Ohjelmointi 1, C#

Martti Hyvönen, Vesa Lappalainen ja Antti-Jussi Lakanen

Versio: 22.9.2020

22. syyskuuta 2020

Jyväskylän yliopisto

Sisällys

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			

Muista katsoa tarvittaessa myös kurssin videohakemisto.

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 itselleen 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".

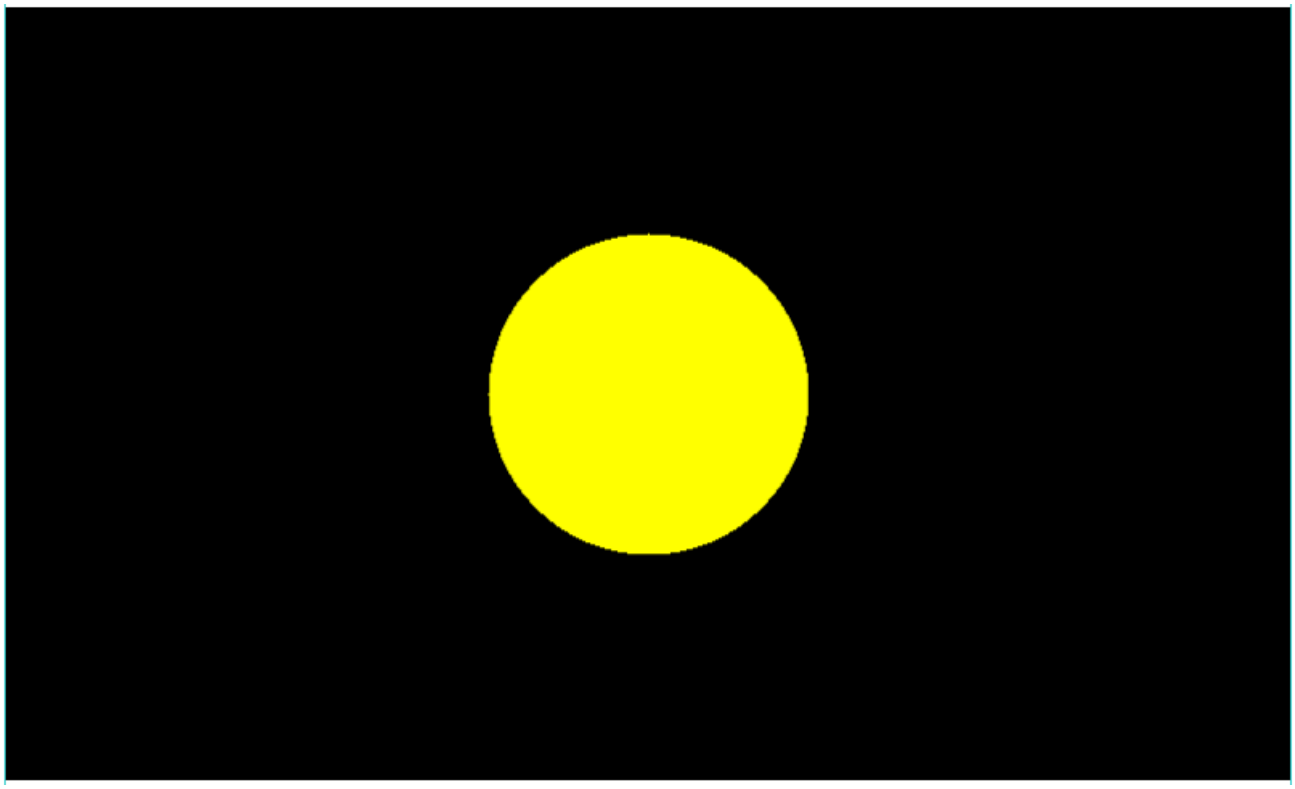
Jos haluat etsiä jotakin, käytä selaimen etsi toimintoa (`Ctrl-F` useimmissa selaimissa).

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.

```
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.

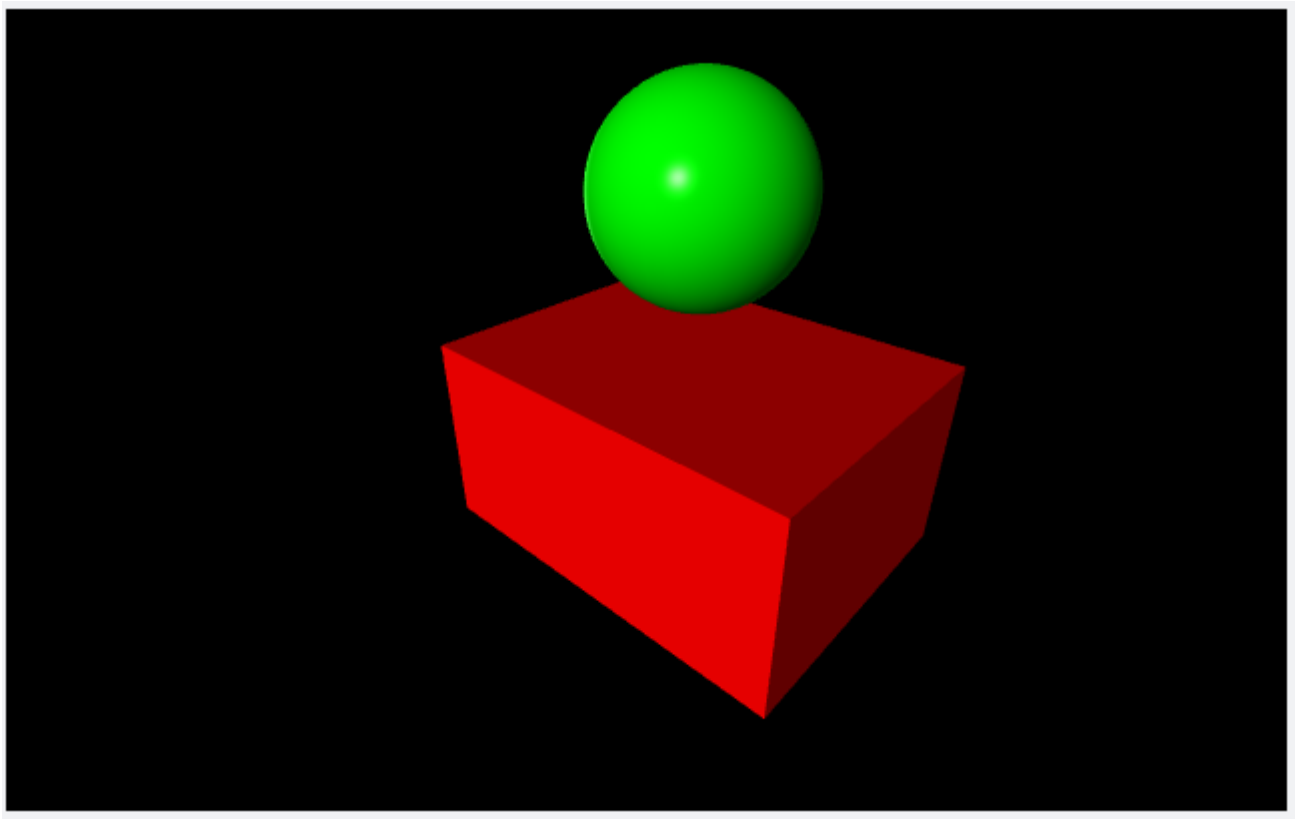
Tehtävä voisi jatkua vielä niin, että: Lisää ennen `Add(pallo)`-riviä rivi

