saadaan varmasti tyhjästäkin jonosta kolme osaa, ja jatkossa olevia indeksiviitteitä ei ole tarvinnut suojata if-lauseella. Tosin pitää tehokkuutta miettiessä muistaa tästä "tempusta" syntyvä uusi merkkijono ja tiukoissa silmukoissa miettiä, onko ylimääräinen ehto sittenkin nopeampi. Yksittäin käytettynä "tempusta" ei ole mitattavaa haittaa.

17.2 String.Trim()

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

```
1         String jono = "  kalle   ja kille    ";
2         Console.WriteLine("|" + jono .Trim() + "|" );
3         // "|kalle   ja kille|"
```

tulostaisi:

```
|kalle   ja kille|
```

Huomaa, että merkkijonon keskellä olevia "ylimääräisiä" välilyöntejä `Trim`-metodi ei kuitenkaan poista. Ylimääräiset keskellä olevat toistot voi poistaa esimerkiksi käyttäen säännöllisiä lausekkeita (regular expressions). Voidaan esimerkiksi sanoa, että vaihdetaan kaikki usean välilyönnin yhdistelmät yhdeksi välilyönniksi:

```
1         string jono = "  kalle   ja kille    ";
2         Regex rgx = new Regex(" +"); // vähintään yksi välilyönti
3         jono = rgx.Replace(jono, " ");
4         Console.WriteLine("|" + jono + "|" );
5         // | kalle ja kille |
```

Jos tästä vielä poistetaan alku- ja loppuvälilyönnit, niin silloin kaikki turhat välilyönnit ovat poistuneet:

```
1         string jono = "  kalle   ja      kille    ";
2         Regex rgx = new Regex(" +"); // vähintään yksi välilyönti
3         jono = rgx.Replace(jono, " ").Trim();
4         Console.WriteLine("|" + jono + "|" );
5         // |kalle ja kille|
```

Rexexpejä voit kokeilla esimerkiksi: regex101 ja debuggex sivustoilla.

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, params char[])
```

Aliohjelma muuttaa annetun merkkijonon kokonaislukutaulukoksi siten, että luvut erotellaan annetun merkkitaulukon (erotinmerkkien) perusteella. Syötteen tulee sisältää vain lukuja ja erotinmerkkejä.

```
int LaskeYhteen(int[])
```

Palauttaa annetun kokonaislukutaulukon alkioiden summan.

```
bool OnkoVainLukuja(String, params 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.

```
1  using System;
2
3  /// @author  Antti-Jussi Lakanen
4  /// @version 22.12.2011
5  ///
6  /// <summary>
7  /// Harjoitellaan merkkijonojen pilkkomista.
8  /// </summary>
9  public class MjLuvuiksi
10 {
11     /// <summary>
12     /// Kysellaan kayttajalta merkkijonoja ja
13     /// tehdaan niista taulukkoja, lasketaan lukuja yhteen ja tulostellaan.
14     /// </summary>
15     public static void Main()
16     {
17         char[] erottimet = new char[] { ' ', ',' };
18         Console.Write("Anna positiivisia kokonaislukuja > ");
19         String lukusyote = Console.ReadLine();
20         // String lukusyote = "23 555 77,,  99";
21         Console.WriteLine(lukusyote);
22
23
24         // Jos kayttaja antanut jotain muuta kuin positiivisia
25         // kokonaislukuja, ei yritetakaan laskea lukuja yhteen
26         if (OnkoVainLukuja(lukusyote, erottimet))
27         {
28             int[] luvut = MerkkijonoLuvuiksi(lukusyote, erottimet);
29             Console.WriteLine("Tulkittiin luvut:");
30             TulostaTaulukko(luvut);
```

```
31              Console.WriteLine("Antamiesi lukujen summa on : " +
32                                LaskeYhteen(luvut));
33          }
34          else
35              Console.WriteLine("Annoit muuta kuin lukuja, tai tyhjan jonon");
36      }
37
38
39      /// <summary>
40      /// Aliohjelma muuttaa annetun merkkijonon
41      /// kokonaislukutaulukoksi siten, etta luvut
42      /// erotellaan annetun merkkitaulukon (erotinmerkkien)
43      /// perusteella. Syötteen tulee sisältää vain
44      /// lukuja ja erotinmerkkejä.
45      /// </summary>
46      /// <param name="lukusyote">Muunnettava merkkijono</param>
47      /// <param name="erottimet">Sallitut erotinmerkit merkkitaulukossa</param>
48      /// <returns>Merkkijonosta selvitetty kokonaislukutaulukko.</returns>
49      /// <example>
50      /// <pre name="test">
51      /// int[] luvut1 = MerkkijonoLuvuiksi("1 2 3",' ');
52      /// String.Join(",", luvut1) === "1,2,3";
53      /// int[] luvut2 = MerkkijonoLuvuiksi(",,1,, 2 ,3", ' ', ',');
54      /// String.Join(",", luvut2) === "1,2,3";
55      /// int[] luvut3 = MerkkijonoLuvuiksi("", new char[] {' '});
56      /// String.Join(",", luvut3) === "";
57      /// </pre>
58      /// </example>
59      public static int[] MerkkijonoLuvuiksi(string lukusyote,
60                                             params char[] erottimet)
61      {
62          // Tyhjat pois edesta ja lopusta (Trim)
63          // Jos on annettu ylimaaraisia valilyonteja, ei lisata niita taulukkoon.
64          String[] pilkottu = lukusyote.Trim().Split(erottimet,
65                              StringSplitOptions.RemoveEmptyEntries);
66          int[] luvut = new int[pilkottu.Length]; // luvut[] saa kookseen saman ↩
        kuin
67                                                  //  pilkottu[]
68          for (int i = 0; i < pilkottu.Length; i++)
69              luvut[i] = int.Parse(pilkottu[i]);
70          return luvut;
71      }
72
73
74      /// <summary>
75      /// Laskee kokonaislukutaulukon alkiot yhteen ja palauttaa alkioiden summan.
76      /// </summary>
77      /// <param name="luvut">Tutkittava kokonaislukutaulukko</param>
78      /// <returns>Taulukon alkioiden summa</returns>
79      /// <example>
80      /// <pre name="test">
81      /// int[] luvut1 = {5, 7, 9, 10};
82      /// LaskeYhteen(luvut1) === 31;
83      /// int[] luvut2 = {-5, 5, -10, 10};
84      /// LaskeYhteen(luvut2) === 0;
85      /// int[] luvut3 = {};
86      /// LaskeYhteen(luvut3) === 0;
87      /// </pre>
88      /// </example>
```

```
89      public static int LaskeYhteen(int[] luvut)
90      {
91          int summa = 0;
92          for (int i = 0; i < luvut.Length; i++)
93              summa += luvut[i];
94          return summa;
95      }
96
97
98      /// <summary>
99      /// Aliohjelmassa tutkitaan sisaltaako merkkijono muitakin
100     /// merkkeja kuin positiivisia kokonaislukuja ja erotinmerkkeja.
101     /// </summary>
102     /// <param name="lukusyote">Tutkittava merkkinojo,
103     /// josta etsitaan vieraita merkkeja</param>
104     /// <param name="erottimet">Sallitut erotinmerkit
105     /// merkkitaulukossa</param>
106     /// <returns>Onko pelkkiä lukuja</returns>
107     /// <example>
108     /// <pre name="test">
109     /// OnkoVainLukuja("1,2,3", ',') === true;
110     /// OnkoVainLukuja("1, 2, 3", ',') === false;
111     /// OnkoVainLukuja("1, 2, 3", ',', ' ') === true;
112     /// OnkoVainLukuja("", ' ') === false;
113     /// </pre>
114     /// </example>
115     public static bool OnkoVainLukuja(string lukusyote, params char[] erottimet)
116     {
117         // Jos yhtaan merkkia ei ole annettu,
118         // palautetaan automaattisesti kielteinen vastaus.
119         if (lukusyote == null || lukusyote.Length == 0) return false;
120         for (int i = 0; i < erottimet.Length; i++)
121             // Korvataan erotinmerkit tyhjalla merkkijonolla,
122             // silla olemme kiinnostuneita vain "varsinaisesta sisallosta"
123             lukusyote = lukusyote.Replace(erottimet[i].ToString(), "");
124
125         foreach (char merkki in lukusyote)
126             // Jos yksikin merkki on jokin muu kuin numero,
127             // palautetaan kielteinen vastaus.
128             if (!Char.IsDigit(merkki)) return false;
129         return true;
130     }
131
132
133     /// <summary>
134     /// Tulostetaan kokonaislukutaulukon osat foreach-silmukassa
135     /// </summary>
136     /// <param name="t">Tulostettava taulukko</param>
137     public static void TulostaTaulukko(int[] t)
138     {
139         foreach (int pala in t)
140             Console.WriteLine(pala);
141         Console.WriteLine("-------------------------------");
142     }
143 }
```

17.4 Komentoriviparametrit

Kun ohjelma käynnistetään komentoriviltä, sille voidaan voidaan antaa käynnistyksen yhteydessä argumentteja. Näitä argumentteja voidaan hyödyntää ohjelman ajon aikana. C#-ohjelmassa nämä argumentit ovat saatavilla pääohjelman `args`-merkkijonotaulukossa:

```
1  /// @author  Vesa Lappalainen
2  /// @version 22.1.2015
3  ///
4  /// <summary>
5  /// Käytetään komentorivin parametreja
6  /// </summary>
7  public class ArgsEsimerkki {
8     /// <summary>
9     /// Tulostetaan kaikki komentorivillä annetut argumentit
10    /// </summary>
11    /// <param name="args">Argumentit komentoriviltä</param>
12    public static void Main(string[] args)
13    {
14        System.Console.WriteLine("Argumentteja on " + args.Length + " kappaletta:"↩
      );
15        for (int i=0; i<args.Length; i++)
16           System.Console.WriteLine(i + ": " + args[i]);
17    }
18 }
```

Jos edellisen esimerkin ohjelma on käännetty ja se ajetaan komentoriviltä, niin sen käynnistyskomennon (ohjelman nimen) perään voidaan kirjoittaa pääohjelmalle menevät parametrit:

```
C:\MyTemp\oma>ArgsEsimerkki kissa istuu puussa
```

Esimerkiksi kutsussa:

```
copy oma.txt oma.vara
```

ohjelma `copy` saa kaksi parametria: `oma.txt` ja `oma.vara` ja tekee niillä tiedoilla mitä sen täytyy tehdä, tässä tapauksessa kopioi tiedoston toiseksi tiedostoksi.

Luku 18

Järjestäminen

Kuinka järjestät satunnaisessa järjestyksessä olevat tuotteet hinnan mukaan järjestykseen halvimmasta kalleimpaan?

Yksi tutkituimmista ohjelmointiongelmista ja algoritmeista on järjestämisalgoritmi. Esimerkiksi, kuinka saamme korttipakan kortit numerojärjestykseen tai vaikkapa verkkokaupan hinnat pienimmästä suurimpaan. Yksinkertainen esimerkki ohjelmoinnin kannalta voisi olla järjestää taulukollinen `int`-lukuja. Vaikka aluksi tuntuu, ettei erilaisia tapoja järjestämiseen ole kovin montaa, on niitä todellisuudessa kymmeniä, ellei satoja, ja toiset ovat erittäin paljon parempia (mittarina on usein olla nopeus, mutta myös intuitiivisuus, lukemisen tai ymmärtämisen helppous voivat olla mittareita) kuin toiset.

Järjestämisalgoritmeja käsitellään enemmän muilla kursseilla (esim. ITKA201 Algoritmit 1, ITKA201 Algoritmit 2 ja TIEP111 Ohjelmointi 2). Tässä vaiheessa meille riittää, että osaamme käyttää C#:sta valmiina löytyvää (staattista) järjestämismetodia `Sort`.

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.

```
1         int[] taulukko = {-4, 5, -2, 4, 5, 12, 9};
2         Array.Sort(taulukko);
3
4         // Tulostetaan alkiot, että nähdää onnistuiko järjestäminen.
5         Console.WriteLine(String.Join(" ",taulukko));
```

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

```
1         int[] taulukko = {-4, 5, -2, 4, 5, 12, 9};
2         Array.Sort(taulukko, 0, 3);
3
4         // Tulostetaan alkiot, että nähdää onnistuiko järjestäminen.
5         Console.WriteLine(String.Join(" ",taulukko));
```

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

Kaikkia alkeistietotyyppisiä taulukoita voidaan järjestää `Sort`-aliohjelmalla. Lisäksi voidaan jär-

jestää taulukoita, joiden alkioiden tietotyyppi *toteuttaa* (implements) `IComparable`-rajapinnan. Esimerkiksi `String`-luokka toteuttaa tuon rajapinnan. Rajapinnoista puhutaan lisää kohdassa 23.1.

