Jirka Ylönen

C#-kielen perusteet

Tietotekniikan LuK-tutkielma
3.3.2002

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Jirka Ylönen

Yhteystiedot: Sähköposti jjylonen@st.jyu.fi

Työn nimi: C#-kielen perusteet

Title in English: Fundamentals of C# language

Työ: Tietotekniikan LuK-tutkielma

Sivumäärä: 27

Linja: Ohjelmistotekniikka

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Tiivistelmä: Tämä tutkielma käsittelee Microsoftin vuonna 2000 julkaisemaa uutta oliopohjaista ohjelmointikieltä, C#:ia. Esitietoina tutkielman lukijalta oletetaan olio-ohjelmoinnin sekä C++-kielen perusosaaminen. Tutkielmassa on painotettu niitä asioita, joita C++:sta ei löydy, tai jotka ratkaisevasti poikkeavat C++:n vastaavista.

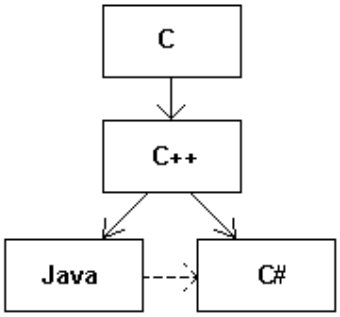
Avainsanat: C#, olio-ohjelmointi, .NET-arkkitehtuuri

Keywords: C#, object-oriented programming, .NET architecture

Sisällysluettelo

1. Johdanto

C# (äännetään C sharp) on Microsoftin C++:n pohjalta kehittämä, vuonna 2000 julkistama uusi oliopohjainen ohjelmointikieli. Nimensä se on saanut vastaavasta musiikkinuotista C# eli ylennetty C. Kuten myöhemmin tullaan näkemään, C# on saanut paljon vaikutteita myös Javasta. Tämän tutkielman lukijan ei kuitenkaan oleteta tuntevan Javaa, vaan ainoastaan C++:n perusteet. Ne ominaisuudet, joita C++:sta ei löydy, on käsitelty uusina, vaikka ne Javasta löytyisivätkin.



Kuva 1. C#:n suhde muihin ohjelmointikieliin

Aloitetaan kielen esittely totuttuun tyyliin tulostamalla kuvaruudulle teksti "Hello World":

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

Ohjelman suoritus alkaa pääohjelmasta Main, joka C++:sta poiketen aloitetaan isolla alkukirjaimella. Tärkeämpi ero kuitenkin on se, että pääohjelma on kirjoitettu luokan HelloWorld metodiksi. C# onkin Javan tapaan puhdas oliokieli, mikä tarkoittaa sitä, että kaikki funktiot ovat luokkien metodeita. C#:lla ei siis voida kirjoittaa muita kuin olio-ohjelmia. Pääohjelman sisältävää luokkaa kutsutaan sovelluksen pääluokaksi.

Totuttuun tapaan tyhjät sulkeet pääohjelman nimen perässä ilmaisevat, että pääohjelma ei saa yhtään parametriä, ja sana void, ettei sillä ole paluuarvoa. Public-määreellä ilmaistaan, että metodi näkyy sovelluksen kaikille luokille. Static puolestaan tarkoittaa, että kyseessä on ns. luokkametodi, jota voidaan kutsua ilman, että luokasta on olemassa ilmentymää eli oliota.

Näytölle tulostus tapahtuu kutsumalla C#:n standardikirjastosta löytyvän Console-luokan metodia WriteLine, joka ottaa parametrikseen merkkijonon. Sana System ilmaisee, että Console-luokka kuuluu System-nimiseen nimiavaruuteen. Käyttäjän omat luokat kuuluvat oletusarvoisesti nimettömään nimiavaruuteen.

Tämän tutkielman luvussa 2 puhutaan ensiksi vähän C#-ohjelmointiympäristöstä. Luvussa 3 käydään läpi C#:n peruskielioppia ja luvussa 4 puolestaan paneudutaan C#:n olio-ominaisuuksiin.

2. C#-ohjelmointiympäristö

Tässä luvussa käydään läpi C#:n käännösprosessi, joka poikkeaa perinteisissä ohjelmointikielissä käytetystä. Lisäksi puhutaan hieman C#:n standardikirjastosta ja Microsoftin uudesta .NET-arkkitehtuurista.

Ainoa kaupallinen C#-kehitin tätä kirjoitettaessa lienee Visual Studio.NET -pakettiin kuuluva Visual C#. Sen käyttöön ei kuitenkaan puututa. Komentorivipohjaiset ohjelmointityökalut ja C#-ohjelmien ajamiseen vaadittava .NET-ympäristö on ladattavissa ilmaiseksi Microsoftin verkkosivuilta.

2.1. Kääntäminen

C#-ohjelma kirjoitetaan yhteen tai useampaan lähdekooditiedostoon, joiden tarkentimeksi on varattu .cs. Yleinen käytäntö on kirjoittaa kukin luokka omaan tiedostoonsa. Mainittakoon vielä, että C#:ssa ei käytetä erillisiä header-tiedostoja.

C#-lähdekooditiedostot voitaisiin tietenkin kääntää normaalisti kääntäjällä objektitiedostoiksi ja sitten linkittää objektitiedostot suoritettavaksi ohjelmaksi. C#:a varten on kuitenkin kehitetty oma välikieli MSIL (Microsoft Intermediate Language), jolle kääntäminen normaalisti tehdään. Ohjelma ajetaan sitten erillisellä välikielitulkilla.

Sama menettely on käytössä Javassa ja sen etuna on, että sama koodi toimii eri laitteistoissa ja käyttöjärjestelmissä, joihin vain on tehty välikielitulkki.
Suoritus on tietysti varsinaiseen konekieliseen ohjelmaan nähden hitaampaa, mutta kuitenkin huomattavasti nopeampaa kuin suoraan lähdekoodista tulkkaaminen. Jälkimmäistä tapaahan käytetään usein esimerkiksi perinteisellä Basicillä ohjelmoitaessa.