```
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

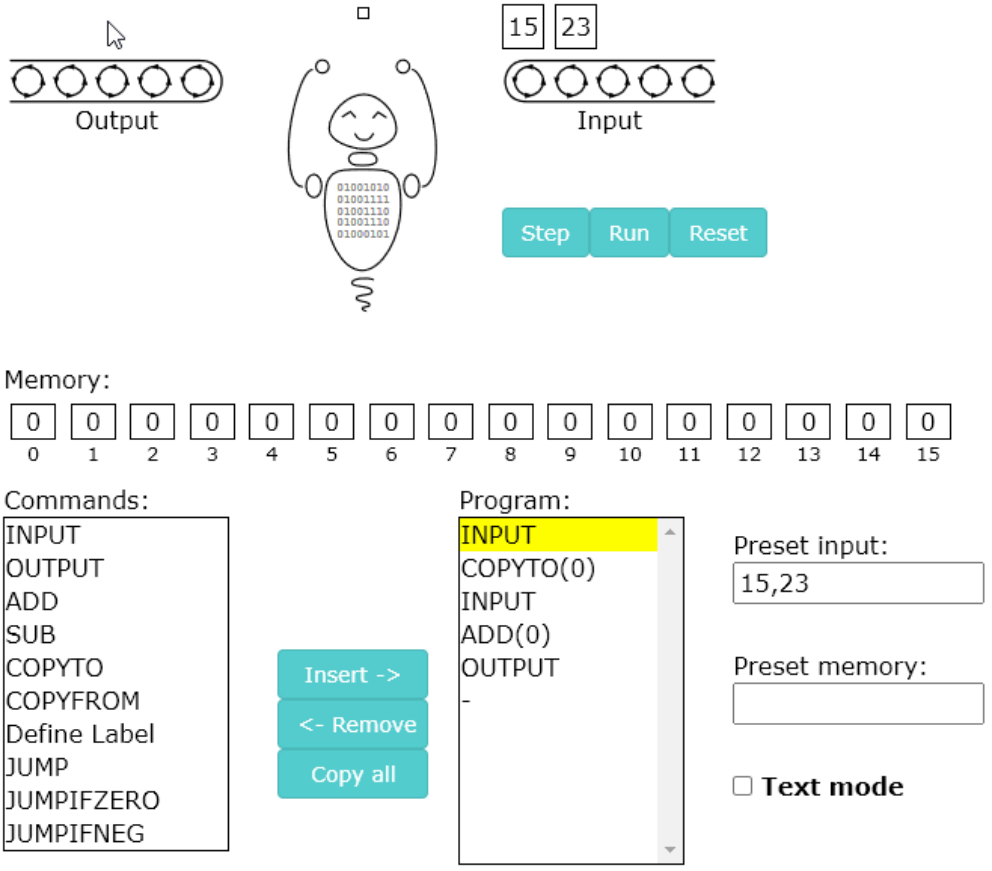
Kielen kääntäjä on ohjelma, joka lukee syötteenään ihmisen kirjoittaman selkokielisen (esim C tai C++ -kieli) ohjelmatiedoston (tekstitiedosto) ja tuottaa siitä binäärimuotoisen suoritettavan (*executable*) konekielisen tiedoston, joka voidaan sitten ajaa. Tämän takia esimerkiksi Windows-järjestelmässä 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:

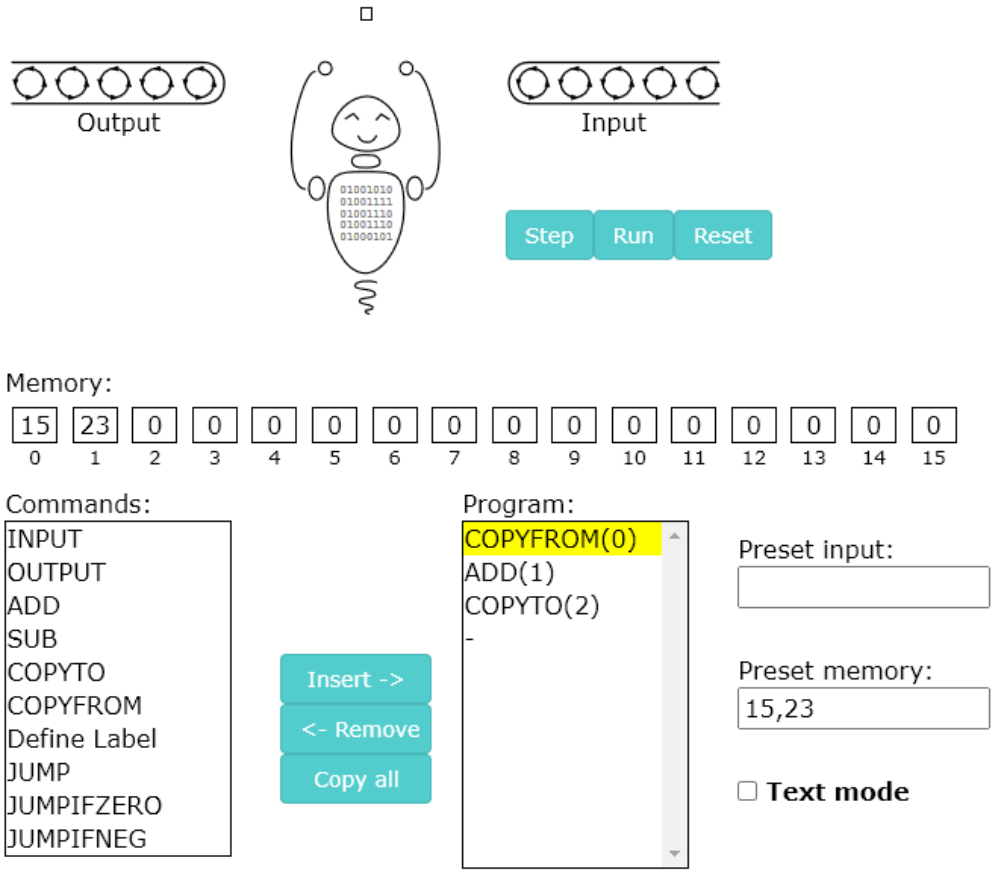
```
00 = INPUT
01 = OUTPUT
02 = ADD
03 = SUB
04 = COPYTO
...
09 = JUMPIFNEG
```