Luku 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);
```

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

https://tim.jyu.fi/view/kurssit/jypeli/oliot/ulkonako#vari.

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.

Luku 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.

Parametreja 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, LiikutaPelaajaaVasemm
                     "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, jossa 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.

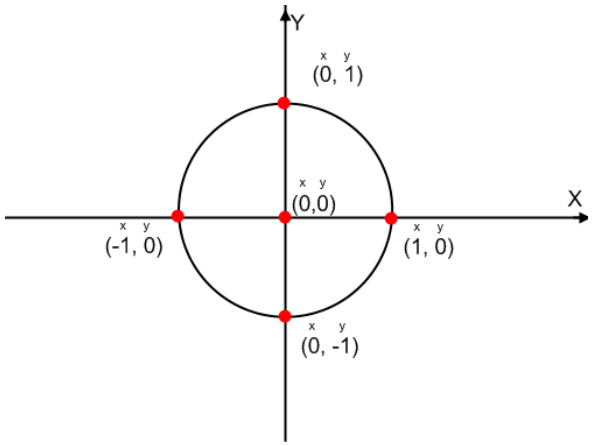
LiikutaPelaajaa-aliohjelman tulee ottaa vastaan seuraavanlainen parametri:

```
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://tim.jyu.fi/view/kurssit/jypeli/ohjaimet/ohjainten-lisays.

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 paikka 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 parametreina 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.Position = Mouse.PositionOnWorld;

    // Jos tarvittaisiin liikkeen koko, se saataisiin:
    Vector hiirenLiike = hiirenTila.MouseMovement;
    // ja sitten jatkettaisiin tämän käsittelyllä
}
```

Tässä esimerkissä luomamme fysiikkaolio nimeltä pallo seuraa hiiren kursoria. Käsittelijää kutsutaan aina kun hiirtä liikutetaan. ListenMovement:in parametreissa herkkyys (tässä 0.1) tar-

koittaa 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.

Tehtävä 20.1

Tee Visual Studiolla ohjelma, jossa olion kokoa voi muuttaa näpppäimillä. Laita luokan nimeksi Peli ja liitä tähän Peli.cs-tiedoston sisältö (ei sitä, missä on peli.Run()). Näppäimet eivät toimi, jos ohjelma ajetaan Timissä. 10pt

Luku 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. Parametreina 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.

```
1     protected override void Paint(Canvas canvas)
2     {
3       canvas.BrushColor = Color.Red;
4       double x = canvas.Left + 100, y = canvas.Top - 100;
5       canvas.DrawLine(new Vector(x - 50, y + 50), new Vector(x + 50, y - 50));
6       canvas.DrawLine(new Vector(x + 50, y + 50), new Vector(x - 50, y - 50));
7
8       canvas.BrushColor = Color.Black;
9       x = canvas.Right - 100;
10      y = canvas.Top - 100;
11      canvas.DrawLine(new Vector(x - 50, y + 50), new Vector(x + 50, y - 50));
12      canvas.DrawLine(new Vector(x + 50, y + 50), new Vector(x - 50, y - 50));
13
14      base.Paint(canvas);
15    }
```

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(ajan
    canvas.DrawLine(keskipiste, keskipiste + reunapiste);
    base.Paint(canvas);
}
```

Luku 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, tosin usein rekursio on funktio ja tuohon liittyy silloin myös arvon palautusta.

```
public static void Rekursio(parametrit)
{
    if (lopetusehto) return;
    // toimenpiteitä ...
    Rekursio(uudet parametrit);  // Itsensä kutsuminen
    // mahdollisesti lisää lauseita
}
```

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. Kertoma voidaan esittää rekursiivisesti `n! = n*(n-1)!, 0! = 1`. Iteratiivisesti aukilaskettuna esimerkiksi viiden kertoma on siis tulo `5*4*3*2*1`. Koska tässä tapauksessa rekursio on helppo purkaa iteraatioksi, ei rekursio välttämättä ole paras tapa laskea kertomaa C#:n kaltaisissa kielissä. Yksinkertainen esimerkki kuitenkin havainnollistaa rekursiota hyvin.

Kirjoitetaan kertoman laskeminen rekursiivisena C#-funktiona. Luonnollisesti laitamme mukaan myös ComTest-testit.

```
1 using System;
2 public class Rekursio
3 {
4     /// <summary>
5     /// Lasketaan luvun kertoma kaavasta
6     /// <code>
7     /// 0! = 1
8     /// 1! = 1
```

```
9     /// n! = n*(n-1)!
10    /// </code>
11    /// </summary>
12    /// <param name="n">Minkä luvun kertoma lasketaan</param>
13    /// <returns>n!</returns>
14    /// <example>
15    /// <pre name="test">
16    /// Kertoma(0) === 1;
17    /// Kertoma(1) === 1;
18    /// Kertoma(5) === 120;
19    /// </pre>
20    /// </example>
21    public static long Kertoma(int n)
22    {
23        if (n <= 1) return 1;
24        return n * Kertoma(n - 1);
25    }
26
27    /// <summary>
28    /// Pääohjelma
29    /// </summary>
30    public static void Main()
31    {
32        long k = Kertoma(5);
33        Console.WriteLine(k);
34    }
35 }
```

Funktio `Kertoma` saa parametrikseen luvun, jonka kertoma halutaan laskea. Funktio palauttaa `long`-tyypin, koska kertoma kasvaa niin nopeasti, että muuten ei voitaisi laskea kovinkaan monen luvun kertomaa. Tutustutaan aliohjelmaan tarkemmin.

```
if (n <= 1) return 1;
```

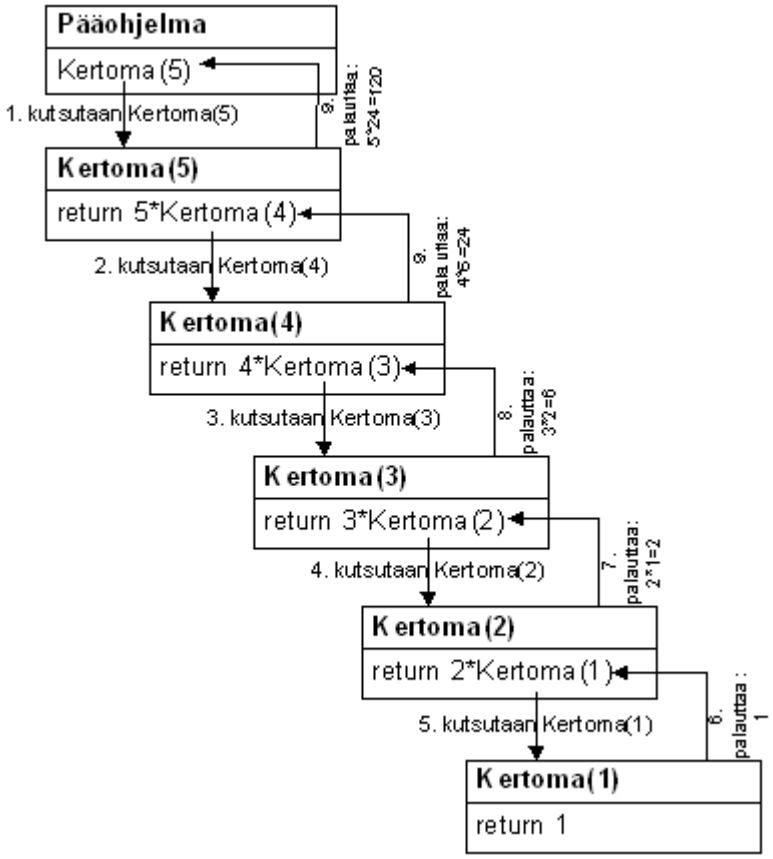
Yllä oleva rivi on ikään kuin rekursion lopetusehto. Jos n on pienempi tai yhtä suuri kuin 1, niin palautetaan luku 1. Oleellista on, että lopetusehto on ennen uutta rekursiivista aliohjelmakutsua.

```
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]

Animaatio: Suorita animaatio rekursiosta

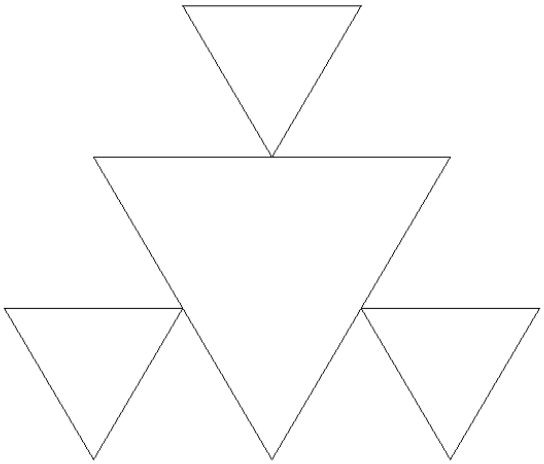
Askella rekursiota vihreällä nuolella. Tutki kertomaa

Animaatio: Suorita Python animaatio rekursiosta

Askella rekursiota vihreällä nuolella. Tutki for-silmukkaa

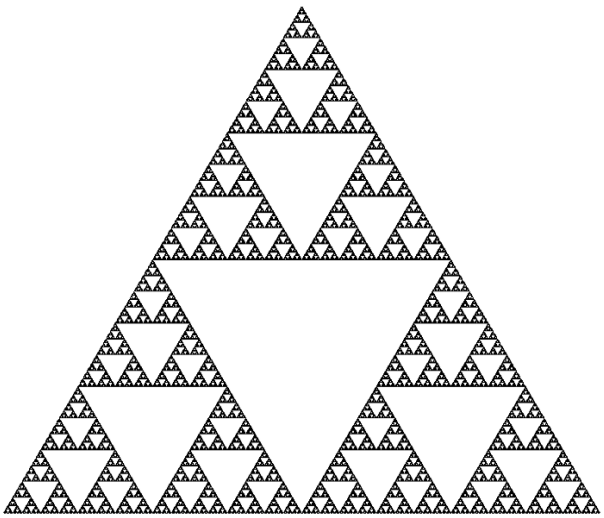
22.1 Sierpinskin kolmio

Sierpinskin kolmio on puolalaisen matemaatikko *Waclaw Sierpinskin* 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 korkeus on puolet suuremman kolmion korkeudesta. 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 olisi 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.

```
1  using System;
2  using Jypeli;
3
4  /// <summary>
5  /// Sierpinskin kolmio
6  /// </summary>
7  public class Peli : Game
8  {
9      private static double pieninKorkeus = 10.0;
10
11     public override void Begin()
12     {
13         Level.Background.Color = Color.White;
14     }
15
16     protected override void Paint(Canvas canvas)
17     {
18         base.Paint(canvas);
19         double korkeus = 200;
20         SierpinskinKolmio(canvas, 0, -korkeus, korkeus);
21     }
22
23     /// <summary>
24     /// Piirtää Sierpinskin kolmion.
25     /// </summary>
26     /// <param name="canvas">Piirtoalusta</param>
27     /// <param name="x">Alareunan x</param>
28     /// <param name="y">Alareunan y</param>
29     /// <param name="h">Korkeus</param>
30     public static void SierpinskinKolmio(Canvas canvas,
31                          double x, double y, double h)
32     {
33         if (h < pieninKorkeus) return;
34
```