Kokonaan toinen menettely on vielä käyttää ns. JIT (Just-In-Time) -kääntämistä. Siinä välikielinen ohjelma käännetään juuri ennen suoritusta konekielelle, jolloin siitä tulee nopeampi. Miinuksena on tietenkin kääntämiseen kuluva aika.

Esimerkki käyttäen Windowsin komentorivipohjaista csc-kääntäjää:

`csc ohjelma.cs`	Ohjelma yhdessä tiedostossa
`csc ohjelma.cs apu.cs apu2.cs`	Ohjelma useammassa tiedostossa

Tuotoksena saadaan tiedosto ohjelma.exe (olettaen, että sovelluksen pääluokka on tiedostossa ohjelma.cs myös jälkimmäisessä tapauksessa). Tiedosto näyttää tavalliselta ohjelmalta ja käynnistetäänkin samalla tavalla, mutta todellisuudessa se sisältää MSIL-välikieltä. Ohjelma ei toimi koneessa, johon ei ole asennettuna .NET-ympäristöä. Se, tulkataanko välikielistä ohjelmaa, vaiko käännetäänkö se heti JIT-kääntäjällä konekieliseksi, riippuu .NET-ympäristön asetuksista.

2.2. Standardikirjasto ja .NET-arkkitehtuuri

.NET on Microsoftin Windowsia varten kehittämä uusi ohjelmistoarkkitehtuuri, joka vuoden 2001 lopulla julkaistussa Windows XP:ssä on integroitu kiinteäksi osaksi käyttöjärjestelmää.

Tässä yhteydessä .NET:iä on yksinkertaisinta ajatella standardiluokkakirjastona, jota C#-ohjelmat käyttävät, ja joka peittää alleen Windowsin melko monimutkaisen perus-API:n. Todellisuudessa .NET on kuitenkin paljon laajempi käsite.

Kuva 2. C#-kielisen ohjelman kutsupino

.NET:iä ei ole suunniteltu pelkästään C#:a varten, vaan sitä voidaan käyttää myös esimerkiksi C++:sta. Samoin Microsoftin uusi Visual Basic.NET käyttää kirjastonaan .NET:iä. Tämän tekee yksinkertaiseksi yhteinen, jo edellisessä luvussa mainittu MSIL-välikieli, jolle eri lähdekieliset ohjelmat ensiksi käännetään.

Mainittakoon vielä, että .NET:iä käytetään yleiseen Windows-tyyliin dynaamisista kirjastoista (dll). Välikielelle käännettävään C#-ohjelmaan ei siis linkitetä kirjastojen koodia, vaan ainoastaan viittaukset dll-tiedostoihin.

3. Kielen perusteita

C#-kielen perusteet ovat varsin samanlaiset kuin C++:ssa. Tämän vuoksi useimmista asioista käydään läpi vain syntaksi. Laajemmin tarkastellaan C#:n uusia ominaisuuksia sekä sellaisia ominaisuuksia, jotka kylläkin löytyvät C++:sta, mutta eivät kuulu aivan ohjelmoinnin alkeisiin.

3.1. Yleistä

C#:ssa pätevät samat kirjoitussäännöt kuin C++:ssa. Isoilla ja pienillä kirjaimilla on eroa. Muuttujien nimet suositellaan kirjoitettavaksi kokonaan pienellä, metodien nimet ensimmäistä sanaa lukuunottamatta isolla ja luokkien nimet kokonaan isolla alkukirjaimella:

```
omamuuttuja
omaMetodi
OmaLuokka
```

Nimet saavat sisältää aakkosten lisäksi numeroita sekä alaviivoja. Numeroa ei kuitenkaan saa käyttää ensimmäisenä merkkinä.

C#:n varatut sanat, joita ei saa käyttää muussa tarkoituksessa:

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

Kielen lauseet lopetetaan totuttuun tyyliin puolipisteeseen. Useampia lauseita sisältävät lohkot merkitään aaltosuluilla:

```
lause;

{
  lause1; lause2; lause3;
}
```

C#-kielinen ohjelma kommentoidaan samalla tavalla kuin C++:ssakin:

```
// Yhden rivin kommentti

/* Useamman
   rivin
   kommentti */
```

Vielä on käytössä kolmas kommenttityyppi, nimittäin XML-kommentti. Niihin ei tässä tutkielmassa puututa. Mainittakoon kuitenkin XML-kommentin synktaksi:

```
/// <tagi> tekstiä </tagi>
```

C#:ssa on käytössä seuraavat tutut operaattorit. Operaattoreilla on tietty laskujärjestys, jota voi normaaliin tapaan muuttaa kaarisulkeilla.

yhteen-, vähennys-, kerto- ja jakolasku sekä jakojäännös +, -, *, / ja %
vertailuoperaattorit <, >, <=, >=, == ja !=
loogiset operaattorit &&, || ja !
bittitason operaattorit &, |, ~, << ja >>

Muut operaattorit on luontevinta käsitellä niihin liittyvien kielen piirteiden yhteydessä.

3.2. Syöttö ja tulostus

Tässä tutkielmassa ei tarkastella laajemmin C#:n standardikirjastoa. Syöttö ja tulostus ovat kuitenkin niin keskeisiä operaatioita, että niitä ei voi jättää mainitsematta.

Tulostus ilman rivinvaihtoa:

```
System.Console.Write(lauseke);
```

Tulostus rivinvaihdon kanssa:

```
System.Console.WriteLine(lauseke);
```

Syöttö:

```
muuttuja = Console.ReadLine();
```

Edellä mainitut metodit on tarkoitettu syöttöön ja tulostukseen konsoliohjelmissa, graafisia ohjelmia varten on kirjastossa omat luokkansa.