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.

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.

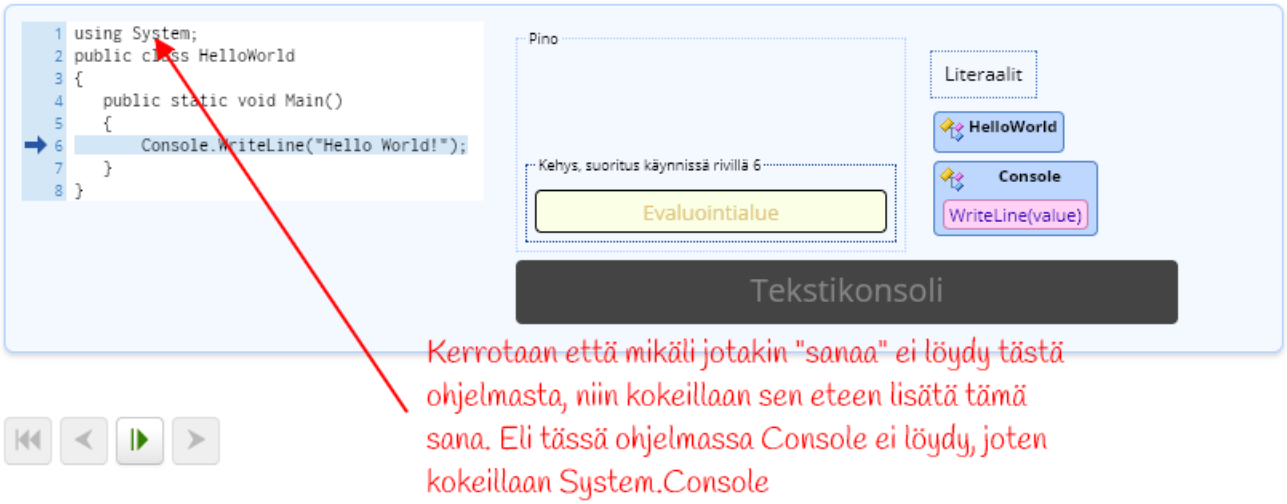
Selvennykseksi vielä video ohjelmakoodin kirjoittamisesta Notepad++:lla Luento 3 (6m40s)

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.

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

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

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

Edellisen esimerkin voisi tehdä myös seuraavasti

```
System.Console.Write("*******      *       *     *\n" +
                     "   *         *       * * * *\n" +
                     "   *         *       *  *  *\n" +
                     "   *         *       *     *\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.

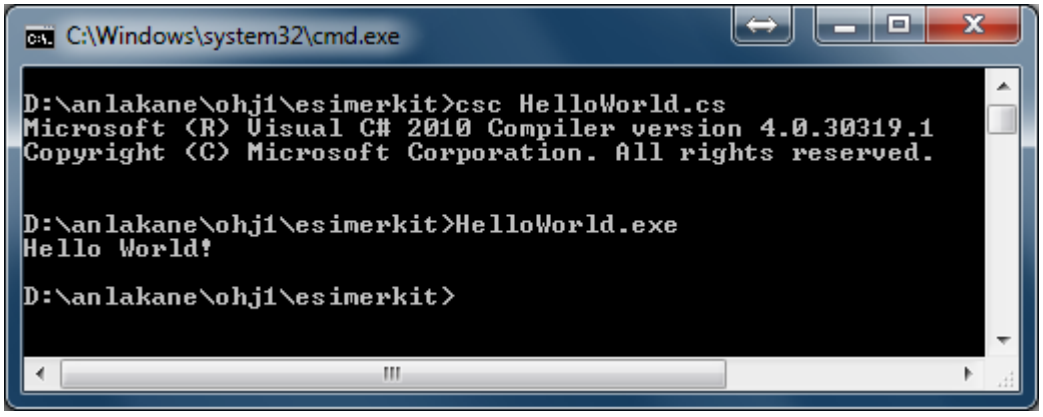
Seuraavaksi opettelemme tekemään nämä vaiheet käsin, jotta ymmärtäisimme paremmin mitä taustalla tapahtuu.

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.

```
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. Kääntäjä ei kuitenkaan tunnista sanoja, joten korvaa sanat C#-kielellä. Kirjoita siis kokonainen ohjelma, joka tulostaa nimesi. Huomaa että ohjelma ei käänny, jos siinä on yksikin tunnistamaton sana.

```
//
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.

```
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.

```
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.

```
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.

```
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

merkintä	tarkoitus
`*/`	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.

```
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 kommenttimerkkiä `/* ... */`.

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. Paina sitten `Document`-linkkiä ja tutki syntynyttä dokumentaatiota.

```
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?

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.