```
35          double s2 = h / Math.Sqrt(3); // sivun pituus s/2
36          Vector p1 = new Vector(x, y);
37          Vector p2 = new Vector(x - s2, y + h);
38          Vector p3 = new Vector(x + s2, y + h);
39          canvas.DrawLine(p1, p2);
40          canvas.DrawLine(p2, p3);
41          canvas.DrawLine(p3, p1);
42
43          SierpinskinKolmio(canvas, x - s2, y, h / 2);
44          SierpinskinKolmio(canvas, x + s2, y, h / 2);
45          SierpinskinKolmio(canvas, x, y + h, h / 2);
46      }
47  }
```

Kokeile muuttaa yllä olevassa koodissa `pieninKorkeus = 100;` jolloin saat 4 kolmiota. Kokeile myös pienempiä arvoja, esimerkiksi 50 (tulee 13 kolmiota) ja vaikka 5 ja 1. Kokeile myös pistää kommentteihin vuorotellen kutakin erikseen tai kaksi kerralla noista kolmesta `SierpinskinKolmio`-kutsusta. Mieti ensin millaisen kuvan saat, paina vasta sitten Aja-painiketta.

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 myös 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 parametreina 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 riittävät parametreina 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.

$$
\begin{aligned}
s^2 &= h^2 + (\frac{s}{2})^2 \\
s^2 - \frac{s^2}{4} &= h^2 \\
\frac{3s^2}{4} &= h^2 \\
s^2 &= \frac{4}{3}h^2 \\
s &= \sqrt{\frac{4h^2}{3}} = \frac{2h}{\sqrt{3}}, s \geq 0, h \geq 0
\end{aligned}
$$

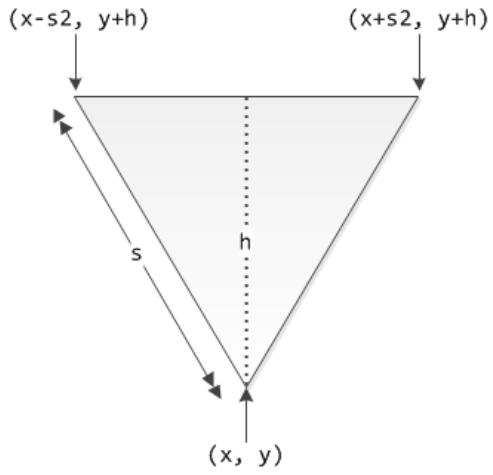
Koska x-akselilla siirrymme kolmion alimmasta kärjestä puolen sivun mitan verran joko vasemmalle tai oikealle, on mielekästä jakaa sivun pituus s vielä kahdella, jotta laskut hieman helpottuvat jatkossa.

$$
\frac{s}{2} = \frac{h}{\sqrt{3}}
$$

Tämä tulos 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. Alla oleva 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?

22.3 Huomautus

Myös sellainen aliohjelma (esimerkiksi aliohjelma A) on rekursiivinen, joka kutsuu toista aliohjelmaa (esimerkiksi aliohjelmaa B), joka puolestaan kutsuu aliohjelmaa A. Tällaisia tilanteita ei kuitenkaan tällä kurssilla käsitellä.

22.4 Rekursio muilla ohjelmointikielillä

Kurssilla *TIEA341 Funktio-ohjelmointi* opetellaan ohjelmoimaan käyttäen funktionaalisia ohjelmointikieliä. Monissa funktiokielissä rekursiota käytetään lähestulkoon kokonaan korvaamaan silmukat. Näiden kielten kääntäjät pystyvät usein optimoimaan rekursiivisia ohjelmia paremmin kuin C#, eikä rekursio tuota oikein käytettynä samankaltaisia suorituskykyongelmia kuin C#:ssa.

Seuraavana pieni esimerkki käyttäen Haskell-nimistä funktio-ohjelmointikieltä:

```
sum []    = 0
sum (x:xs) = x + sum xs
```

Yllä on määritelty funktio, joka laskee listan alkioiden summan. Tämä funktio on määritelty kahdessa palassa. Ensimmäinen näistä kertoo, että tyhjän listan (`[]`) summa on nolla. Toinen sääntö kertoo, että listan, jossa on vähintään yksi alkio (merkittynä muuttujalla `x`), summa on `x`:n ja loppulistan (merkittynä muuttujalla `xs`) summan summa.

Aukilaskettuna tämä funktio toimisi näin:

```
sum [63,25,27]
 Seuraavaksi lasketaan summan toisen säännön mukaan, x:=63, xs:=[25,27]
63 + sum [25,27]
 Taas summan toinen sääntö, x:=25, xs:=[27]
63 + (25 + sum [27])
 Summan toinen sääntö, x:=27, xs:=[]
63 + (25 + (27 + sum []))
 Summan ensimmäinen sääntö
63 + (25 + (27 + 0))
 Lopuksi lasketaan yhteenlasku
115
```

Yllä lasketaan listan `[63,25,27]` summa käsin. Olemme merkinneet aukilaskennassa sekä välitulokset että selitykset siitä, miten laskenta etenee, helpottamaan lukemista.

Luku 23

Dynaamiset tietorakenteet

Dynaaminen tietorakenne on tietorakenne, jonka koko voi muuttua ohjelman suorituksen aikana.

Taulukot ovat hyvä tietorakenne moneen käyttöön. Taulukoiden ongelmana on kuitenkin niiden koon staattisuus, eli kun taulukko on syntynyt, se on koko elinaikansa ajan saman kokoinen. Tämä toimii hyvin niin kauan kuin etukäteen voidaan tietää tilantarve. Mikäli tilantarvetta ei voida ennakoida etukäteen, tarvitaan dynaamisempia tietorakenteita. Eli sellaisia, joiden koko voi muuttua niiden elinkaaren aikana.

Mietitäänpä vaikka tilanne, jossa meidän tarvitsisi laskea käyttäjän syöttämien lukujen keskimmäinen, eli mediaani. Käyttäjä saisi syöttää niin monta lukua kuin haluaa ja lopuksi painaa `enter`, jolloin meidän täytyisi järjestää luvut ja tulostaa näytölle käyttäjän syöttämien lukujen mediaani. 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#:ssa on dynaamisia tietorakenteita eli kokoelmia. Niiden koko kasvaa sitä mukaa 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 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. `Generic`-kokoelmaluokat nimiavaruudessa

```
System.Collections.Generic
```

ovat tyyppiturvallisia, toisin sanoen kokoelman jäsenten (ja mahdollisen avaimen) tyyppi voidaan määritellä. Nimiavaruudessa

```
System.Collections.ObjectModel
```

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. Tässä geneerisyys tarkoittaa sitä, että tietorakenne kykenee tallentamaan mitä tahansa tietotyyppiä, joka sille on etukäteen ilmoitettu. `List<T>` muistuttaa jonkin verran taulukkoa; taulukoilla ja listoilla on paljon yhteistä:

- Niissä voi olla vain yhden tyyppisiä alkioita (tai saman rajapinnan toteuttavia oliota)
- Yksittäiseen alkioon päästään käsiksi laittamalla alkion paikkaindeksi hakasulkujen sisään, esimerkiksi `luvut[15]`, tai `pallot[4]`.
- Molemmilla on metodeja (funktioita, aliohjelmia) sekä ominaisuuksia
- Merkittävä ero on että listaan voidaan lisätä ja poistaa alkioita.
- Taulukon pituus, eli alkioiden lukumäärä, saadaan `Length`-ominaisuudella ja listan `Count`-ominaisuudella.

`List<T>`-olioon (kuten taulukkoonkin) 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<TalletettavienOlioidenTyyppi> rakenteenNimi =
  new TietorakenneLuokanNimi<TalletettavienOlioidenTyyppi>();
```

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:

```
1         elokuvat.Add("Casablanca");
2         elokuvat.Add("Star Wars");
3         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.

```
1         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.

```
1         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.

```
1         Console.WriteLine(elokuvat.Count); //tulostaa 3
2         elokuvat.Add("Full Metal Jacket");
3         Console.WriteLine(elokuvat.Count); //tulostaa 4
```

Tiettyyn alkioon pääsee käsiksi taulukon tapaan, eli laittamalla haluttu paikkaindeksi hakasulkeiden sisään. Ensimmäisen alkion voisi tulostaa esimerkiksi seuraavaksi:

```
1         Console.WriteLine(elokuvat[0]); // tulostaa "Casablanca"
```

Näillä metodeilla pärjää jo melko hyvin. Muista metodeista voi lukea List<T>-luokan dokumentaatiosta:
https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.

Tehdään toinen esimerkki int-tyyppisillä luvuilla. Tässä esimerkissä annetaan listalle sisältö heti listaa alustettaessa, joka on mielekästä, kun listan sisältö on tiedossa alustusta tehdessä. Muussa tapauksessa lista on järkevämpää alustaa tyhjäksi ja täyttää sen mukaan, kun tarve vaatii.

```
1         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.

```
1         List<int> luvut = new List<int>() { 3, 3, 1, 7, 3, 5, 7 };
2         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, mikä saattaisi sitten myöhemmin aiheuttaa vakavia ongelmia.

Tarkistetaan vielä listan sisältämien alkioiden lukumäärä.

```
1         Console.WriteLine(luvut.Count); // Tulostaa 7
```

Poistetaan sitten kaikki ne alkiot, joiden arvo on 3. Tässä voimme käyttää hyväksemme `while`-silmukkaa ja `Remove`-funktiota. `Remove`-funktion totuusarvotyyppinen paluuarvo käy hyvin `while`-ehdoksi.

```
1         while (luvut.Remove(3)); // huono!
```

Listan alkiot näyttävät tämän jälkeen seuraavalta.

```
1 7 5 7
```

Tosin edellisen ratkaisun kompleksisuus on $O(n^2)$, eli tuhannen alkion taulukolle jouduttaisiin tekemään karkeasti 1 000 000 operaatiota, kun esimerkiksi `RemoveAll` on $O(n)$, eli se olisi tähän tapaukseen merkittävästi parempi. Merkinnästä `=>` lue lisää luvusta Anonyymit funktiot (lambda-lausekkeet).

```
1         luvut.RemoveAll(alkio => alkio==3);
```

Metodi `Add` lisäsi aina alkion listan loppuun. Muuhun kohtaan listassa voi lisätä metodilla `Insert`.

```
1         luvut.Insert(2, 99);  // lisätään keskelle
2         luvut.Insert(0, 88);  // lisätään alkuun
```

Listan alkiot näyttävät tämän jälkeen seuraavalta.

```
88 1 7 99 5 7
```

Mitä tapahtuisi jos edellä `Insert` lauseiden järjestys vaihdettaisiin keskenään?

23.2.3 Listan sisäinen rakenne

Sisäisesti C#:issa ja Javassa vastaava lista on toteutettu niin, että siinä on pieni olio, joka ylläpitää listan tämän hetkistä alkioiden määrää (`count`) ja viite taulukkoon, jossa on itse alkiot. Taulukolle on aluksi varattu joku koko ja jos tila loppuu, niin silloin luodaan uusi isompi taulukko, johon kopioidaan nykyisen taulukon alkiot. Tämän jälkeen sisäinen viite käännetään siihen. Itse asiassa myös `StringBuilder` on toteutettu vastaavalla tavalla. Katso ideaa seuraavasta animaatiosta [>] nuolella liikkumalla. Mikäli listasta poistetaan alkioita, ei välttämättä tilaa pienennetä.

Lista muuttaa kokoaan automaattisesti

Koska tuollainen kuva on aika työläs piirtää ja useinkaan ei tarvitse ajatella mitä listalle sisäisesti tapahtuu piirretään listan kuvat jatkossa yksinkertaisemmin, kuten tehtiin `StringBuilder`-luokan tapauksessakin. Merkitään kuitenkin käytettyjen alkioiden määrä johonkin erotukseksi taulukosta. Tärkeää on kuitenkin ymmärtää, että samaan listaan voi olla useampiakin viitteitä ja silloin jos lista-olioita muutetaan jonkin viitteen kautta, muuttuu se luonnollisesti toisenkin viitteen perspektiivistä. Tätä käytetään erityisesti kun listoja (tai mitä tahansa muita olioita) viedään parametreina aliohjelmille ja aliohjelma haluaa muuttaa listaa.

Eli jos meillä olisi edellisten lauseiden lisäksi lopussa sijoitus

```
List<int> lista2 = lista:
```

olisi kuva seuraavan näköinen:

23.2.4 Esimerkki listaa käsittelevästä funktiosta ja sen testaamisesta

Listat voidaan käydä silmukassa läpi kuten taulukotkin käyttäen sopivaa indeksiin perustuvaa silmukaa jos indeksiä tarvitaan. Mikäli indeksiä ei tarvita, on `foreach` usein sujuvin vaihtoehto.

```
1     public static int LaskeSanat(List<string> sanat, int n)
2     {
3         int lkm = 0;
4         foreach (string sana in sanat)
5             if ( sana.Length == n ) lkm++;
6         return lkm;
7     }
```

```
1     public static void Main()
2     {
3         List<string> sanat = new List<string>{ "kissa", "kana", "koira", "mato" ↩
      };
4         Poista(sanat,4);
5         Console.WriteLine("Muutettu lista: " + String.Join(" ", sanat));
6     }
7
8
9     /// <summary>
10    /// Poistetaan listasta kaikki sanat, joissa on n kirjainta.
11    /// Palautetaan sanojen lukumäärä.
12    /// </summary>
13    /// <param name="sanat">lista jota muutetaan</param>
14    /// <param name="">minkä mittaiset sanat poistetaan</param>
15    /// <returns>montako sanaa poistettiin</returns>
16    /// <example>
17    /// <pre name="test">
18    ///    List<string> sanat = new List<string>{ "kissa", "kana", "koira", "mato↩
      " };
19    ///    Poista(sanat,4) === 2;
20    ///    String.Join(" ", sanat) === "kissa koira";
```

```
21      ///     sanat = new List<string>{ "kissa", "kotka", "koira" };
22      ///     Poista(sanat,5) === 3;
23      ///     String.Join(" ", sanat) === "";
24      /// </pre>
25      /// </example>
26      public static int Poista(List<string> sanat, int n)
27      {
28          int lkm = 0;
29          int i = 0;
30          while (i < sanat.Count)
31          {
32              if ( sanat[i].Length == n )
33              {
34                  lkm++;
35                  sanat.RemoveAt(i);
36              }
37              else i++;
38          }
39          return lkm;
40      }
```

Edellä indeksiä `i` ei ole kasvatettu normaaliin tapaan `for`-silmukan kasvatuslausekkeessa. Mieti miksi!

Edellä oleva tapa on kuitenkin tehoton suuren alkiomäärän poistamiseksi, koska jokaisessa poistossa loppuja alkioita siirretään taaksepäin. Tehokkaampi tapa on käyttää `RemoveAll`-metodia, jonka käyttö kuitenkin edellyttää predikaatti-funktion tekemistä. Predikaattifunktio on sellainen, joka saa yhden alkion ja palauttaa sen perusteella `true` tai `false` sen mukaan, halutaanko alkio käsitellä vaiko ei. Alla olevassa esimerkissä predikaattifunktio on toteuttettu Lambda-lausekkeena, joista enemmän seuraavassa aliluvussa.