3.3. Tavalliset muuttujat

Muuttujien käsittely ei C#:ssa poikkea C++:sta.

Muuttujan määrittely: `int a;`

Useampia samantyyppisiä muuttujia voidaan määritellä samalla rivillä: `int a, b, c;`

Kuten C++:ssakin, muuttujan arvo on määrittelemätön ennen ensimmäistä sijoitusta siihen. C++:sta poiketen yritys käyttää tällaista muuttujaa aiheuttaa kuitenkin virheilmoituksen.

Sijoituslause: `a = 1;`

Muuttujat voidaan alustaa (sijoittaa niille arvo määrittelyn yhteydessä): `int a = 1;`

Lyhennysmerkinnät: `a += 5;` <==> `a = a + 5;`

Muuttujan lisäys tai vähennys yhdellä:

```
a++; <==> a = a + 1;
a--; <==> a = a - 1;

++a; <==> a = a + 1;
--a; <==> a = a - 1;
```

Yksittäisenä lauseena esi- ja jälkilisäysmuodot toimivat samoin, mutta eroavat toisistaan, jos muuttujan arvoa samassa lauseessa käytetään johonkin:

```
a = 1;
b = --a;
c = b++;
```

a:n arvoksi tulee 0, samoin b:n
b:n arvoksi tulee 1, c:n arvoksi 0

Muuttujille on käytettävissä seuraavat tietotyypit:

Etumerkilliset kokonaisluvut:

`sbyte`	1 tavu
`short`	2 tavua

	int	4 tavua
	long	8 tavua
Etumerkittömät kokonaisluvut:	**byte**	1 tavu
	ushort	2 tavua
	uint	4 tavua
	ulong	8 tavua
Reaaliluvut:	**float**	4 tavua
	double	8 tavua
	decimal	12 tavua
Totuusarvo:	**bool**	
Merkkityyppi:	**char**	2 tavua

Huomattakoon ensiksi, että kaikkien tietotyyppien varaama tila on vakio. C++:ssahan int-tyyppi saattoi kääntäjän toteutuksesta riippuen viedä 2 tai 4 tavua tilaa. Merkkityyppi char (käytännössä sama kuin short) vie C++:sta poiketen 2 tavua. Tämä siksi, että voitaisiin käyttää UniCode-merkistöä. UniCodea voidaan lisäksi käyttää myös itse lähdekoodissa. Totuusarvotyypin bool arvoiksi löytyvät kielestä varatut sanat true ja false.

Merkkijonoille C#:ssa on näennäinen tyyppi string. Todellisuudessa käytetään kuitenkin standardikirjastosta löytyvää String-luokkaa, jonka olemassaolosta kääntäjä siis on tietoinen. Merkkijonojen käsittely tapahtuu kuten muidenkin muuttujien, merkkijonon sijoitukseen toiseen ei tarvita erillisiä funktioita, kuten C-kielessä:

```
string a, b, c = "jono";
b = "jono2";
a = b;
```

Merkkijonojen yhdistäminen onnistuu +-merkillä:

```
string a = "jono", b = "jono2", c;
c = a + b;
```

c saa arvon "jonojono2"

3.4. Osoittimet

Osoittimia ei C#:ssa suositella käytettäväksi. Jos niitä kaikesta huolimatta tarvitaan, on osoittimia sisältävään metodiin kirjoitettava sana unsafe:

```
public unsafe void OsoittimiaKayttavaMetodi( ) {
  ...
}
```

Osoittimien syntaksi on sama kuin C++:ssa.

Osoitinmuuttujan esittely:	`int *a;`
Viittaaminen osoittimen osoittaman muistipaikan sisältöön:	`int b = *a;`
Tavallisen muuttujan muistiosoitteeseen viittaaminen:	`int *c = &b;`

3.5. Taulukot

Taulukko on C#-kielessä itse asiassa olio. Koska oliot ovat viittauksia, kuten luvussa 4 nähdään, vaaditaan taulukon käyttöönottamiseksi taulukkomuuttujan esittely sekä tilanvaraus taulukolle.

Esimerkki: Varataan tila 100-alkioiselle kokonaislukutaulukolle:

```
int [] i;
i = new int[100];
```

tai suoraan:

```
int [] i = new int[100];
```

Taulukon voi myös alustaa tilanvarauksen yhteydessä, tällöin kokoa ei välttämättä tarvitse erikseen mainita:

```
int [] i = new int[] {1, 2, 3};
```

tai lyhyemmin

```
int [] i = {1, 2, 3};
```

Taulukon indeksointi alkaa nollasta, kuten C++:ssakin ja arvon sijoitus taulukkoon menee totuttuun tapaan:

```
i[0] = 1;
```

Vaikka taulukko onkin dynaamisesti varattu, sitä ei tarvitse erikseen vapauttaa, koska C#:ssa on käytössä ns. automaattinen roskienkeruu, joka vapauttaa automaattisesti kaikki ne muistialueet, joihin ei enää ole viittauksia.

Moniulotteisia taulukoita voidaan käsitellä seuraavanlaisella syntaksilla:

```
int[,] i = new int[2,3];

i[1,1] = 2;
```

3.6. Numeroitu tyyppi

C#:sta löytyy C++:n tapaan numeroitu tyyppi enum. Vaikka ominaisuus onkin hieman harvemmin käytetty, käydään sekin kuitenkin läpi.