```
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.

```
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.

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 metodeja `Console`-luokalla on kuin `WriteLine()` ? Mitä tekee `Write`?

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)

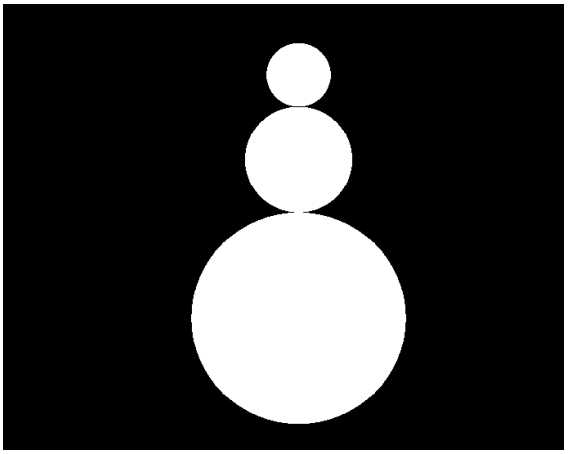
Piirretään lumiukko käyttämällä Jypeli-kirjastoa. Katso sen tekeminen videolta Lumiukko Macillä (7m21s)

```
// 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 pallo

```
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:

```
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:

```
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?

```
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         ball.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.

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

```
1         Level.Background.Color = Color.Black;
2
3         PhysicsObject nelio = new PhysicsObject(200, 200, Shape.Rectangle);
4         nelio.CollisionIgnoreGroup = 1; // Ei haluta että kappaleet törmäävät ↩
      toisiinsa.
5         Add(nelio, 0);
6
7         PhysicsObject pallo = new PhysicsObject(200, 200, Shape.Circle);
8         pallo.Color = Color.Red;
9         pallo.CollisionIgnoreGroup = 1; // Samaan ryhmään kuuluvat eivät törmää
10        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.

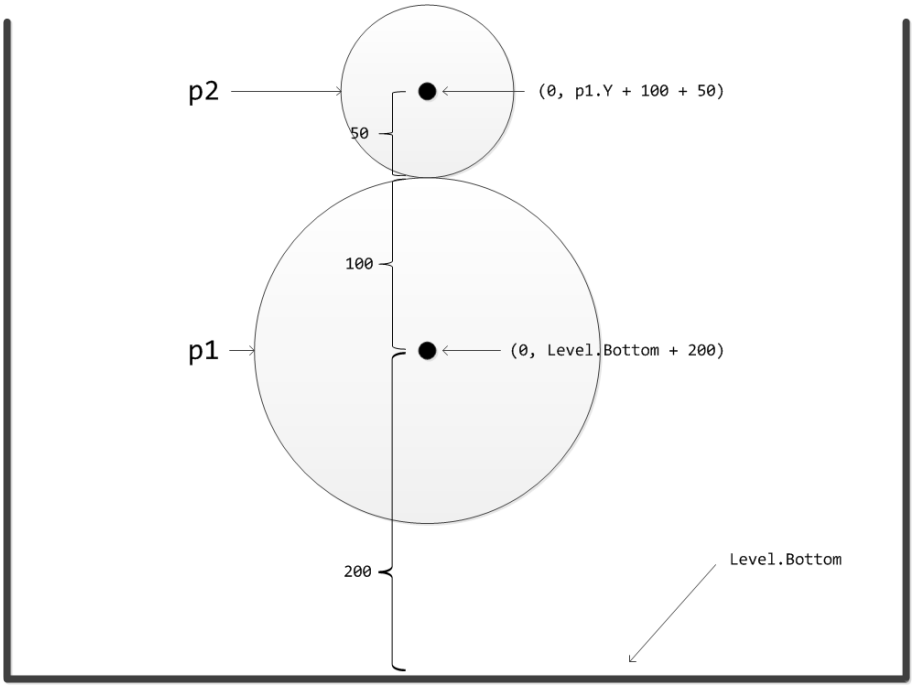
Lumiukko-esimerkissä koordinaattien laskemiseen on käytetty C#:n *aritmeettisia operaatioita*. Voisimme tietenkin laskea koordinaattien pisteet myös itse, mutta miksi tehdä niin, jos tietokone voi laskea pisteet puolestamme? C#:n aritmeettiset perusoperaatiot ovat summa (`+`), vähennys (`-`), kerto (`*`), 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
`pallo.Position=new Vector(jokux,jokuy);`

```
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.

```
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.

```
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?

```
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?

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.

```
using Jypeli;

/// <summary>
/// Piirretään lumiukko.
/// </summary>
public class Lumiukko : PhysicsGame
{
  /// <summary>
  /// Aliohjelma, jossa
  /// piirretään ympyrät.
  /// </summary>
  public override void Begin()
  {
    Camera.ZoomToLevel();
    Level.Background.Color = Color.Black;

    PhysicsObject p1, p2, p3;

    // Eka ukko
    p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
    p1.Y = Level.Bottom + 200;
    Add(p1);

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

    p3 = new PhysicsObject(2 * 30, 2 * 30, Shape.Circle);
    p3.Y = p2.Y + 50 + 30;
    Add(p3);

    // Toinen ukko
    p1 = new PhysicsObject(2 * 100, 2 * 100, Shape.Circle);
    p1.X = 200;
    p1.Y = Level.Bottom + 300;
    Add(p1);

    p2 = new PhysicsObject(2 * 50, 2 * 50, Shape.Circle);
    p2.X = 200;
    p2.Y = p1.Y + 100 + 50;
    Add(p2);

    p3 = new PhysicsObject(2 * 30, 2 * 30, Shape.Circle);
    p3.X = 200;
    p3.Y = p2.Y + 50 + 30;
    Add(p3);
  }
}
```

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

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

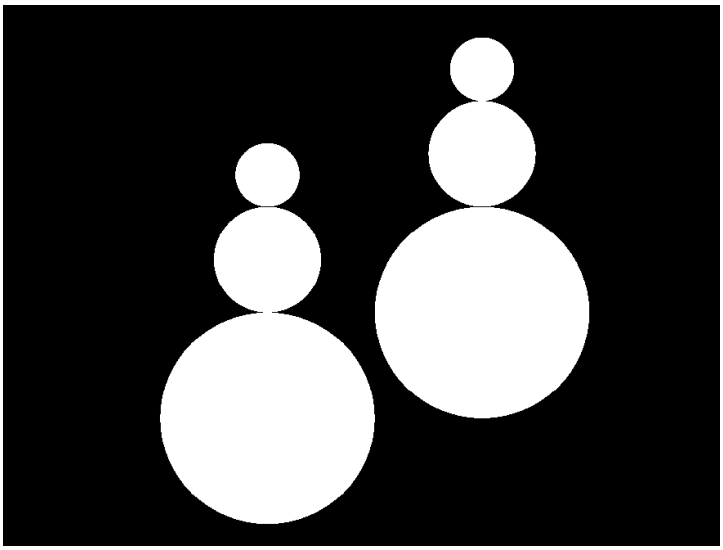
```
x = 0; y = Level.Bottom + 200;
```

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.