```
1       public static int Poista(List<string> sanat, int n)
2       {
3           return sanat.RemoveAll(sana => sana.Length == n);
4       }
```

Lambda-funtiota käyttäen myös edellinen sanojen laskeminen onnistuisi lyhyemmin:

```
1       public static int LaskeSanat(List<string> sanat, int n)
2       {
3           return sanat.Count(sana => sana.Length == n );
4       }
```

23.3 Anonyymit funktiot (lambda-lausekkeet)

Listoja ja taulukoita sekä muita tietorakenteita voidaan käsitellä silmukoiden lisäksi myös `List<T>`-luokan metodeilla. Monet näistä metodeista ottavat parametrina aliohjelman, jonka avulla listaa käsitellään. Tällaisia metodeja ovat esimerkiksi `Find`, `FindIndex`, `Exists`, `FindAll` ja `ForEach`. Näille metodeille voidaan antaa argumenttina aliohjelman osoite metodin kutsun kaarisulkeiden sisään. Valmiilla metodeilla käsittely silmukoiden sijaan mahdollistaa usein vastaavan asian tekemisen huomattavasti pienemmällä määrällä koodia.

Usein helpoin keino antaa näille metodeille parametreja on tehdä aliohjelmat *lambda-lausekkeina*,

jossa parametrina annettava aliohjelma määritellään nimettömästi suoraan parametrilausekkeen sisään. Esimerkiksi jos meillä on lista peliolioita (`List<GameObject>`), jonka nimi on `lista`, seuraava kutsu etsisi listasta ensimmäisen olion, joka on ympyrän muotoinen, ja asettaisi sen muuttujaan `pallo`:

```
GameObject pallo = lista.Find(olio => olio.Shape == Shape.Circle);
```

Ylläolevassa koodissa kohta `olio => olio.Shape == Shape.Circle` on lambda-lausekkeella toteutettu anonyymi funktio, joka ottaa yhden parametrin (`olio`). Lambda-lausekkeessa parametrit määritellään ennen nuolta `=>`. Funktio palauttaa lausekkeen `olio.Shape == Shape.Circle` arvon, eli `true` mikäli parametrina annetun olion muoto on ympyrä, ja muuten `false`. Listan `Find`-metodi suorittaa tämän sille lambda-lausekkeella parametrina annetun aliohjelman jokaiselle listan alkiolle, kunnes löytyy alkio, jolle lambda-lausekkeella määritelty aliohjelma palauttaa `true`.

Vastaava koodi silmukalla tehtynä olisi seuraavanlainen:

```
GameObject pallo;
foreach (GameObject olio in lista)
{
    if (olio.Shape == Shape.Circle)
    {
        pallo = olio;
        break;
    }
}
```

Lambda-lausekkeet käyttäytyvät kuin normaalit aliohjelmat ja funktiot, mutta niillä ei ole nimeä ja niihin ei voi viitata muualla koodissa. Tosin lambda-lausekkeen voi sijoittaa muuttujaan ja tätä kautta sitä voi tarvittaessa käyttää muualla koodissa.

Seuraavaksi esitellään olennaisia `List<T>` luokan metodeja listojen käsittelyyn esimerkkien kera. Suuri osa esitetyistä esimerkeistä toimii myös taulukolle C#-kielessä.

23.3.1 `Find`

Kuten edellä mainittiin, listan `Find`-metodi ottaa parametrina funktion, joka palauttaa `bool`-arvon. `Find` palauttaa listan ensimmäisen alkion, jolle annettu aliohjelma palauttaa `true`. Käytännössä siis `Find`-metodille annetaan parametriksi ehto, ja se etsii listasta ensimmäisen ehdon täyttävän alkion. Mikäli ehdon täyttävää alkiota ei löydy, niin `Find` palauttaa arvon `null`. Yksi esimerkki `Find`-metodin käytöstä annettiin edellisessä luvussa. Sama esimerkki vielä jos halutaan etsiä merkkijonot, jotka ovat pidempiä kuin `n`:

```
1    public static string EtsiMerkkijono(List<string> lista, int n)
2    {
3        return lista.Find(jono => jono.Length > n);
4    }
```

Sama silmukalla:

```
1    public static string EtsiMerkkijono(List<string> lista, int n)
2    {
```

```
3         foreach (string jono in lista)
4             if (jono.Length > n) return jono;
5
6         return null;
7     }
```

Find-metodin dokumentaatio MSDN:ssä

23.3.2 Delegaatit ja Lambda-lausekkeet

Alkuperäinen idea on, että `Find` saa parametrinaan predikaattifunktion osoitteen. Predikaattifunktio palauttaa totuusarvon sen mukaan, toteuttaako kohdalla oleva alkio halutun ehdon. Tehdään aluksi edellinen siten, että tehdään erillinen funktio, joka tutkii jonon pituuden

```
1     public static string EtsiMerkkijono(List<string> lista, int n)
2     {
3         return lista.Find(OnkoYli4Pitka);
4     }
5
6
7     public static bool OnkoYli4Pitka(string jono)
8     {
9         return jono.Length > 4;
10    }
```

Eli nyt `Find` käy jokaisen listan alkion kohdalla kutsumassa (predikaatti)funktiota `OnkoYli4Pitka` ja mikäli funktio palauttaa tosi, todetaan että alkio löytyi ja `Find` palauttaa kohdalla olevan alkion. Tämän ratkaisun vika on siinä, että ei saada helpolla tavalla vietyä parametrina kuinka pitkiä jonoja halutaan tutkia. Eli ratkaisu toimii, jos tyydytään siihen, että predikaattifunktio saadaan toimimaan vaan itse tutkittavalla alkiolla.

C#-kielessä voidaan käyttää myös aliohjelmien sisäisiä aliohjelmia, jotka pääsevät käsiksi "ulkoaliohjelman" muuttujiin. Silloin tämä voitaisiin kirjoittaa:

```
1     public static string EtsiMerkkijono(List<string> lista, int n)
2     {
3         bool OnkoYliNPitka(string jono)
4         {
5             return jono.Length > n;
6         }
7         return lista.Find(OnkoYliNPitka);
8     }
```

Sinällään tässä ei ole mitään vikaa, paitsi että sisäfunktiolle joudutaan aina keksimään oma nimi.

C#-kielessä seuraava ratkaisu olisi käyttää nimettömiä funktioita, delegaatteja, siten että luodaan funktio siihen kohti, jossa sitä tarvitaan:

```
1     public static string EtsiMerkkijono(List<string> lista, int n)
2     {
3         return lista.Find(delegate(string jono) { return jono.Length > n; });
4     }
```

Tämä jo helpottaa paljon kirjoittamista. Mutta vielä tulee jonkin verran turhia sanoja. Ja siksi onkin Lambda-lausekkeet, jotka karkeasti ovat synonyymejä edelliselle eli:

```
delegate(string jono) { return jono.Length > n; }
```

voidaan lyhyemmin kirjoittaa:

```
jono => jono.Length > n
```

Delegaattiin verrattuna Lambda-lausekkeen kiva puoli on myös se, että tyypeistä ei tarvitse huolehtia.

23.3.3 FindIndex

`FindIndex`-metodi toimii kuten edellä mainittu `Find`-metodi, mutta se palauttaa itse alkion sijaan indeksin. Esimerkiksi seuraava aliohjelma ottaa vastaan listan merkkijonoja, etsii listasta ensimmäisen sellaisen merkkijonon, jonka pituus on enemmän kuin 5 merkkiä, ja palauttaa sen indeksin.

```
1    public static int EtsiMerkkijononIndeksi(List<string> lista, int n)
2    {
3        return lista.FindIndex(jono => jono.Length > n);
4    }
```

Vastaava aliohjelma toteutettuna silmukalla näyttäisi seuraavalta:

```
1    public static int EtsiMerkkijononIndeksi(List<string> lista, int n)
2    {
3        for (int i = 0; i < lista.Count; i++)
4            if (lista[i].Length > n) return i;
5
6        return -1;
7    }
```

FindIndex-metodin dokumentaatio MSDN:ssä

23.3.4 Exists

`Exists`-metodi tarkistaa, löytyykö listasta tietyn ehdon täyttävää alkiota. Mikäli alkio löytyy, `Exists` palauttaa `true`, muuten `false`. Esimerkiksi seuraava aliohjelma tarkistaa, onko kokonaislukulistassa (`List<int>`) olemassa lukua, joka on suurempi kuin 10.

```
public static bool OnkoSuurempaaKuin10(List<int> lista)
{
    return lista.Exists(luku => luku > 10);
}
```

```
1    public static bool OnkoSuurempaaKuin(List<int> lista, int n)
2    {
3        return lista.Exists(luku => luku > n);
4    }
```

Vastaava aliohjelma toteutettuna silmukalla voisi näyttää esimerkiksi seuraavalta:

```
    public static bool OnkoSuurempaaKuin(List<int> lista, int n)
    {
        foreach (int luku in lista)
            if (luku > n) return true;
        return false;
    }
```

Yleinen virhe on kuitenkin innostua asiasta ja lähteä tekemään sama asia kaksi kertaa:

```
        List<int> luvut = new List<int>() { 3, 3, 1, 7, 3, 5, 7 };
        if (luvut.Exists(luku => luku > 3)) {
           int luku3 = luvut.Find(luku => luku > 3);
           Console.WriteLine($"Ainakin {luku3} on suurempi kuin 3.");
        }
        else
           Console.WriteLine("Yksikään luku ei ole suurempi kuin 3.");

        // Tämä on järkevämpi tehdä suoraan hakemalla:
        int i = luvut.FindIndex(luku => luku > 9);
        if (i >= 0)
            Console.WriteLine($"Ainakin {luvut[i]} on suurempi kuin 9.");
        else
           Console.WriteLine("Yksikään luku ei ole suurempi kuin 9.");
```

Miksi? Koska ensimmäinen versio käy lukua etsiessään taulukon kertaalleen läpi. Ja sitten jos luku löytyy, niin se käy taulukon uudelleen läpi löytääkseen sen luvun. Jälkimmäinen versio käy taulukon vain kerran läpi.

Indeksiä tuossa on käytetty, koska kokonaislukulistan `Find` palauttaa 0 jos lukua ei löydy ja silloin tulisi ristiriita jos taulukossa oli myös 0-alkioita. Esimerkiksi merkkijonolistasta `Find` palauttaa `null` mikäli tietoa ei löydy ja tämä on helppo erottaa oikeista alkioista. Eli oliolistoille ja -taulukoille suora `Find` käyttö on edellisissä tapauksissa aivan käyttökelpoinen valinta.

Exists-metodin dokumentaation MSDN:ssä

23.3.5 FindAll

`FindAll`-metodi toimii kuten `Find`-metodi, mutta palauttaa listan, joka sisältää kaikki ne alkiot, jotka täyttävät annetun ehdon, kun `Find` palauttaa alkioista vain ensimmäisen. `FindAll` sisällyttää tulokset uuteen listaan, jonka se palauttaa. Esimerkiksi seuraava aliohjelma etsii ja palauttaa pelioliolistasta (`List<GameObject>`) kaikki punaiset suorakulmiot. Esimerkki näyttää samalla, miten `Find`- ja `FindAll`-metodille annettava parametri voi tarkistaa useamman ehdon.