Numeroitu tyyppi esitellään luokan jäseneksi seuraavasti:

```
public enum VIIKONPAIVAT {
  Maanantai,
  Tiistai,
  Keskiviikko
  Torstai,
  Perjantai,
  Lauantai,
  Sunnuntai
}
```

Esimerkissä Maanantai vastaa kokonaislukua 0 ja sunnuntai kokonaislukua 6. Luvut voidaan kuitenkin itse valita seuraavalla syntaksilla:

```
...
Maanantai = 1,
...
```

Jollei muuta määrätä, seuraava tekstivakio saa kaikissa tapauksissa yhtä suuremman arvon kuin edellinen. Edellisessä esimerkissä siis Tiistai saisi arvon 2 ja Sunnuntai arvon 7.

Numeroitua tyyppiä oleva muuttuja esitellään ja alustetaan nyt seuraavasti:

```
VIIKONPAIVAT paiva = Maanantai;
```

3.7. Ehto- ja valintarakenteet

If-else -rakenteessa ei ole eroa C++:aan nähden:

```
if (ehto) {
  ...
}
else {
   ...
}
```

Jos ehto toteutuu, suoritetaan if-osan, muutoin else-osan jäljessä olevat lauseet. Else-osa voi myös puuttua, jolloin ehdon ollessa epätosi ei tehdä mitään.

Switch-case -valintarakenteeseen on tullut muutamia muutoksia:

```
switch (muuttuja) {
  case arvo1:
  case arvo2:
    ...
    break;
  case arvo3:
    ...
    goto default;
  case arvo4:
    ...
    goto case2;
  case default;
    ...
}
```

Tuttuun tapaan suoritetaan sen case-osan jälkeiset lauseet, johon muuttujan arvo täsmää, ja default-osan jälkeiset lauseet, jos arvo ei täsmää mihinkään case-osaan. Jollei default-osaa ole, ei jälkimmäisessä tapauksessa tehdä mitään.

C++:ssa piti case-osan loppuun kirjoittaa sana break, jollei haluttu suorittaa myös seuraavaa case-osaa. Usein tämä jäi pois vahingossa ja aiheutti ajonaikaisia virheitä. Siksipä C#:ssa break onkin pakollinen muissa paitsi viimeisessä case- tai default-osassa. Break voi jäädä pois myös, jos case-osassa ei ole mitään koodia, jolloin siirrytään C++:n tapaan seuraavaan case-osaan.

Break-sanan asemesta voidaan käyttää myös goto case - ja goto default -lauseita, joista ensimmäinen jatkaa suoritusta halutusta case-osasta ja jälkimmäinen default-osasta.

3.8. Silmukkarakenteet

C#:ssa on samat silmukkarakenteet kuin C++:ssakin.

While-silmukkaa suoritetaan niin kauan kuin ehto on tosi, ei välttämättä kertaakaan tai sitten ikuisesti:

```
while (ehto) {
  ...
}
```

Do-while -rakenteessa silmukan sisältö suoritetaan ainakin kerran:

```
do {
  ...
} while (ehto);
```

For-silmukkaa suoritetaan niin kauan kuin ehto on tosi. Alustuslause suoritetaan ensimmäisellä kierroksella ja sen jälkeen päivityslause seuraavilla kierroksilla. Normaalissa käytössä ehto tulee päivityslauseen ansiosta lopulta epätodeksi:

```
for (alustuslause; ehto; päivityslause) {
  ...
}
```

Uutena C#:ssa on foreach-silmukkarakenne. Sen toiminta on helpointa selvittää esimerkillä:

```
int[] a = new int[]{1,2,3};

foreach (int b in a) {
  System.Console.WriteLine(b);
}
```

Taulukon lisäksi foreach-silmukassa voidaan käydä läpi mikä tahansa olio, jonka luokka toteuttaa IEnumerable-liittymän. Liittymistä kerrotaan jonkin verran luvussa 4, mutta muuten aihetta ei tässä tutkielmassa tämän enempää käsitellä.

Silmukan suorituksen voi keskeyttää C++:n tapaan break-lauseella ja pelkästään suoritettavan kierroksen continue-lauseella. Goto-lausekin löytyy, mutta kuten muissakin ohjelmointikielissä, sitä ei suositella käytettäväksi.

3.9. Poikkeukset

Poikkeukset toimivat C#:ssa lähes samalla tavalla kuin C++:ssakin. Koska ne eivät kuitenkaan kuulu aivan ohjelmoinnin perusteisiin, käsitellään asia käytännössä nyt uutena.

Jos ohjelmassa esimerkiksi on lause:

```
a = b / c;
```

kääntyy ohjelma kylläkin normaalisti, mutta kaatuu ajonaikaiseen virheeseen, jos muuttujan c arvo suoritushetkellä sattuu olemaan nolla. Muita vastaavanlaisia virhetilanteita voi syntyä esim. taulukolle varatun tilan ylittämisestä ja yrityksestä viitata muistipaikkaan NULL-arvoisen osoittimen avulla.

Normaalisti ongelma ratkaistaisiin kirjoittamalla:

```
if (c != 0) {
  a = b / c;
else {
   ...
}
```

Poikkeukset tarjoavat kuitenkin edellistä hallitumman tavan käsitellä virhetilanteita. Kirjoitetaankin:

```
try {
  a = b / c;
}
catch (System.DivideByZeroException e) {
  ...
}
```

Mahdollisen virheen aiheuttava koodi sijoitetaan try-lohkon sisään ja virheen tapahtuessa suoritettava koodi catch-lohkon sisään. Catch-lohkolle tulee parametriksi (vastaavasti kuin metodeille) tässä tapauksessa DivideByZeroException-luokan olio, jota voidaan tarvittaessa hyödyntää.

Muita virhetilanteita varten on olemassa omat poikkeusluokkansa. Jos try-lohkossa on mahdollista tapahtua useampia virheitä, voidaan kaikille poikkeuksille kirjoittaa peräjälkeen omat catch-lohkonsa. Virhettä vastaavan poikkeuskäsittelijän puuttuminen johtaa ohjelman kaatumiseen.