```
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.

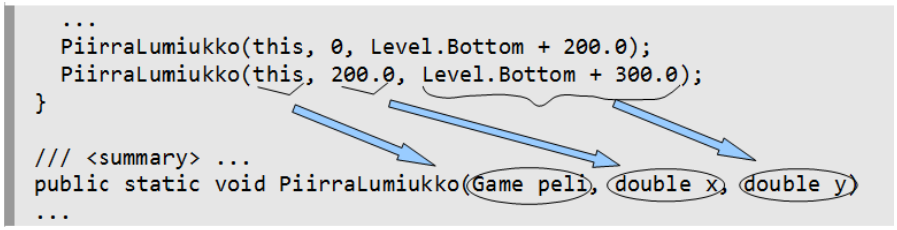
6.2 Aliohjelman kirjoittaminen

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

Aliohjelma määritellään myös staattiseksi (`static`). Staattisen aliohjelman toteutuksessa ei voi käyttää `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 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*. Parametrinä annettiin lumiukon alimman pallon x- ja y-koordinaatit. Molempien tietotyyppi on `double`, joten myös vastinparametrien tyyppien tulee olla double. Annetaan myös nimet kuvaavasti `x` ja `y`.

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 parametrinä 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.

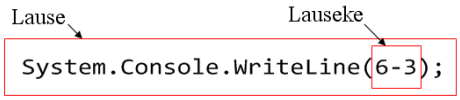
```
PiirraLumiukko(this, 0, Level.Bottom + 200);
```

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 lumiukkoa

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

Parametrit voidaan antaa myös nimettyinä, jolloin niiden järjestystä voidaan muuttaa kutsussa. Lisää ohjelmaan kaksi muuta lumiukkoa. Kokeile, miten voit nimettyjä parametreja käyttää eri tavoilla. Kokeile myös lisätä `Peli.` `PiirraLumiukko`-kutsun eteen. Miksi `Peli.`?

```
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.

```
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:

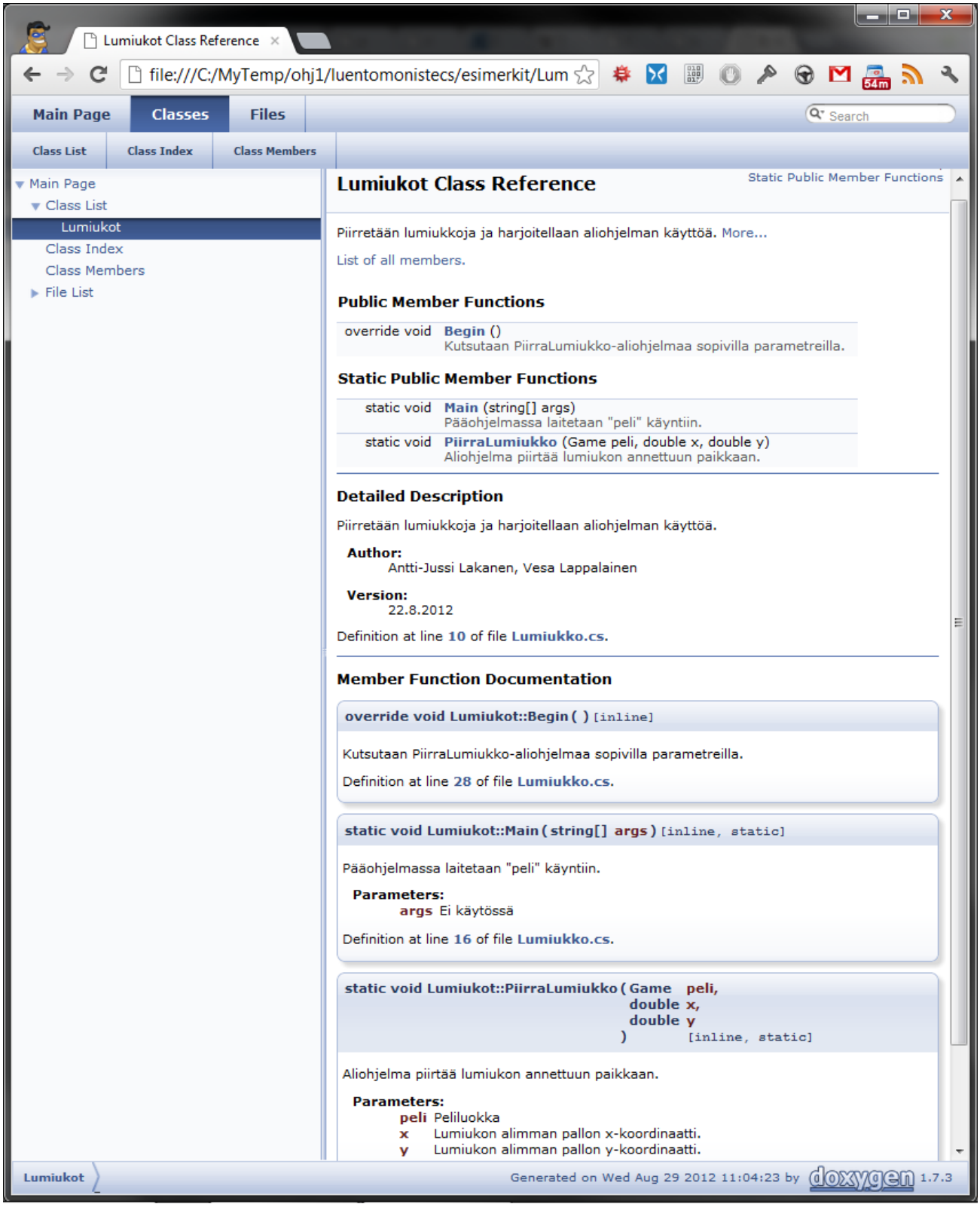
- Dokumentaatiokommentin alkuun laitetaan `summary`-tagien väliin lyhyt ja selkeä kuvaus aliohjelman toiminnasta.
- Jokainen parametri selitetään omien `param`-tagien väliin ja
- paluuarvo `returns`-tagien väliin.