```
public static List<GameObject> HaePunaisetSuorakulmiot(List<GameObject> lista)
{
    return lista.FindAll(olio => olio.Shape == Shape.Rectangle && olio.Color == Color.
}
```

Vastaava aliohjelma toteutettuna silmukalla näyttäisi seuraavalta:

```
public static List<GameObject> HaePunaisetSuorakulmiot(List<GameObject> lista)
{
    List<GameObject> tulokset = new List<GameObject>();

    foreach (GameObject olio in lista)
```

```
            if (olio.Shape == Shape.Rectangle && olio.Color == Color.Red)
                tulokset.Add(olio);

        return tulokset;
    }
```

Merkkijonolistaesimerkki:

```
1     public static List<string> EtsiPituudenMukaan(List<string> lista, int n)
2     {
3         return lista.FindAll(jono => jono.Length > n);
4     }
```

Ja sama silmukalla:

```
1     public static List<string> EtsiPituudenMukaan(List<string> lista, int n)
2     {
3         List<string> tulos = new List<string>();
4
5         foreach (string jono in lista)
6             if (jono.Length > n) tulos.Add(jono);
7
8         return tulos;
9     }
```

FindAll-metodin dokumentaatio MSDN:ssä

23.3.6 Usean lauseen sisältävät anonyymit funktiot

Listan `ForEach`-metodilla (älä sekoita `foreach`-silmukkaan) pystyy suorittamaan jonkin aliohjelman listan jokaiselle alkiolle. Esimerkiksi seuraava aliohjelma vaihtaa kaikkien yli 35 yksikköä korkeiden peliolioiden värin keltaiseksi ja lyö niitä ylöspäin.

```
1     public static void VaihdaVari(List<PhysicsObject> lista)
2     {
3         lista.ForEach(olio =>
4             {
5                 if (olio.Height > 35.0)
6                 {
7                     olio.Color = Color.Yellow;
8                     olio.Hit(new Vector(0.0, 100.0));
9                 }
10            }
11        );
12    }
```

Useimmat lambda-lausekkeilla tehdyt anonyymit funktiot ovat yksinkertaisia ja sisältävät vain yhden lauseen tai lausekkeen. Tällöin nuolen oikealle puolelle tuleva funktion toteutus ei tarvitse aaltosulkuja `{ }` koodinsa ympärille kuin tavalliset aliohjelmat. Yllä olevan esimerkin mukaisesti lambda-lausekkeilla tehdyt aliohjelmat voivat tosin sisältää useammankin lauseen. Tällöin koodin ympärille tulee aaltosulut vastaavasti kuin tavallistenkin aliohjelmien ympärille. Lambda-funktiossa voi myös käyttää kaikkia tavallisia C#-kielen ominaisuuksia, kuten ehtolauseita.

Vastaava esimerkki silmukoilla toteutettuna:

```
    public static void VaihdaVari(List<PhysicsObject> lista)
    {
        foreach (var olio in lista)
            if (olio.Height > 35.0)
            {
                olio.Color = Color.Yellow;
                olio.Hit(new Vector(0.0, 100.0));
            }
    }
```

ForEach-metodin dokumentaatio MSDN:ssä

23.3.7 Lambda-lausekkeen ulkopuolisten muuttujien käyttö

Lambda-funktiot voivat myös muokata itsensä ulkopuolella määriteltyjä paikallisia muuttujia. Esimerkiksi kokonaislukulistan alkioiden yhteenlasku toteutettaisiin seuraavasti:

```
    public static void Main()
    {
        List<int> luvut = new List<int>() { 3, 3, 1, 7, 3, 5, 7 };
        int summa = 0;
        luvut.ForEach(luku => summa += luku);

        Console.WriteLine($"Summa = {summa}");

        // Toki summan saa helpomminkin:
        Console.WriteLine($"Summa = {luvut.Sum()}");
    }
```

23.3.8 Muita yleisiä valmiita metodeja

Monessa tapauksessa listan funktioille annetaan parametrina predikaattifunktio. Predikaattifunktio on sellainen, joka palauttaa totuusarvon `true` tai `false`. Predikaattifunktiolla, joka usein toteutetaan lambda-lausekkeena, määritellään käsitelläänkö kohdalla olevaa alkiota vai ei.

```
    public static void Main()
    {
        List<int> luvut = new List<int>() { 3, 3, 1, 7, 3, 5, 7 };
        List<int> luvut2 = new List<int>() { 4, 1, 0, 2 };

        Console.WriteLine("Summa = {0}", luvut.Sum());

        // ehto ? a; b  palauttaa a jos ehto on totta, muuten b
        Console.WriteLine("Summa yli 3 = {0}", luvut.Sum(a => a > 3 ? a : 0));

        var yli3 = luvut.Where(a => a > 3);
        Console.WriteLine("yli 3 lkm = {0}", yli3.Count());
        Console.WriteLine("yli 3 = {0}", String.Join(" ", yli3));

        // voidaan myös suoraan laskea lkm:
        Console.WriteLine("yli 3 lkm = {0}", luvut.Count(a => a > 3));

        Console.WriteLine("min = {0}", luvut.Min());
```

```
19          Console.WriteLine("max = {0}", luvut.Max());
20
21          var plus3 = luvut.Select(x => x + 3);
22          Console.WriteLine("luvut +3 = {0}", String.Join(" ", plus3));
23
24          // Aggregage käy läpi kaikki alkio aloittaen niin, että
25          // acc apumuutuja alustetaan alkuarvolla (esimerkissä 1)
26          // ja sitten joka kierroksella saadaan käyttöön nykyinne acc ja luku
27          // ja sitten tulos sijoitetaan uudeksi acc-muuttujan arvoksi
28          int tulo = luvut.Aggregate(1, (acc, n) => acc*n);
29          Console.WriteLine("Tulo = {0}", tulo);
30
31          tulo = luvut.Aggregate(1, (acc, n) => n > 5 ? acc*n: acc);
32          Console.WriteLine("Tulo yli 5 olevista = {0}", tulo);
33
34          var luvut3 = luvut.Zip(luvut2, (a,b) => a + b);
35          Console.WriteLine("Summa = {0}", String.Join(" ", luvut3));
36
37          var sisatulo = luvut.Zip(luvut2, (a,b) => a * b).Sum();
38          Console.WriteLine("Sisätulo = {0}", sisatulo);
39      }
```

Mikäli lambda-lausekkeet herättävät enemmänkin kiinnostusta, niihin voi tutustua syvemmin MSDN:ssä.

Luku 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.

```
1         int[] taulukko = new int[5];
2         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
{
   //lauseita, joita yritetään 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.

```
1        int n1 = 7, n2 = 0, n3 = 4;
2
3        try
4        {
5           Console.WriteLine("{0}", 10 / n1);
6           Console.WriteLine("{0}", 10 / n2);
7           Console.WriteLine("{0}", 10 / n3);
8        }
9        catch (DivideByZeroException e)
10       {
11          Console.WriteLine("Nollalla jako: " + e.Message);
12       }
```

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 ovat 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
{
   //lauseita, joita yritetään 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. 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
{
   //lauseita, joita yritetään 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.

Luku 25

Tietojen lukeminen ulkoisesta lähteestä

Muuttujat toimivat tiedon talletuksessa niin kauan kuin 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 Mobiili- ja Xbox-ympäristöissä jätetään tämän kurssin osaamistavoitteiden ulkopuolelle.

25.1 Tekstin lukeminen tiedostosta

System.IO-nimiavaruus sisältää muun muassa tiedostojen käsittelyyn tarvittavia aliohjelmia. Seuraavassa esimerkissä luetaan tekstitiedostosta tietoa sekä kirjoitetaan tiedostoon.

```
Kalle, 5
Pekka, 10
Janne, 0
Irmeli, 15
```

Näiden voidaan ajatella olevan vaikka topten-listan pisteitä. Annetaan tiedoston nimeksi data.txt.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

/// <summary>
/// Harjoitellaan tiedostoon kirjoittamista
/// sekä tiedostosta lukua.
/// </summary>
public class TiedostostaLuku
```

```
{
    /// <summary>
    /// Luetaan ja kirjoitetaan tekstitiedostosta.
    /// </summary>
    public static void Main()
    {
        // Määritellään tiedostopolku vakioksi
        const string POLKU = @"C:\MyTemp\data.txt";

        // Jos tiedostoa ei ole olemassa, luodaan se ja kirjoitetaan sinne
        if (!File.Exists(POLKU))
        {
            // Taulukon alkiot ovat tiedoston rivejä
            string[] uudetRivit = { "A-J, 0", "Pekka, 0", "Kalle, 0" };
            File.WriteAllLines(POLKU, uudetRivit);
        }

        // Tämä teksti lisätään jokaisella ajokerralla,
        // jolloin tiedosto pitenee aina yhdellä rivillä
        string appendText = "Tämä on ylimääräinen rivi" + Environment.NewLine;
        File.AppendAllText(POLKU, appendText);

        // Avataan tiedosto ja kirjoitetaan sen sisältö ruudulle
        string[] luetutRivit = File.ReadAllLines(POLKU);
        foreach (string s in luetutRivit)
        {
            Console.WriteLine(s);
        }
    }
}
```

Tutkitaan tärkeimpiä kohtia hieman tarkemmin

```
        if (!File.Exists(POLKU))
```

Tarkistetaan, onko tiedostoa olemassa. Mikäli ei ole, kirjoitetaan polun osoittamaan tiedostoon muutama nimi, ja nimien perään pilkku sekä “pistemäärä”.

```
        File.WriteAllLines(POLKU, uudetRivit);
```

File-luokka sisältää WriteAllLines-metodin, joka kirjoittaa kaikki String-taulukon alkiot annettuun tiedostoon.

```
        File.AppendAllText(POLKU, appendText);
```

`AppendAllText`-metodi lisää annettuun tiedostoon String-olion sisältämän tekstin siten, että teksti tulee tiedoston loppuun.

```
        string[] luetutRivit = File.ReadAllLines(POLKU);
```

ReadAllLines-metodi lukee annetusta polusta kaikki rivit String-taulukkoon. Yksi tiedoston rivi vastaa yhtä taulukon alkiota.

Huomaa, että mikäli tiedostolle ei ole annettu absoluuttista polkua, ohjelma hakee tiedostoa

suhteessa siihen kansioon, missä ajettava exe-tiedosto on. Tässä esimerkissä annettiin tiedoston absoluuttinen polku kokonaisuudessaan.

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.

```
1  using System;
2  using System.Net.Http;
3
4  /// @author  vesal
5  /// @version 11.9.2022
6  ///
7  /// <summary>
8  /// Tulostetaan kayttajan antaman www-osoitteen
9  /// sisältö (GET-pyynnön palauttama HTML-koodi).
10 /// </summary>
11 public class TiedotNetista
12 {
13     /// <summary>
14     /// Haetaan tiedot netistä listaan ja tulostetaan listan sisältö.
15     /// </summary>
16     public static void Main()
17     {
18         try
19         {
20             string url = "http://users.jyu.fi/~vesal/kurssit/ohj1/elukat.html";
21             string html = LueNetista(url);
22             string[] rivit = html.Split('\n');
23             foreach (string rivi in rivit)
24             {
25                 // if (rivi.Contains("kissa")) // ota kommentti pois jos haluat ↩
       vain kissa-rivit
26                 Console.WriteLine(rivi);
27             }
28         }
29         catch (NotSupportedException e) { Console.WriteLine(e.Message); }
30         catch (HttpRequestException e) { Console.WriteLine(e.Message); }
31     }
32
33     /// <summary>
34     /// Luetaan annetun url-osoitteen koko html-koodi
35     /// ja palautetaa se yhtenä merkkijonoja
36     /// </summary>
37     /// <param name="url">URL-osoite, mika halutaan lukea.</param>
38     /// <returns>html-koodi</returns>
39     public static string LueNetista(string url)
40     {
41         HttpClient client = new HttpClient();
42         string content = client.GetStringAsync(url).GetAwaiter().GetResult();
43         return content;
44     }
45 }
```

Mukana on yksi try-catch -rakenne. Verkkoyhteydet ovat alttiita kaikenlaisille virheille, joten

poikkeusten “kiinniottaminen” on perusteltua ja suorastaan välttämätöntä.

25.3 Satunnaisluvut

Random-luokasta tehdyn olion avulla voi arpoa näennäisesti satunnaisia (ns. pseudo-satunnaisia) lukuja.

Arpomista varten 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[.

Random-olion saa luodan **vain yhden kerran** ohjelman aikana, muuten voi pilata satunnaisuuden.

Alla oleva koodinpätkä arpoo luvun 0:n ja 10:n väliltä niin, että luvut 0 ja 10 kuuluvat arvottaviin lukuihin.

```
1         Random rand = new Random();
2         int satunnaisluku = rand.Next(11);
```

Jos haluaisimme arpoa luvun esimerkiksi suljetulta väliltä [50, 99], sanoisimme

```
1         Random rand = new Random();
2         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/

Luku 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 tietokeen käyttämä 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. Heksajärjestelmän yleisin käyttö on esittää ihmiselle binäärijärjestelmän lukuja lyhyemmin luettavassa muodossa.

Lukujärjestelmä	Käytettävät merkit	Kantaluku
Binäärijärjestelmä	0 1	2
Oktaalijärjestelmä	0 1 2 3 4 5 6 7	8
Desimaalijärjestelmä	0 1 2 3 4 5 6 7 8 9	10
Heksajärjestelmä	0 1 2 3 4 5 6 7 8 9 A B C D E F	16

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.