Haluttaessa käsitellä kaikki virhetilanteet yhdessä catch-lohkossa, kirjoitetaan:

```
catch (System.Exception e) {
  ...
}
```

Yhden tai useamman catch-lohkon jälkeen voidaan vielä kirjoittaa finally-lohko:

```
finally {
  ...
}
```

Lohkon sisältö suoritetaan riippumatta siitä, syntyikö try-lohkossa virhe vai ei.

Joskus voi olla tarpeen aiheuttaa poikkeus tahallaan. Tämä tehdään try-lohkossa varatulla sanalla throw:

```
throw new System.Exception();
```

Esimerkissä luodaan yleinen poikkeus, mutta sanan Exception tilalla voi olla myös erityisen poikkeusluokan nimi.

Ohjelmoija voi lisäksi luoda omia poikkeuksiaan. Asiaa ei kuitenkaan käsitellä tämän tutkielman puitteissa.

4. Olio-ominaisuuksia

Kuten jo johdannossa mainittiin, C# on puhdas oliokieli, joten perinteisiä rakenteellisia ohjelmia sillä ei pysty kirjoittamaan. Lukijan oletetaan tuntevan olio-ohjelmoinnin peruskäsitteet, ja niinpä niistä käydäänkin läpi lähinnä syntaksi. Syvällisempiin ominaisuuksiin, kuten perintään, paneudutaan astetta tarkemmin.

4.1. Luokat ja oliot

Luokan määrittely:

```
class Luokka {
  private int a = 1;
  private int b;
  public void Metodi1(int x) {
    ...
  }
  public int Metodi2(int x, int y) {
    ...
    return z;
  }
}
```

Huomattakoon ensiksi, että C#:ssa ei tunneta erikseen luokan esittelyä ja määrittelyä, koska erillisiä header-tiedostojakaan ei käytetä. Attribuuttien tyypit ilmaistaan samoin kuin C++:ssa. C#:ssa attribuutit voi alustaa suoraan luokan määrittelyssä, kun C++:ssa tämä on tehtävä konstruktorissa.

Kaikkien metodien toteutus kirjoitetaan luokan sisään eli C++:sta tuttua erillistä inline-metodin käsitettä ei C#:ssa ole. Metodien paluuarvo ilmaistaan totuttuun tyyliin ja palautetaan return-lauseella. Return-lausetta voidaan käyttää myös palaamaan void-tyyppisestä metodista.

Metodien parametrit ilmaistaan myös vastaavasti kuin C++:ssa. Oletuksena parametrit viedään arvoina eli vastaava muuttuja ei kutsuvassa ohjelmanosassa muutu, vaikka parametrin arvo metodissa muuttuisikin. Jos muutosten parametrien arvoissa halutaan näkyvän myös kutsujalle, on parametrin eteen ennen tyyppiä kirjoitettava sanaa ref. C++:ssahan sama asia saatiin aikaan &-viittausoperaattorilla.

```
public int Metodi(ref int x) {
  ...
}
```

Kolmas mahdollinen tapa parametrin välittämiseksi on käyttää avainsanaa **`out`**. Kahden jälkimmäisen tavan ero on siinä, että muuttujaa ei out-parametria käytettäessä tarvitse kutsuvassa ohjelmanosassa alustaa.

```
public int Metodi(out int x) {
  ...
}
```

Attribuuttien ja metodien saantimääreet ilmaistaan jokaiselle attribuutille ja metodille erikseen. Private-määreellä varustettu luokan jäsen on käytössä vain luokan sisällä, public-määreellä varustettu myös muissa luokissa. Jollei saantimäärettä kirjoiteta, se on oletuksena private.

Aivan kuten muissakin oliokielissä, attribuutit suositellaan esiteltäviksi private-tyyppisinä ja kirjoittamaan niille tarvittaessa erilliset saanti- ja asetusmetodit:

```
public int GetAttribuutti() {
  return attribuutti;
}

public void SetAttribuutti(int arvo) {
   attribuutti = arvo;
}
```

Asetusmetodiin on nyt helppo lisätä esimerkiksi arvon oikeellisuustarkistus, joka attribuutin arvoa suoraan muutettaessa tulisi lisätä jokaiseen muutoskohtaan koodissa erikseen.

C#:ssa kaikki oliot luodaan dynaamisesti seuraavanlaisella syntaksilla:

```
Luokka olio = new Luokka();
```

Luokan jäseniin viittaaminen tapahtuu pisteellä (C++:ssahan dynaamisesti luodun olion jäseniin viitataan nuolioperaattorilla):

```
olio.Metodi();
```

4.2. Konstruktori ja destruktori

Vaikka C#-luokan attribuutit voidaankin alustaa suoraan luokan määrittelyssä, on usein tarvetta määrätä alkuarvot kutsuvassa ohjelmanosassa, tai sitten tehdä oliota luotaessa muita alustustoimenpiteitä. Tähän tarkoitukseen käytetään erikoista alustusmetodia, konstruktoria.