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"

```
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.

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

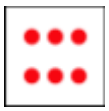
Toki edellisen tehtävän kaltaiset aliohjelmat eivät ole järkeviä, vaan järkevämpää olisi viedä aliohjelmille parametrina että mitä pitää tulostaa.

Tehtävä 6.3

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

```
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 sa-

mannimistä 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 myöhemmin esiteltävää *String Interpolation*, jolla muuttujen arvoja on helppo tulostaa tekstin sekaan.
Tehtävä: Lisää koodiin oikeaoppiset kommentit.

```
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}");
```

```
    }


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


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

Sama käyttäen C#:in oletusparametreja. Oletusparametrin idea on, että jos kutsussa ei ole riittävästi parametreja, kääntäjä lisää kutsuun automaattisesti vastaavan vakion. Tehtävä: Lisää koodiin oikeaoppiset kommentit.

```
public class KuormitusEsimerkki2
{
    public static void Main()
    {
        double summa3 = Summa(5, 10.5, 30.9);
        // kääntäjä tekee seuraavasta kutsun Summa(5, 10.5, 0.0)
        double summa2 = Summa(5, 10.5);
        System.Console.WriteLine($"summa3 = {summa3}, summa2 = {summa2}");
    }


    public static double Summa(double a, double b, double c=0.0)
    {
        return a + b + c;
    }
}
```

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: https://trac.cc.jyu.fi/projects/ohj1/raw-attachment/wiki/kuormittaminen/ukot.png)

Koko esimerkki kuormiteutusta lumiukosta

```
1  using System;
2  using System.Collections.Generic;
3  using Jypeli;
4  using Jypeli.Assets;
5  using Jypeli.Controls;
6  using Jypeli.Effects;
7  using Jypeli.Widgets;
8
9  /// @author Antti-Jussi Lakanen
10 /// @version 30.1.2014
11 ///
12 /// <summary>
13 /// Aliohjelmien kuormittaminen
14 /// </summary>
15 public class Kuormittaminen : PhysicsGame
16 {
17     /// <summary>
18     /// Kutsutaan PiirraLumiukko-aliohjelmaa kahdella eri tavalla.
19     /// Ensimmäisessä ei anneta parametrina kokoa, jolloin tulee "vakiokokoinen" ↩
       lumiukko.
20     /// Toisessa tavassa annetaan parametrina haluttu koko.
21     /// </summary>
22     public override void Begin()
23     {
24         Level.Background.Color = Color.Black;
25         PiirraLumiukko(this, -200, Level.Bottom + 300.0);
26         PiirraLumiukko(this, 0, Level.Bottom + 200.0, 50.0);
27
28         PhoneBackButton.Listen(ConfirmExit, "Lopeta peli");
29         Keyboard.Listen(Key.Escape, ButtonState.Pressed, ConfirmExit, "Lopeta ↩
       peli");
```

```
30      }
31
32      /// <summary>
33      /// Aliohjelma piirtää vakiokokoisen lumiukon
34      /// annettuun paikkaan.
35      /// </summary>
36      /// <param name="peli">Peli, johon lumiukko tehdään.</param>
37      /// <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.

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;
```

```
1 //
2         bool onkoKalastaja = true;
3         char merkki = 't';
4         int kalojenLkm = 0;
5         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:

```
1 //
2         int ika = 2.5; // TÄMÄ KOODI EI KÄÄNNY
```

mutta reaalilukuun voi sijoittaa kokonaisluvun

```
1 //
2         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.

```
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.

Tehtävä 7.2

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

```
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.

```
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. Muuttujaan voi asettaa vain määrittelyssä annetun tietotyypin mukaisia arvoja tai sen kanssa *sijoitusyhteensopivia* arvoja. Esimerkiksi liukulukutyyppeihin (float ja double) voi sijoittaa myös kokonaislukutyyppisiä arvoja, sillä kokonaisluvut 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.

```
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 3

```
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?

7.3.3.1 Tehtävä 7.5 i:n kasvatus, mitä ohjelma tulostaa

Älä vielä aja ohjelmaa, vastaa ensin alla olevaan kysymykseen.

```
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.

```
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)

```
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.

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

Kirjoita mitä tulostaa milläkin operaattorilla

```
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?

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.

```
//
        double tulos;
        int jakaja = 3;
        int jaettava = 7;
        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.

```
        byte pieninByte  =  byte.MinValue;
        byte suurinByte  =  byte.MaxValue;
        int pieninInt  =  int.MinValue;
        int suurinInt  =  int.MaxValue;
        long pieninLong  =  long.MinValue;
        long suurinLong  =  long.MaxValue;
        float pieninFloat  =  float.MinValue;
        float suurinFloat  =  float.MaxValue;
        float nollaaLahinFloat = float.Epsilon;
        double pieninDouble =  double.MinValue;
        double suurinDouble  =  double.MaxValue;
        double nollaaLahinDouble = double.Epsilon;
        Console.WriteLine($"Pienin byte on {pieninByte}, suurin on {suurinByte}");
    ;
        Console.WriteLine($"Pienin int on {pieninInt}, suurin on {suurinInt}");
        Console.WriteLine($"Pienin long on {pieninLong}, suurin on {suurinLong}");
    ;
        Console.WriteLine($"Pienin float on {pieninFloat}, suurin on {suurinFloat
    }");
        Console.WriteLine($"Nollaa lähin {nollaaLahinFloat}");
        Console.WriteLine($"Pienin double on {pieninDouble}, suurin on {
    suurinDouble}");
        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.

```
1  /// @author  Antti-Jussi Lakanen
2  /// @version 22.8.2012
3  ///
4  /// <summary>
5  /// Ohjelma, joka laskee painoindeksin
6  /// pituuden (m) ja painon (kg) perusteella.
7  /// </summary>
8  public class Painoindeksi
9  {
10     /// <summary>
11     /// Pääohjelma, jossa painoindeksi tulostetaan ruudulle.
12     /// </summary>
13     public static void Main()
14     {
15       double pituus = 1.83;
16       double paino = 75.0;
17       double painoindeksi = paino / (pituus*pituus);
18       System.Console.WriteLine("Painoindeksisi on {0:0.00}",painoindeksi);
19    }
20 }
```

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.