Heksajärjestelmän lukua voidaan merkitä esimerkiksi D_{16}, DH tai joissakin ohjelmointikielissä etuliitteellä `0x`, eli 0xD. Jos kaikki tietävät muusta yhteydestä että käytetään heksajärjestelmää, niin sillon voidaan puhua vaan luvusta D (joka siis on 13 kymmenjärjestelmässä). Vastaavastihan normaalielämässä ilman erillistä merkintää puhutaan kymmenjärjestelmän luvuista.

Kaikissa yllä mainituissa lukujärjestelmissä symbolin paikalla on oleellinen merkitys. Kun symboleja laitetaan peräkkäin, ei siis ole yhdentekevää, millä paikalla luvussa tietty symboli on. [MÄN]

Tarkista tietosi

	True	False
Binäärijärjestelmän kantalukuja ovat 0 ja 1	☐	☐
Oktaalijärjestelmän kantaluku on 8	☐	☐
0xA on heksajärjestelmän luku.	☐	☐
Kantaluku ilmoittaa montako symbolia järjestelmässä on käytössä.	☐	☐
0B on binääriluku.	☐	☐
Heksaluku A vastaa 10-järjestelmän lukua 10	☐	☐

26.2 Paikkajärjestelmät

Käyttämämme lukujärjestelmät ovat paikkajärjestelmiä, eli jokaisen numeron paikka luvussa on merkitsevä. Jos numeroiden paikkaa luvussa vaihdetaan, muuttuu luvun arvokin. Luvun

$n_3 n_2 n_1 n_0$

arvo on

$n_3 \cdot k^3 + n_2 \cdot k^2 + n_1 \cdot k^1 + n_0 \cdot k^0$

missä k on käytetyn järjestelmän kantaluku. Esimerkiksi 10-järjestelmässä:

$2536_{10} = 2 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0$
$= 2 \cdot 1000 + 5 \cdot 100 + 3 \cdot 10 + 6 \cdot 1$

Sanomme siis, että luvussa 2536 on 2 kappaletta tuhansia, 5 kappaletta satoja, 3 kappaletta kymmeniä ja 6 kappaletta ykkösiä.

Jos luvussa olevat symbolien paikat 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

$25.36 = 2 \cdot 10^1 + 5 \cdot 10^0 + 3 \cdot 10^{-1} + 6 \cdot 10^{-2}$
$= 2 \cdot 10 + 5 \cdot 1 + 3 \cdot 0.1 + 6 \cdot 0.01$

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 virtoina tai jännitteinä. Tietty jänniteväli (esimerkiksi 0-2 V) vastaa arvoa 0 ja tietty jänniteväli (esimerkiksi 4-5 V) arvoa 1.

- Katso miten seuraavaan binäärilukuun päästään Video (25s)

Katso edellistä videota ja jatka alle sen avulla 16 ensimmäistä binäärilukua10pt

```
0000
0001
```

Tarkista tietosi

Mitkä seuraavista voisi olla binäärilukuja:

	True	False
1	☐	☐
0111B 01000	☐	☐
01001101 01101111 01101001B	☐	☐
02B	☐	☐
0 2	☐	☐

Alla on esimerkki miten virtalähde (DC, *direct current*, tasasähkö), painike ja led (*light emiting diode*) voisivat muodostaa yhden bitin "tietokoneen". Jos kytkin on painettu, led palaa. Jos ei ole painettu, led ei pala. Informaatio (eli yksi bitti) on se tieto, onko kytkimen jälkeen jännitettä vaiko ei ja led näyttää tuon tiedon ihmiselle.

Vastaavasti meillä voisi olla kahden bitin tietokone, jossa on kaksi painiketta. Tässä esimerkissä painikkeet on vaihdettu "lukkiintuviksi", jolloin tilat saadaan helpommin säilymään. Näillä voisimme esittää jo 4 erilaista tilaa. Kokeile!

26.3.1 Binääriluku 10-järjestelmän luvuksi

Esimerkiksi binääriluku 10110 voidaan muuttaa 10-järjestelmän luvuksi seuraavasti.

$$10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 0 + 4 + 2 + 0 = 22_{10}$$

Desimaalijärjestelmässä luvun seuraava paikka on kymmenkertainen. Binärijärjestelmässä paikkojen arvo kasvaa kaksinkertaisesti.

Lukujärjestelmien kahdeksan ensimmäisen "bitin" arvo desimaalilukuna

n	7	6	5	4	3	2	1	0
Binärijärjestelmä 2^n	128	64	32	16	8	4	2	1
10-järjestelmä 10^n	10_000_0001 000 000		100 000	10 000	1000	100	10	1

Tarkista tietosi

Ovatko seuraavat muutokset tehty oikein:

	True	False
0 2 = 0 10	☐	☐
1 2 = 1 10	☐	☐
01 2 = 2 10	☐	☐
0000000001 2 = 1 10	☐	☐
10 2 = 2 10	☐	☐
11 2 = 3 10	☐	☐
1000 2 = 9 10	☐	☐
0110 0110 2 = 102 10	☐	☐

Tehtävä 26.1

Muunna seuraavat binääriluvut 10-järjestelmään10pt

```
1                0100 =
2                1111 =
3             1100100 =
4             1111111 =
5            11111111 =
6            00000000 =
7  10000000000000001 =
8  00000000000001111 =
```

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:

$$101.1011_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}$$
$$= 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 5.687510$$

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 lukemalla 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.

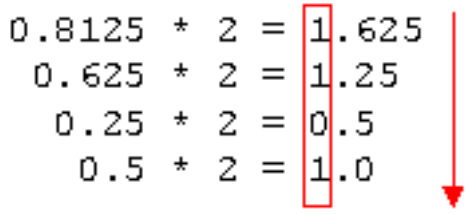
```
126 = 2*63+0
 63 = 2*31+1
 31 = 2*15+1
 15 =  2*7+1
  7 =  2*3+1
  3 =  2*1+1
  1 =  2*0+1
```

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 suurempi tai yhtä suuri 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.

Yritetään 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 $23.375_{10} = 10111.011_2$.

Tarkista tietosi

Ovatko seuraavat muutokset tehty oikein:

	True	False
5_{10} binääriluvuksi		
5/2 = 2, jakojäännös 1		
2/2 = 1, jakojäännös 0		
1/2 = 0, jakojäännös 1		
tulos: 101_2	☐	☐
50_{10} binääriluvuksi		
50 / 2 = 25, jakojäännös 0		
25 / 2 = 12, jakojäännös 1		
12 / 2 = 6, jakojäännös 0		
6 / 2 = 3, jakojäännös 0		
3 / 2 = 1, jakojäännös 1		
1 / 2 = 0, jakojäännös 1		
tulos: 010011_2	☐	☐
13.102_{10} binääriluvuksi		
13 / 2 = 6, jakojäännös 1		
6 / 2 = 3, jakojäännös 0		
3 / 2 = 1, jakojäännös 1		
1 / 2 = 0, jakojäännös 1		
0.102 * 2 = 0.204, 0		
0.204 * 2 = 0.408, 0		
0.408 * 2 = 0.816, 0		
0.816 * 2 = 1.632, 1		
1.632 * 2 = 3.264, 1		
3.264 * 2 = 6,528, 1		
tulos: 1101.111000	☐	☐

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

Useimmiten nykytietokoneissa käytetään negatiivisille luvuille niin sanottua kahden komplementtia. Eli positiivinen luku muutetaan negatiiviseksi muuttamalla kaikki bitit päinvastaisiksi ja sitten lisäämällä saatuun lukuun 1. Esimerkiksi:

```
3 =  0000 0011
-3 tehdään seuraavasti:  1)  kaikki päinvastoin 1111 1100
                         2)  +1                 = 1111 1101 = -3
```

Vastaavasti kun lukua muutetaan “ihmismuotoon”, katsotaan sen ensimmäinen bitti ja jos se on 1, niin kyseessä on negatiivinen luku ja se muutetaan positiiviseksi ottamalla siitä kahden komplementti (kaikki bitit päinvastoin ja +1). Tällöin tulostuksessa tulostetaan ensin -merkki ja sitten itse luvun arvo.

Esimerkiksi jos meillä on binääriluvut `0010 1101` ja `1101 1111` ja ne pitäisi tulkita, niin tulkinta aloitetaan seuraavasti:

```
0010 1101 luku on positiivinen, eli 45
1101 1111 luku on negatiivinen, siis ensin 2:n komplementti
           0010 0000 + 1 = 0010 0001 = 33, eli tulos on -33
```

Huom! Komplementin kääntämisen jälkeen tehtävä +1 lisäys tehdään myös alla olevien bittien yhteenlasku sääntöjen mukaan eli esim.

```
1111 1110 luku on negatiivinen, siitä ensin 2:n komplementti
           0000 0001 + 1 = 0000 0010 = 2, eli tulos on -2
```

Bittien yhteenlasku

```
    0 + 0 =  0  =>  0 ja 0 muistiin
    0 + 1 =  1  =>  1 ja 0 muistiin
    1 + 0 =  1  =>  1 ja 0 muistiin
    1 + 1 = 10  =>  0 ja 1 muistiin
1 + 1 + 1 = 11  =>  1 ja yksi muistiin
```

Esimerkki yhteenlaskusta allekkain 4-bittisillä luvuilla kaikki vastinbiteistä saadut muistinumerot merkiten. Esimerkissä muistinumero on merkitty myös oikeanpuoleiseen pariin vaikka se aina onkin 0.

```
               esim1         esim2
muistinumero  01110          11110
luku 1         0101           1111
luku 2      +  0011        +  1111
              =====          =====
summa          1000           1110
```

Vinkki binäärilukujen yhteenlaskuun

Tämän 2-komplementti esitystavan etuna on se, että yhteenlasku toimii totuttuun tapaan myös negatiivisilla luvuilla. Vähennyslasku suoritetaan summaamalla luvun vastaluku:

2-3 = 2+(-3)

Muunnetaan 3 negatiiviseksi luvuksi

```
luku 3:           0011
käännetään bitit: 1100
lisätään 1:       1101
```

Saatiin, että -3 on kahden komplementtina 1101.

Nyt voidaan laskea 2 + -3:

```
muistinumero   0000
                0010
              + 1101
                ====
                1111
```

Eli tulos on negatiivinen luku, koska se alkaa 1:llä. Siksi sen itseisarvon selvittämiseksi pitää tehdä etumerkin muunnos, eli

```
1-komplementti  0000
+1              0001
```

Vastaus on siis -1.

Voidaanko luvut muuttaa samalla menetelmällä takaisin positiivisiksi luvuiksi? Kokeile!

Tarkista tietosi

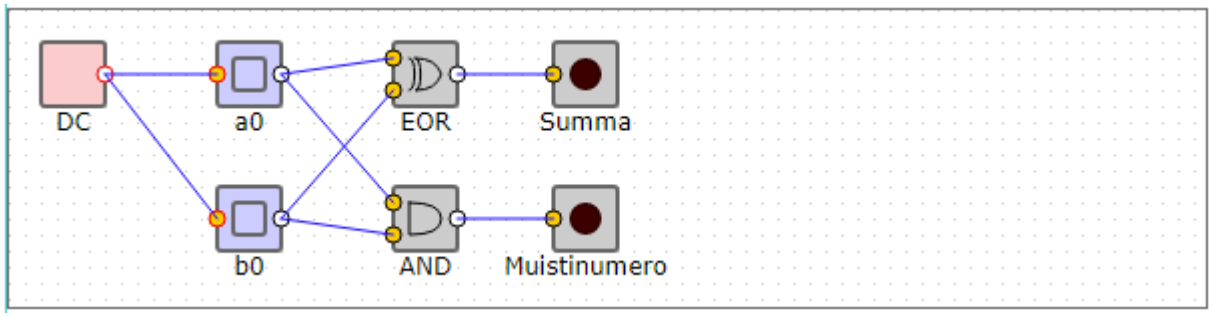
Ovatko seuraavat muutokset 2-komplementiksi tehty oikein:

	True	False
muutetaan luku 4 negatiiviseksi:		
luku 4: 0100_2		
käännetään bitit: 1011		
lisätään 1: 1100_2	☐	☐
-16 2-järjestelmässä:		
luku 16 : 0001 0000_2		
tulos: 1111 0000_2	☐	☐

26.4.4 Bittien yhteenlasku

Yhteenlasku on eräs tärkeimmistä alkeisoperaatioista. Jos aluksi tutkitaan kahta yhden bitin yhteenlaskua, niin jos molemmat bitit ovat 0, niin niiden summakin on 0. Jos toinen on 1, niin silloin summa on 1. Jos molemmat ovat 1, niin summa ei mahdu enää yhteen bittiin vaan tulos on 0 ja lähtee muistinumero seuraavaan bittiin. Operaatiot `xor` ja `and` vastaavat juuri tätä toimintoa. Totuustauluna:

Eli kaksi bittiä voidaan laskea yhteen alla olevalla kytkennällä (*Half adder*). Kokeile kaikki vaihtoehdot.