Konstruktorilla, kuten C++:ssakin, on sama nimi kuin luokalla eikä sillä ole paluuarvoa. Konstruktoreita voi olla useita, jos vain niiden parametrien määrät ja tyypit poikkeavat toisistaan. Jollei ohjelmoija kirjoita luokalle muuta konstruktoria, generoi kääntäjä sille automaattisesti tyhjän parametrittoman konstruktorin.

```
public Luokka( ) {
  ... alustustoimenpiteitä ...
}
```
parametriton konstruktori

```
public Luokka(int a) {
  ... alustustoimenpiteitä ...
}
```
yhden int-tyyppisen parametrin saava konstruktori

Olio luotaisiin nyt edellistä parametrillistä konstruktoria käyttäen seuraavasti:

```
Luokka olio = new Luokka(arvo);
```

Olion elinkaaren aikana tehtyjä dynaamisia muistinvarauksia ei C#:ssa tarvitse automaattisen roskienkeruun ansiosta itse vapauttaa. Silti olion kuollessa on joskus tarpeen tehdä joitakin lopetustoimenpiteitä. Tämä tehdään destruktorissa, jolla ei ole parametreja eikä paluuarvoa. Destruktoreita voi olla vain yksi ja sen syntaksi on sama kuin C++:ssa:

```
~Luokka( ) {
  ... lopetustoimenpiteitä ...
}
```

4.3. Luokan staattiset jäsenet

Jos luokan attribuutin tai metodin eteen kirjoitetaan sana static, tulee jäsenestä luokkaan luoduista olioista riippumaton.

```
private static int attribuutti;

public static void Metodi() {
  ...
}
```

Staattista metodia kutsutaan suoraan luokan nimellä. Luokasta ei tarvitse olla luotuna yhtään oliota.

```
Luokka.Metodi();
```

Pääluokan metodi Main on aina staattinen. Sitä ei tosin koskaan kutsuta itse, vaan käyttöjärjestelmä kutsuu sitä sovelluksen käynnistyessä.

```
public static void Main() {
  ...
}
```

Staattista attribuuttia, joka kannattaa muiden attribuuttien tavoin määritellä private-tyyppiseksi, voidaan käyttää staattisissa metodeissa. Normaaleita attribuutteja niissä ei voida suoraan käyttää.

4.4. Ominaisuudet

Kehitetään saanti- ja asetusmetodien ideaa eteenpäin. Edellisen kappaleen esimerkkimetodeitahan käytettäisiin tyyliin:

```
muuttuja = olio.GetAttribuutti();
olio.SetAttribuutti(arvo);
```

Tehdäänkin nyt attribuutista ominaisuus kirjoittamalla luokan määrittelyyn saanti- ja asetusmetodien tilalle ominaisuusmetodi:

```
public int Attribuutti {
  get { return attribuutti; }
  set { attribuutti = value; }
}
```

Ominaisuusmetodille suositellaan annettavaksi sama nimi kuin muuttujalle, mutta isolla alkukirjaimella aloitettuna. Sana "value" vastaa esimerkissä nyt asetusmetodin parametrin nimeä, eikä sitä saa muuttaa toiseksi.

Ominaisuuden käyttäminen:

```
muuttuja = olio.attribuutti;
attribuutti = arvo;
```

Ominaisuuksien ideana on siis saada saanti- ja asetusmetodien käyttö näyttämään attribuutin suoralta käytöltä, mikä selkeyttää koodia. Visual Basicillä ja Delphillä ohjelmoineet ovatkin ominaisuuksien käyttöön tottuneet.

4.5. Indeksoijat

Indeksoijat ovat C#:n uusi ominaisuus. Niiden avulla luokkaa voidaan käyttää kuin taulukkoa. Parhaiten asia selvinnee esimerkin avulla:

```
class Luokka {

  private int[] taulukko;

  public Luokka() {
    taulukko = new int[10];
  }

  public int this[int a] {
    get {
      return taulukko[a];
    }
    set {
      taulukko[a] = value;
    }
  }

  ...

}
```

Indeksoijalla ei ole nimeä, vaan se muodostetaan avainsanalla this. Indeksoijassa on vastaavat get- ja set-osiot kuin ominaisuuksissa. Sanaa "value" ei saa muuttaa toiseksi.

Luokkaan luotua oliota voitaisiin nyt käyttää seuraavasti:

```
muuttuja = olio[2];
olio[5] = muuttuja;
```

4.6. Delegaatit

C++:lla ohjelmoitaessa tarvitaan joskus funktio-osoittimia. C#:ssa niitä vastaavat delegaatit. Funktio-osoittimista poiketen ne voivat osoittaa useampaankin metodiin kerrallaan.

Delegaattityyppi esitellään luokan jäseneksi seuraavasti:

```
public delegate void Tyyppi(int a);
```

Kyseistä tyyppiä oleva delegaatti voi nyt osoittaa metodeihin, joilla ei ole paluuarvoa, ja jotka saavat yhden int-tyyppisen parametrin.

Koodissa itse delegaatti esitellään ja laitetaan osoittamaan johonkin metodiin seuraavasti:

```
Tyyppi delegaatti = new Tyyppi(Metodi);
```

Jos kyse on jonkin muun luokan metodista, kirjoitetaan:

```
Tyyppi delegaatti = new Tyyppi(Luokka.Metodi);
```

Delegaatin ei alustettaessa välttämättä tarvitse osoittaa mihinkään:

```
Tyyppi delegaatti = null;
```

Seuraavalla syntaksilla delegaattiin voi lisätä uusia metodeita, siis myös useamman kuin yhden:

```
delegaatti = delegaatti + new Tyyppi(Metodi);
delegaatti += new Tyyppi(Metodi2);
```

Vastaavasti voi metodeita poistaa delegaatista operaattoreilla - ja -=.

Delegaatin osoittamia funktioita kutsutaan nyt seuraavasti:

```
delegaatti(parametrit);
```

4.7. Tapahtumat

Tapahtumat ovat oleellinen osa graafisten käyttöliittymien ohjelmointia. Esimerkiksi käyttäjän klikatessa hiirellä jotakin nappulaa, syntyy tapahtuma, johon reagoidaan suorittamalla jokin funktio.