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.

Tehtävä 7.11

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

```
1 using System;
2
3 public class Tulostus
4 {
5     public static void Main()
6     {
7         String teksti = "Jeps Jeps";
```

```
8           TulostaTeksti();
9
10      }
11
12
13      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.

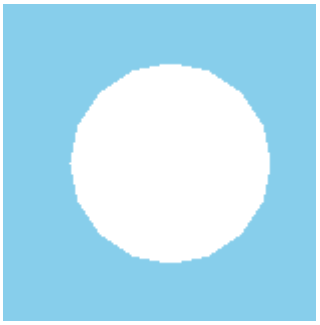
Samaan olioon voi viitata useampi muuttuja. Vertaa alla olevia koodinpätkiä.

```
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:

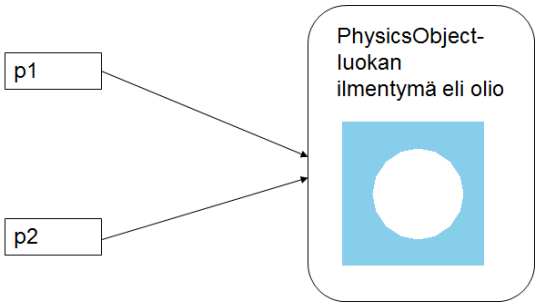


Kuva 8: Molemmat muuttujat, p1 ja p2, liikuttelevat samaa ympyrää. Lopputuloksena ympyrä seisoo pisteessä x=100.

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.

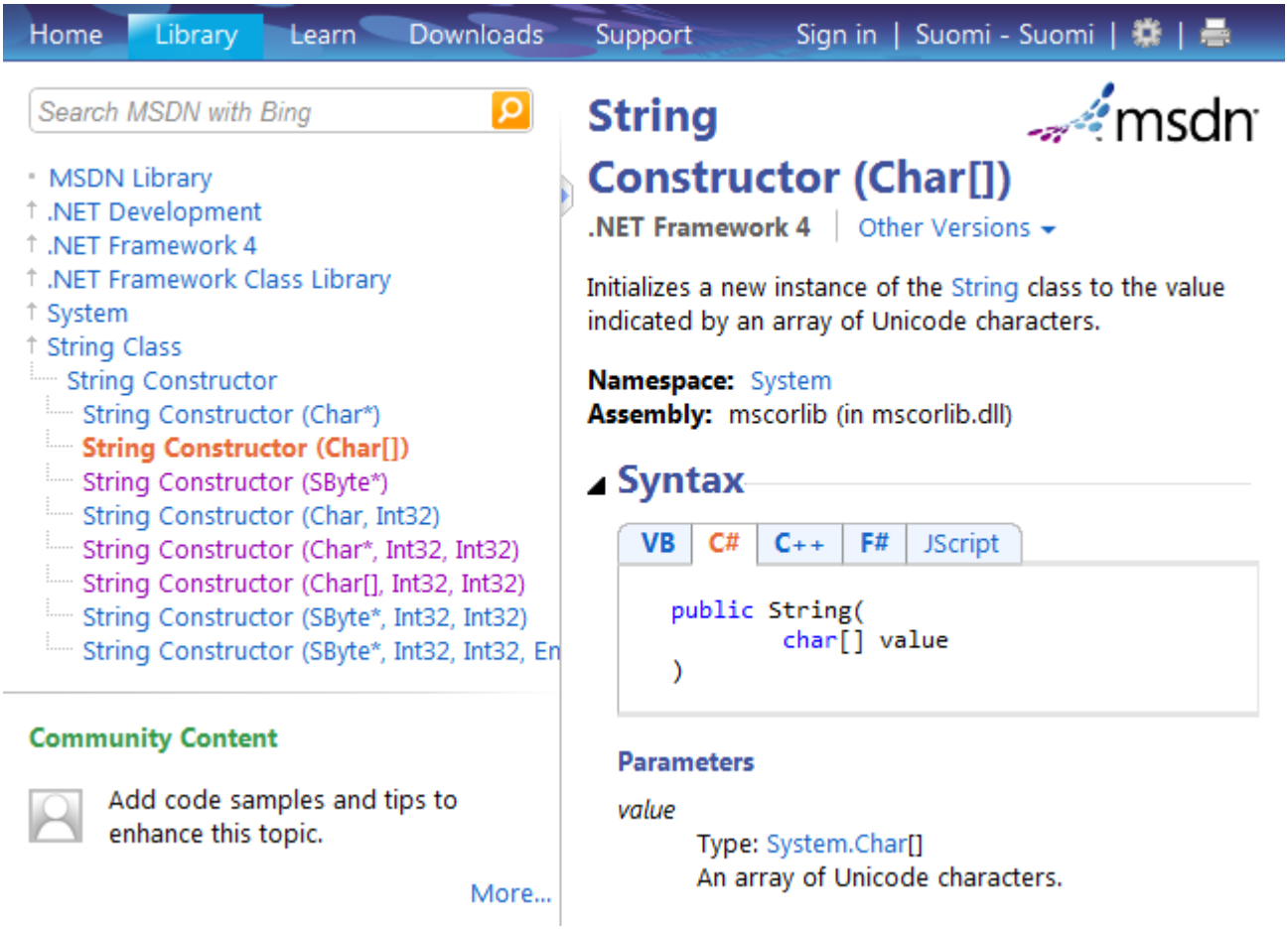
Kunkin konstruktorin kohdalla on lyhyesti kerrottu mitä se tekee, ja sen jälkeen minkä tyyppisiä ja montako parametria konstruktori ottaa vastaan. Kaikista konstruktoreista saa lisätietoa klikkaamalla konstruktorin esittelyriviä. Esimerkiksi linkki

[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?

```
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.