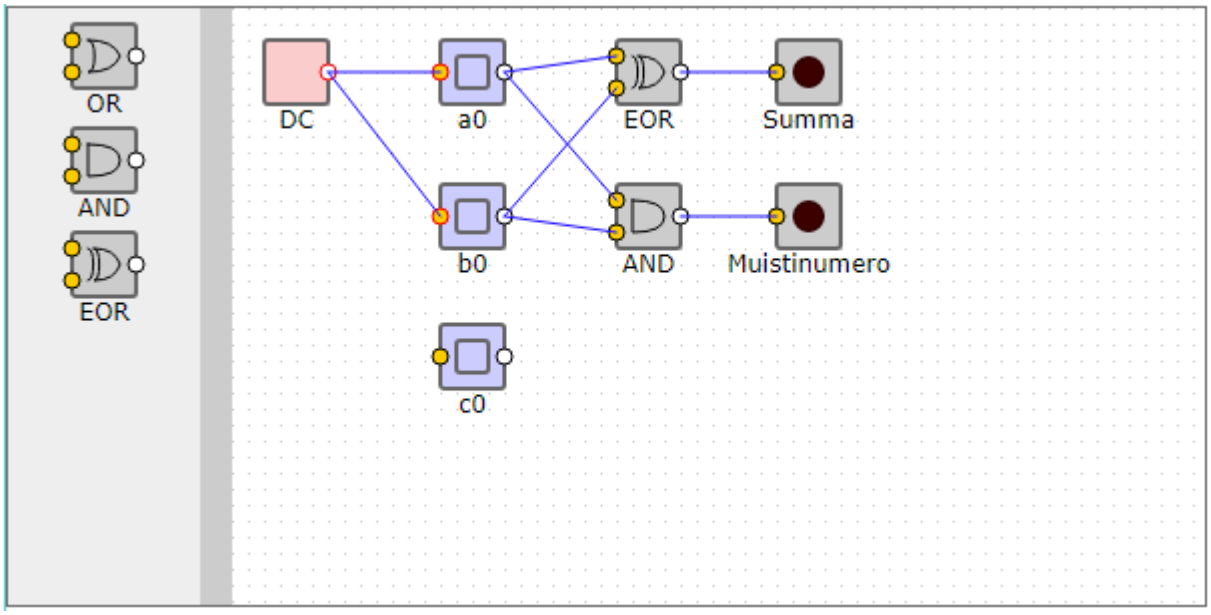


Halfadder

Mikäli haluttaisiin laskea yhteen kaksi kahden bitin lukua, tarvitaan siis kaksi kappaletta yllä olevia kytkentöjä. Tai oikeastaan sama vielä hieman monimutkaisemmassa muodossa, koska toiseen bittipariin pitää ottaa huomioon edellisestä tuleva muistinumero. Merkitään tällaista kykentaa (joka siis sisältää edellisen sekä tulevan muistinumeron huomioimisen) FullAdder-piirillä (FA), jonka totuustaulu on alla (c = tuleva muistinumero).

Tehtävä: Full adder

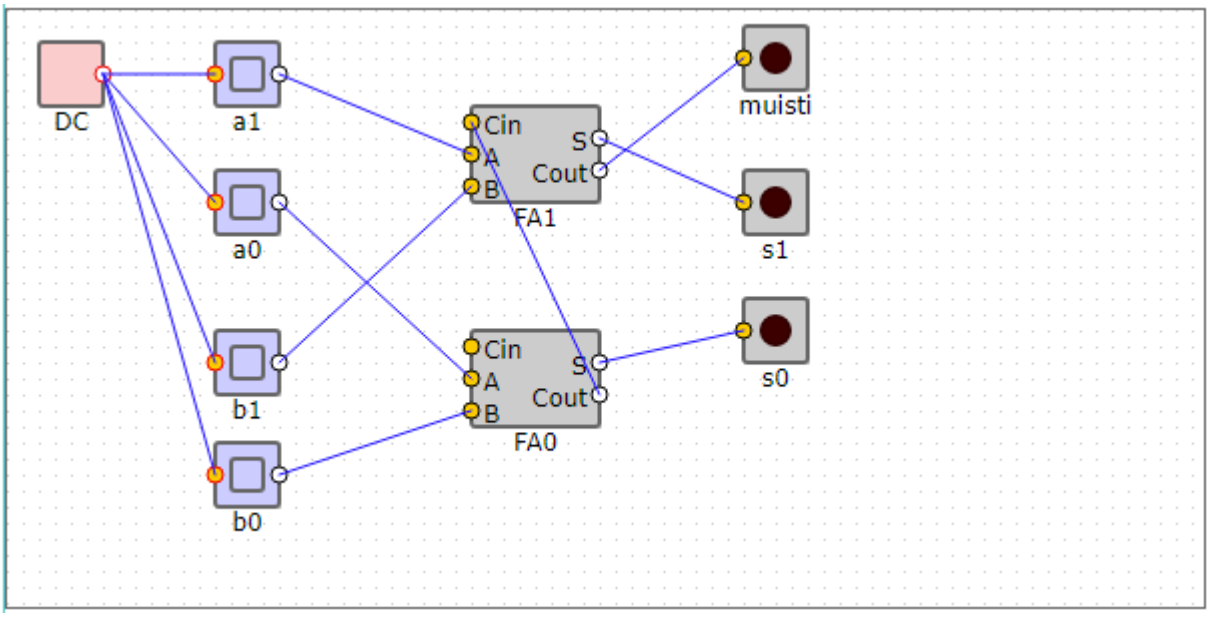
Muunna edellinen kytkentä Full adderiksi.10pt



Full adder

Silloin kaksi kahden bitin lukua voitaisiin laskea yhteen alla olevalla kytkennällä. Tässä 0-paikassa olevia bittejä yhteenlaskeva piiri ei saa muistinumeroa, eli se voisi olla kuten yllä tehty `Half adder`, mutta symmetriäsyistä kaikkien piirien kannattaa olla samanlaisia. Vastaavasti bittejä 1 yhteenlaskeva piiri saa (*Carry In*) bittien 0 yhteenlaskusta tulevan muistinumeron (*Carry out*). Kokeile kytkennällä eri laskuja

Lasku	bitteinä	tulos	desim
$0 + 0$	$00 + 00$	00	0
$0 + 1$	$00 + 01$	01	1
$1 + 2$	$01 + 10$	11	3
$2 + 2$	$10 + 10$	1 00	4
$3 + 3$	$11 + 11$	1 10	6



Full adder -ketju

Vastaavasti ketjuttamalla peräkkäin lisää Full adder -piirejä, saadaan tehtyä niin monibittinen yhteenlaskin kuin halutaan. Ongelmaksi vaan käytännössä muodostuu se, että lopputulos on valmis vasta kun muistibitti on kulkenut koko ketjun läpi ja siksi oikeasti tehdään "fiksumpia" yhteenlaskimia jotta muistinumeron kulkemista ei tarvitse odottaa.

Lue lisää porttipiireistä ja logiikasta

26.4.5 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 4-bitin "koneessa" esimerkiksi `2+1`:

```
 0000
  0010
+ 0001
 -----
  0011
```

Merkitsevin bitti on 0, joten vastaus on $0011_2 = 3_{10}$. Lasketaan seuraavaksi 1-2.

```
 0000
  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.

```
 1100
  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

```
 0110
  0110
+ 0111
  ----
  1101   => 0010 + 1  => -3 (siis neg. luku kahden pos. luvun yhteenlaskusta)
```

Vastaavasti -7-6

```
 1000
  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 bitillä 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 nykyisin useimmiten 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}

Tarkista tietosi

Mikä heksaluku vastaa kymmenjärjestelmän lukua 20?

	True	False
$5F_{16}$	☐	☐
13_{16}	☐	☐
14_{16}	☐	☐
5_{16}	☐	☐

Yhdellä 16-järjestelmän symbolilla voidaan siis esittää 4-bittinen binääriluku. Binääriluku voidaankin muuttaa heksajärjestelmän luvuksi järjestelemällä bitit oikealta alkaen neljän bitin ryhmiin ja käyttämällä kunkin 4 bitin yhdistelmän heksavastinetta. Muutetaan luku 11101101_2 heksajärjestelmään.

- 11101101_2 =1110 1101_2
- $1110_2 = E_{16}$
- $1101_2 = D_{16}$
- 11101101_2 =1110 $1101_2 = ED_{16}$

Vastaavasti voitaisiin muuttaa binääriluku 8-järjestelmän luvuksi, mutta nyt vain järjesteltäisiin bitit oikealta alkaen kolmen bitin ryhmiin.

Alla olevassa taulukossa on esitetty 10-, 2-, 8- ja 16-järjestelmän luvut $0_{10}..15_{10}$. Lisäksi on esitetty, mikä olisi vastaavan binääriluvun 2-komplementti -tulkinta.

Taulukko 9: 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

Tehtävä 26.2

Muunna seuraavat binääriluvut heksaluvuiksi10pt

```
//
                                    0010 0101 =
          1111 1111 1111 1111 1111 1111 1111 =
                           0001 0000 0010 0000 =
                           1010 1011 1100 1101 =
```

Tehtävä 26.3

Muunna seuraavat 10-järjestelmän luvut heksaluvuiksi ja binääriluvuiksi: Voit merkitä heksalukuja etumerkillä 0x ja binärilukuja loppumerkillä B.10pt

```
//
        24  = 0x18 = 11000B
         9  =
        10  =
        15  =
        16  =
        17  =
        19  =
        25  =
```

26.6 Liukuluku (floating-point)

Liukulukua käytetään 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:

$$x = (\text{-}1)^s \cdot m \cdot k^c$$

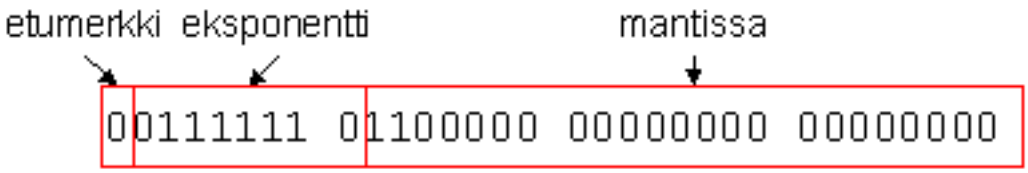
Tietotekniikassa yleisimmin käytetyssä standardissa IEE 754 kantaluku on 2, jolloin kaava saadaan muotoon:

$$x = (\text{-}1)^s \cdot m \cdot 2^c$$

IEEE 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.

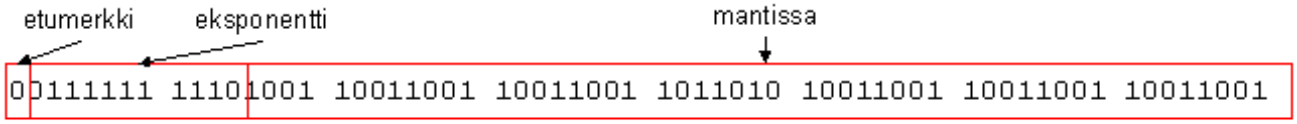
Tutustutaan seuraavaksi kuinka float ja double esitetään bittimuodossa.

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 floatissa 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:

$$x = (-1)^0 \cdot 1.0 \cdot 2^0 = 1.0$$

Kyseessä olisi siis reaaliluku 1.0. Kunhan muistetaan ottaa huomioon ensimmäinen bitti etumerkkinä, voidaan liukuluvun laskemiseen käyttää vielä yksinkertaisempaa kaavaa:

$$x = m \cdot 2^c$$

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:

$$1.75 \cdot 2^{-1} = 0.875$$

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:

$$x \cdot 2^0$$

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

$$m \cdot 2^c$$

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 :

$$0.1 = 0.1 \cdot 2^0$$

Nyt kerrotaan lukua kahdella kunnes se on välillä [1,2[ja muistetaan vähentää jokaisen kertomisen jälkeen eksponenttia yhdellä, jotta luvun arvo ei muutu.

$$0.1 = 0.1 \cdot 2^0 = 0.2 \cdot 2^{-1} = 0.4 \cdot 2^{-2} = 0.8 \cdot 2^{-3} = 1.6 \cdot 2^{-4}$$

Eksponentiksi saatiin -4, ja liukuluvun binääriesityksessä siihen lisätään BIAS, eli saadaan 10-järjestelmän luku -4 + 127 = 123, joka on binäärilukuna 01111011. 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: 01111011

- Mantissa: 10011001100110011001101

Eli yhdistämällä saadaan:

```
1 01111011 10011001100110011001101
```

Binääriluku voidaan vielä järjestellä tavuittain:

```
10111101 11001100 11001100 11001101
```

Intelin prosessoreissa on vähiten merkitsevä tavu ensin, eli muistissa tämä voisi olla muodossa:

```
11001101 11001100 11001100 10111101
```

Lukua 0.1 ei siis voi esittää liukulukuna tarkasti, vaan pientä heittoa tulee aina.

Leikkikalu reaalilukumuodon kokeilemiseksi

https://evanw.github.io/float-toy

- toinen vastaava työkalu

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 doublella (52 bittiä mantissalle) voi tarkasti käsitellä suurempia kokonaislukuja kuin int-tyypillä (32 bittiä luvun esittämiseen). long-tyypin 64-bitillä päästään vielä doublea 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 doublea 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).

Esimerkiksi seuraava ohjelma ei tulosta lukua 100 vaikka sen pitäisi:

```
        float s = 0;
        float d = 0.1f;
        for (int i=0; i<1000; i++) s += d;
        Console.WriteLine("{0:0.00000000}",s);
```

Jos epätarkan 0.1 tilalle vaihtaa tarkan 0.25, niin tulostuu tasan 250 kuten pitääkin.

Vielä pahempi tilanne on, mikäli lähdetään lisäämään pieniä lukuja isoon lukuun. Seuraavassa esimerkissä 10 miljoonaan lisätyt luvt eivät vaikuta mitään.