C# tukee tapahtumia suoraan kielen tasolla. Ensinnäkin siinä luokassa, jota tapahtuma koskee, esitellään delegaattityyppi. Parametrit kirjoitetaan samoiksi kuin tapahtuman jälkeen suoritettavalla metodilla halutaan olevan.

```
public delegate void TapahtumaDelegaattiTyyppi
  (string teksti);
```

Kyseistä tyyppiä oleva tapahtumadelegaatti esitellään seuraavasti:

```
public event TapahtumaDelegaattiTyyppi tapahtuma;
```

Sitten kirjoitetaan tapahtuman laukaiseva metodi:

```
public void LaukaisevaMetodi(string teksti) {
   tapahtuma(teksti);
}
```

Mitä hyödyllistä tällä saadaan aikaan? Oletetaan nyt edellisen luokan nimeksi TapahtumaLuokka. Kirjoitetaan seuraavanlainen pääluokka:

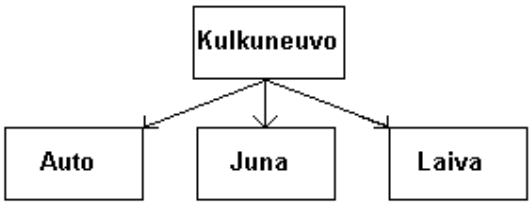
```
class PaaLuokka {

  public static void Main() {

    TapahtumaLuokka olio = new TapahtumaLuokka();
    olio.tapahtuma =
      new TapahtumaDelegaattiTyyppi(TapahtumanKasittelija);

    olio.LaukaisevaMetodi();

  }

  public static void TapahtumanKasittelija(string teksti) {
    System.Console.WriteLine(teksti);
  }

}
```

Tapahtuma siis laukeaa, kun pääohjelmassa suoritetaan metodi LaukaisevaMetodi. Koska metodi TapahtumanKasittelija on liitetty tapahtumaan, siirtyy suoritus seuraavaksi sinne.

Esimerkki on sikäli huono, koska yleensä graafisten käyttöliittymien ohjelmoinnissa tapahtumaa ei yleensä laukaise ohjelmoija itse, vaan se laukeaa esimerkiksi juuri hiirellä nappulaa klikattaessa.

4.8. Perintä

Perintä on yksi olio-ohjelmoinnin keskeisimmistä käsitteistä. Sitä kannattaa käyttää, jos ohjelmassa tarvitaan useita luokkia, joissa suurin osa attribuuteista ja metodeista on samoja. Luokan yhteiset jäsenet kirjoitetaan yliluokkaan ja kullekin luokalle erityiset jäsenet siitä perittäviin aliluokkiin.



Kuva 3. Esimerkki luokkien periytymisestä

Yllä olevassa esimerkissä yliluokalla Kulkuneuvo voisi olla attribuuttina vaikkapa väri ja metodina Kulje( ). Aliluokka Auto perii nämä jäsenet ja siinä voisi edelleen olla määriteltynä vaikkapa metodi TuuttaaTorvea( ), jota ei ole yleisellä Kulkuneuvo-luokalla. Vastaavasti olisi luokissa Juna ja Laiva määritelty näille erityiset attribuutit ja metodit. Luokkahierarkia voi myös olla useampitasoinen, esimerkiksi Auto-luokasta voitaisiin edelleen periä aliluokat Henkilöauto ja Pakettiauto.

Periytyminen ilmoitetaan C#:ssa samalla tavalla kuin C++:ssakin:

```
class Aliluokka: Yliluokka {
  ...
}
```

Aliluokan oliota luotaessa kutsutaan normaalisti sen konstruktoria. Jos kuitenkin yliluokalle on kirjoitettu samanlainen konstruktori (samat parametrimäärät ja tyypit), kutsutaan sitä ensiksi.

Jos aliluokan konstruktorissa halutaan kutsua yliluokan erilaista konstruktoria, tehdään se seuraavalla syntaksilla:

```
public Aliluokka(): base(parametrit) {
  ...
}
```

Mikäli aliluokkaan kirjoitetaan samanniminen metodi kuin yliluokassa, peittyy yliluokan metodi näkyvistä. Jos peittynyttä metodia kuitenkin halutaan kutsua, on kirjoitettava

```
((Yliluokka)olio).Metodi();
```

Jos ollaan jossakin aliluokan metodissa, yliluokan peittynyttä metodia saa kutsutuksi syntaksilla:

```
base.Metodi();
```

Jos yliluokan jäsenten halutaan näkyvän aliluokassa, muttei kuitenkaan muissa luokissa, on ne esiteltävä saantimääreellä protected.

4.9. Polymorfismi

On myös mahdollista luoda aliluokan oliota käyttämällä yliluokan oliomuuttujaa.

```
Yliluokka olio = new Aliluokka(parametrit);
```

Toimintatapaa kutsutaan polymorfismiksi ja siitä on erityisesti hyötyä silloin, kun yliluokan oliomuuttujat ovat taulukossa. Taulukon eri indekseihin voidaan nimittäin luoda eri aliluokkien oliot. Jos eri aliluokissa on samannimiset, eri asian tekevät metodit, voidaan niitä kutsua silmukassa samanlaisella syntaksilla.

Ongelmaksi kuitenkin tulee se, että edellä puhuttu metodi on pakko olla määriteltynä myös yliluokassa, koska sitä kutsutaan yliluokan oliomuuttujan avulla. Tällöin ei aliluokan metodi kuitenkaan tule kutsutuksi.

Ratkaisu on kirjoittaa yliluokassa metodin nimen eteen sana virtual ja aliluokassa puolestaan sana override.

```
class Yliluokka {

  public virtual void Metodi() {
    ...
  }

  ...

}

class Aliluokka: Yliluokka {

  public override void Metodi() {
    ...
  }

  ...

}
```

Nyt yliluokan oliomuuttujaa käytettäessä metodi tulee aina kutsutuksi aliluokasta.

4.10. Liittymät

Moniperinnässä aliluokka peritään kahdesta tai useammasta yliluokasta. Monet olio-ohjelmoinnin asiantuntijat ovat kuitenkin sitä mieltä, että moniperinnästä on enemmän haittaa kuin hyötyä.