```
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)

```
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));
```

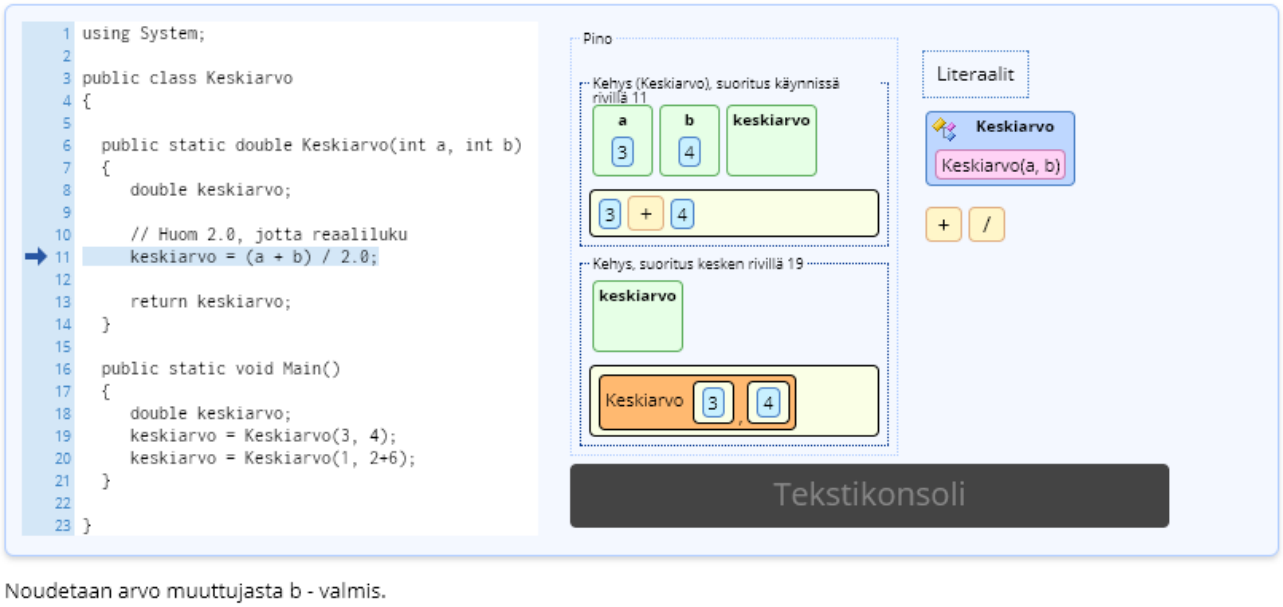
Koska Keskiarvo-aliohjelma palauttaa aina double-tyyppisen liukuluvun, voidaan kutsua käyttää kuten mitä tahansa double-tyyppistä arvoa. Se voidaan esimerkiksi tulostaa tai tallentaa muuttujaan.

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
...
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

```
public class Painoluokka
{
    public static void Main()
    {
      double pituus = 1.83;
      double paino = 75.0;
      double painoindeksi = paino / (pituus*pituus);
      System.Console.WriteLine(painoindeksi);
   }
}
```

Mallivastaus

Mallivastaus

```
public class Painoluokka
{
    public static void Main()
    {
      double pituus = 1.83;
      double paino = 75.0;
      double painoindeksi = Painoindeksi(pituus, paino);
      System.Console.WriteLine(painoindeksi);
   }


   public static double Painoindeksi(double pituus, double paino)
   {
       double tulos = paino / (pituus*pituus);
       return tulos;
   }
}
```

Tehtävä: Aliohjelmat

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

```
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);
   }
}
```

- 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.

```
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:

```
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
{
```

```
8       /// <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.

- 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.

```
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 näyttää

```
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 valmiina

```
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.

```
    /// <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.

```

    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.

Tehtävä 11.4

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

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

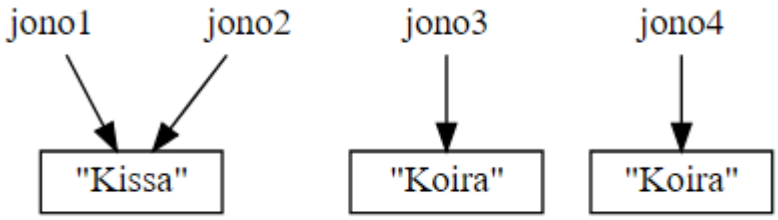
```
jono1 += " istuu";
```

on sama kuin

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

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

Katso animaatiota liikkumalla nuolilla »



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 nuolilla

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 nuolilla

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.

```
        string eiViitetta = null;
        string tyhja = "";
        string valilyonti = " ";

        int tyhjanPituus = tyhja.Length;              // 0
        int valilyonninPituus = valilyonti.Length;    // 1
        char vali = valilyonti[0];                    // ' '
        // char eka = tyhja[0];       // kaatuisi Array index is out of range.
        // 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.

```
        string eiViitetta = null;
        string tyhja = "";
        string nimi = "Matti";

        if ( nimi.Length > 0 ) Console.WriteLine("Nimi ok");   // tulostuu
        if ( tyhja.Length > 0 ) Console.WriteLine("Tyhja ok"); // ei tulostu
        if ( tyhja != null ) Console.WriteLine("Tyhja ei null"); // tulostuu
        if ( eiViitetta == null ) Console.WriteLine("on null"); // tulostuu
        if ( String.IsNullOrEmpty(eiViitetta) ) Console.WriteLine("on null");
        if ( !String.IsNullOrEmpty(tyhja) ) Console.WriteLine("ei tulostu");
        if ( !String.IsNullOrEmpty(nimi) ) Console.WriteLine("Nimi ok");
        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*).

```
//
        string k = "Kissa";
        string kissaPienena = k.ToLower(); // syntyy uusi jono
        Console.WriteLine(kissaPienena); // tulostaa "kissa"
```

Katso animaatiota liikkumalla nuolilla

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 nuolilla

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 nuolilla

`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 nuolilla

- 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 nuolilla

```
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 nuolilla

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?

```
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.

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 nuolilla

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 nuolilla

Huomaa, että `Insert`-metodi ei korvaa indeksissä 7 olevaa merkkiä, vaan lisää merkkijonoon kokonaan uuden merkin/merkkejä, jolloin merkkijonon pituus kasvaa siis lisättävän jonon pi-

tuudella. 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 nuolilla

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 nuolilla

- 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 nuolilla

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      }
```