```
        float s = 10000000; // 10E6
        float d = 0.1f;
        for (int i=0; i<1000; i++) s += d;
        Console.WriteLine("{0:0.00000000}",s);
```

Tämän takia esimerkiksi sarja pitäisi laskea aloittaen summaaminen pienimmästä luvusta.

Katso myös Tietokoneen rakenne ja arkkitehtuuri -kurssin materiaali.

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 oikeasti häirinnyt tavallista toimistokäyttäjää millään tavalla. Tästä ja muista listan bugeista voi lukea lisää alla olevasta linkistä.

https://en.wikipedia.org/wiki/Pentium_FDIV_bug

Luku 27

ASCII-koodi

ASCII (American Standard Code for Information Interchange) on merkistö, joka käyttää seitsemän-bittistä 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 10: ASCII-merkistö.

Des	Hex	Merkki	Des	Hex	Merkki	Des	Hex	Mer	Des	Hex	Mer
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

Tässä käytetään ASCII-merkistön tulostuvia merkkejä ja tavallista välilyöntiä ja yritetään muodostaa jonkinlainen hahmo. Koita tehdä tilalle jotain muuta. 10pt

```
1
2         Tulosta("   ############");
3         Tulosta("   |          |");
4         Tulosta("   |          |");
5         Tulosta("   |          |");
6         Tulosta(" =================");
7         Tulosta("   |   9    9 |");
8         Tulosta("   |      >   |");
9         Tulosta("   |     .    | ^^^^^");
10        Tulosta("   \\-      - / /   /");
11        Tulosta("   / / \\     \\/   /");
12        Tulosta("   | |   |    |___/");
```

```
13        Tulosta("   | |   |    |");
14        Tulosta("   | |   |    |");
15        Tulosta("   |-vvvvv----|");
```

Tehtävä 27.1

Muunna '|'-merkkien sisällä oleva teksti heksaluvuiksi ja desimaaliluvuiksi (sen verran mitä jaksat). Ensimmäinen teksti on mallina. 10pt

```
1  //
2
3      ASCII                      Heksa                          Desimaali
4     |Help Help|               = 48 65 6c 70 20 48 65 6c 70 = 72 101 108 112 72 101 ↩
       108 112
5     |Hello World!|            =
6     |int a = 3;|              =
7     |p2.Y = p1.Y + 100 + 50;|=
```

Monissa ohjelmointikielissä, esimerkiksi C:ssä ja Javassa, ASCII-merkkien desimaaliarvoja voidaan sijoittaa suoraan char-tyyppisiin muuttujiin. Esimerkiksi pikku-a:n (a) voisi sijoittaa muuttujaan c seuraavasti:

```
char c = 97;
```

C#:ssa näin ei voi tehdä, vaan on tehtävä tyyppimuunnoksen kautta:

```
1
2         char c = (char)97;
3         System.Console.WriteLine(c);
```

Kaikki merkit ASCII-koodistossa eivät ole tulostettavia. 10pt

```
1
2     System.Console.Write((char)10);
3     System.Console.Write((char)9);
4     System.Console.Write((char)27);
```

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
```

Tehtävä 27.2

Muunna ‘|‘-merkkien sisällä oleva teksti desimaaliluvuiksi (ota valmis kohdasta 27.1 jos olet tehnyt sen) ja muunna desimaaliluvut vielä binääriluvuiksi.10pt

```
1 //
2
3    ASCII                      Desimaali                              Binääri
4    |Help Help|               = 72 101 108 112 72 101 108 112     =
5    |Hello World!|            =
6    |int a = 3;|              =
```

27.1 Muut merkistöt

Lue lisää merkistöistä.

Luku 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:

symboli	Selitys
<>	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 (complitatonunit) 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 lokaalin 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://cui.unige.ch/isi/bnf/JAVA/BNFindex.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.

symboli	Selitys
{}	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.

Yleisen muuttujan määrittelyn syntaksi voidaan kuvata EBNF:llä näin:

```
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/isi/bnf/JAVA/BNFindex.html

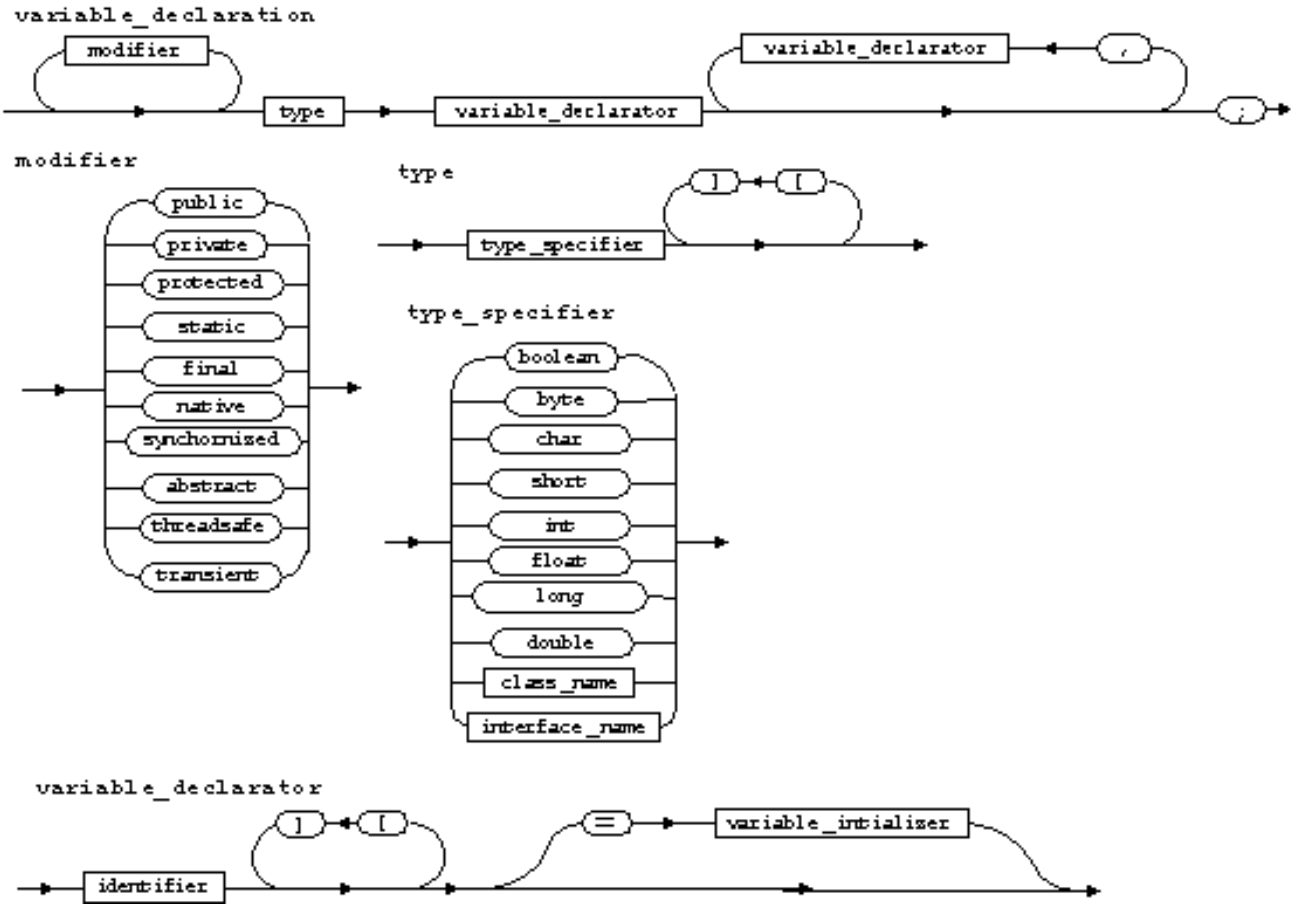
Tosin em. syntaksi ei vaikuta täydelliseltä. Virallinen Javan syntaksi löytyy eri metasyntaksilla osoitteesta:

https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

Vastaavasti syntaksia voidaan kuvata “junaradoilla”.

Yllä oleva syntaksi junaradoilla tehtynä

Tämä on eräs graafinen tapa kuvata syntaksia. Junaradoissa non-terminaalit on kuvattu suorakulmioilla ja terminaalit vähän pyöreämmillä suorakulmioilla. 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.



Kuva 37: Muuttujan määrittelyn syntaksia “junaradoilla” esitettynä

Kuvasta puuttuu vielä tekstiesimerkissä olevien identifier ja variable_initializer non-terminaalien junarataesitys. Piirrä niiden “junaradat” samaan tapaan.

Junanratoja voit piirrellä vaikkapa Railroad Diagram Generatorilla. Huomaa kuitenkin, että tuossa käytetään hieman eri syntaksia mm. toiston esittämiseen.

Lisätietoa:

- TIEA241 Automaatit ja kieliopit .
- Paavo Niemisen 2007 pitämän Ohjelmointi 1-kurssin luentokalvot. Junaratoja löytyy myös muista Paavon kalvoista.

- Javan syntaksi EBNF:nä kuvattuna. Sisältää myös graafiset junaradat. Tämä kuvaus ei ole ihan yhtä tarkka kuin Oraclen oma.
- Wikipedia: Bachus Naur Form
- C#:n syntaksia on kuvattu muun muassa MSDN-dokumentaatiossa ja EBNF-muodossa esim Extern Softin sivuilla.

Luku 29

Jälkisanat

Joskus ohjelmoidessa tulee vaan tämmöinen olo:

https://www.youtube.com/watch?v=GzhH900EjgE

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	koodaus-käytänteet	coding conventions	roskienkeruu	garbage collection
alivuoto	underflow	kääntäjä	compiler	roskien-kerääjä	garbage collector
alkeistieto -tyyppi	primitive types	kääriä	wrap	sijoitus-lause	assignment statement
alkio	element	lause	statement	sijoitus-operaattori	assignment operator
alustaa	initialize	lippu	flag	silmukka	loop
aritmeettinen operaatio	arithmetic operation	lohko	block	sovellus-kehitin	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 virhetulostus-virta	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
keskeytys-kohta	breakpoint	poikkeus	exception	vakio	constant

komentorivi	Command Prompt	poikkeusten-hallinta	exception handling	yksikkö-testaus-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.

Katso tämän luvun lisäksi kurssin lisätietosivuilta virheilmoituksia ja niiden tulkintoja.

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.

1	'Peli.LuoPallo()': not all code paths return a value	Peli.cs	13	19	FysiikkaPeli6

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 lokaalina (paikallisena) muuttujana, se pitää alustaa ennen sen käyttöä. Jos muuttujaa tarvitaan useammassa metodissa, voi sen esitellä attribuuttina luokan sisällä:

```
public class Peli : PhysicsGame
{
    private double massa; // Attribuutti, joka näkyy kaikille luokan metodeille

    public override void Begin()
    {
        massa = 100.0;
        PhysicsObject pallo = new PhysicsObject(50.0, 50.0, Shape.Circle);
        pallo.Mass = massa;
        TulostaMassa();
    }

    public void TulostaMassa() // HUOM! Ei ole static
    {
        MessageDisplay.Add("Massa on " + massa);
    }
}
```

Älä kuitenkaan innostu liikaa attribuuteista, koska niitä käytetään yleensä aivan liikaa. Parempi on viedä asioita parametrina.

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, https://tim.jyu.fi/view/2
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 University of Jyväskylä

Information Technology