C#:ssa ei ole moniperintää. Sen sijaan käytössä ovat Javan tapaan liittymät. Liittymät ovat muuten kuin luokkia, mutta niissä on esitelty vain metodien rungot. Myöskään attribuutteja ei niissä ole.

Esimerkki:

```
interface Liittyma {
  void Metodi();
}
```

Jos luokka toteuttaa jonkun liittymän, siinä pitää olla toteutettuna liittymässä esitellyt metodit:

```
class Luokka: Liittyma {
  public void Metodi() {
    ...
  }
}
```

Moniperintää vastaa se, että luokka toteuttaa useampia liittymiä:

```
class Luokka: Liittyma1, Liittyma2 {
  ...
}
```

4.11. Tietueet

C#:ssa ei ole varsinaista erillistä tietuetyyppiä, kuten ei C++:ssakaan, vaan avainsanalla struct määritellään myös luokka. C++:ssa struct-avainsanan käyttö vaikuttaa luokan näkyvyysominaisuuksiin, mutta C#:ssa struct tarkoittaa, että luokkaa ei voi periä toisesta luokasta, eikä siitä voi periä muita luokkia.

Kolmas mahdollinen avainsana luokan määrittelyyn C++:ssa oli union. Sitä ei C#:sta enää löydy.

4.12. Nimiavaruudet

Nimiavaruudet olivat käytössä jo C++:ssa, mutta C#:ssa ne ovat tärkeämpiä, koska standardikirjaston luokat on ryhmitelty nimiavaruuksiin. Ryhmittelyn lisäksi nimiavaruuksista saatava hyöty on se, että käytössä voi olla useampia samannimisiä luokkia, kunhan ne vain kuuluvat eri nimiavaruuksiin.

Nimiavaruudet määritellään avainsanalla namespace. Ne voivat olla myös sisäkkäisiä. Nimiavaruuteen kuuluviksi haluttujen luokkien määrittelyt kirjoitetaan namespace-lohkojen sisään.

```
namespace Avaruus {
  public class Luokka {
    ..
  }
  namespace AliAvaruus {
    public class Luokka {
      ...
    }
  }
}
```

Nyt esimerkin luokkiin luotaisiin oliot seuraavasti:

```
Avaruus.Luokka olio = new Avaruus.Luokka(parametrit);
Avaruus.AliAvaruus.Luokka olio =
  new Avaruus.AliAvaruus.Luokka(parametrit);
```

Static-määreellä esiteltyjä luokkametodeita puolestaan kutsuttaisiin näin:

```
Avaruus.Luokka.Metodi(parametrit);
Avaruus.AliAvaruus.Luokka.Metodi(parametrit);
```

Avainsanaa using käyttämällä voidaan nimiavaruuden luokat esitellä tunnetuiksi, jolloin nimiavaruutta ei tarvitse erikseen mainita.

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

Ilman using lausetta olisi jouduttu kirjoittamaan:

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

Mainittakoon vielä, että C#:ssa paitsi nimiavaruudet, niin myös itse luokat voivat olla sisäkkäisiä. Tätä ominaisuutta ei kuitenkaan käsitellä tässä tutkielmassa.

5. Yhteenveto

Tässä tutkielmassa käytiin läpi Microsoftin C#-kielen perusteet. Kielen jokaista yksityiskohtaa ei käsitelty, ja siksipä C#-ohjelmoijiksi vakavissaan aikovien kannattaakin tutustua johonkin laajempaan lähdeteokseen. Lisäksi tutkielmassa ei juurikaan puhuttu C#:n standardikirjastosta - useinhan uuteen ohjelmointikieleen siirryttäessä suurin ongelma on nimenomaan sille tehtyjen kirjastojen oppiminen, ei niinkään itse kieli.

C++, vaikka ei ollutkaan ensimmäinen oliokieli, toi olio-ohjelmoinnin suuren yleisön tietoisuuteen. Javan suurin uutuus taas oli välikielelle kääntäminen, joka mahdollisti ohjelmille paremman laitteisto- ja käyttöjärjestelmäriippumattomuuden. C# ei sen sijaan ainakaan kirjoittajan mielestä tuo ohjelmointimaailmaan mitään kauhean mullistavaa. Se on, vaikka Microsoft ei sitä suureen ääneen mainostakaan, selkeästi kehitetty kilpailijaksi Javalle. C#:n tulevaisuutta on vaikea ennustaa, mutta jo 1990-luvun puolivälistä käytössä olleen Javan suosion saavuttaminen ei varmasti tule olemaan helppoa.

Lähteet

[1] Albahari Ben, "A Comparative Overview of C#", saatavilla WWW-muodossa <URL: http://genamics.com/developer/csharp_comparative.htm>, 31.7.2000.

[2] Bartlett Michael ja Touray Sam, "C# Fundamentals", saatavilla WWW-muodossa <URL: http://www.learn-c-sharp.com/csharpfundamentalsoverview.asp>, viitattu 5.2.2002.

[3] Hietanen Päivi, "C++ ja olio-ohjelmointi", Teknolit Oy, Jyväskylä, 1998.

[4] Mayo Joe, "C# Station Tutorial", saatavilla WWW-muodossa <URL: http://www.csharp-station.com/Tutorial.htm>, viitattu 6.2.2002.

[5] Schildt, Herbert, "C#: A Beginners Guide, Sample Chapter", saatavilla WWW-muodossa <URL: http://www.osborne.com/programming_webdev/0072133295/0072133295_ch01.pdf>, viitattu 27.2.2002.

[6] Softsteel Solutions Ltd, "C# Tutorial", saatavilla WWW-muodossa <URL: http://www.softsteel.co.uk/tutorials/cSharp/cIndex.html>, viitattu 27.2.2002.

[7] Wille Christoph, "C# Toolkit", Docendo Finland Oy, Jyväskylä, 2001.