Ohjelmointikielten periaatteet:

C# -kieli

Veli-Matti Sivonen
veli-matti.sivonen@luukku.com

Launonen 4. huhtikuuta 2004
Seminaariesitelmä

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

C# on moderni, yksinkertainen, olio-orientoitunut ja tyyppiturvallinen korkeamman tason ohjelmointikieli [Arc01]. Samalla C# on ensimmäinen komponenttisuuntautunut ohjelmointikieli C/C++ kieliperheessä [DAN03]. C#:n suunnitteluun ja kehittämiseen vaikuttivat monet muut olio-orientoituneet kielet kuten esimerkiksi C++, SmallTalk, Java [PW01]. Päämääränä oli luoda kieli, jossa on: laajentuva tyyppijärjestelmä; 1. luokan komponentien tuenta; luotujen ohjelmien vakaus ja näiden versioiden hallinta sekä yhteensopivuus aiemmin tehtyjen ohjelmistojen ja komponenttien kanssa. Tavoitteena oli myös luoda kieli, joka yhdistää MS Visual Basicin tuottavuuden ja C++:n tehon [Arc01].

1.1 Historiaa

Ensimmäiset huhut Microsoftin uudesta ohjelmointikielestä ilmestyivät 1998, jolloin sitä kutsuttiin nimellä ”Cool”. Microsoft kiisti kuitenkin vielä tällöin nämä huhut, mutta uuden kielen kehitystyö oli alkanut jo 1996 nimellä COM+. Kesäkuussa 2000 julkaistiin C# ja sen jälkeen nopeasti koko .NET Framework SDK. Elokussa 2000 Microsoft, Hewlett-Packard ja Intel esittivät yhdessä Common Language Infrastucture’n (CLI) ja C# ohjelmointikielen kansainvälisen standardointi järjestön, ECMA:n standardoitavaksi. Joulukussa 2001 ECMA julkaisi näistä standardit ECMA-334 (C#) ja ECM-335 (CLI). Samalla CLI:hin liittyvät tekninen raportti ECMA TR84 ratifioitiin [ECMA]. Kielen pääsuunnittelijoina mainitaan Anders Hejlsberg (Turbo Pascal ja Delphi), Scott Wiltamuth ja Peter Golde [DAN03].

2 .NET -alustasta

Kielenä C# on symbioottinen .Net alustan kanssa. C#:lla ei itsellään ole esimerkiksi omia luokkakirjastoja, mutta toisaalta suurin osa .Net -luokkakirjastoista on toteutettu C# -kieltä käyttäen. Tämän vuoksi ennen varsinaista C#-kielen esittelyä on oleellista käsitellä hiukan sen alla toimivaa .Net alustaa.

.Net Framework -koostuu kahdesta osasta: CRL:stä (Common Language Runtime) ja luokkakirjastoista (Framework Class Library, FCL), jotka tarjoavat modernien ohjelmointikielten tarvitsemat palvelut.

2.1 Common Language Runtime, CLR

Alimmaisena ja osittain käyttöjärjestelmän päällä on CRL, joka hoitaa kaikki laitteiston ja käyttöjärjestelmän välisen tiedonvälityksen. CLR on eräänlainen virtuaalikone, jossa ohjelmat suoritetaan toisistaan eristetyissä ja hallituissa ympäristöissä erillään koneen muista prosesseista. Jos jokin resurssi on jaettu, niin jakaminen täytyy tehdä eksplisiittisesti. Tämä merkitsee sitä, että järjestelmistä voidaan tehdä turvallisia niin kutsutun ”hiekkalaatikko periaatteen mukaisesti” [PW01].

Ensimmäiset CLR -versiot löytyvät vuoden 1997 alkupuolelta esimerkiksi Microsoft Transcaction Servet:ssä (MTS), jossa esiintyi lupauksia kuvaavammasta, palveluorientoituneesta ohjelmointimallista. Tämä uusi malli salli ohjelmoijien kommentoida komponentit niiden kehitysaikana, ja luottaa ajoaikaisen ympäristön huolehtivan komponenttien aktivoinnista ja metodikutsujen tulkinnasta sekä palvelujen läpinäkyvästä kerrostamisesta, kuten esimerkiksi tilisiirroista ja JIT -aktivoinniesta. [DAN03].

2.2 Framework Class Library, FCL

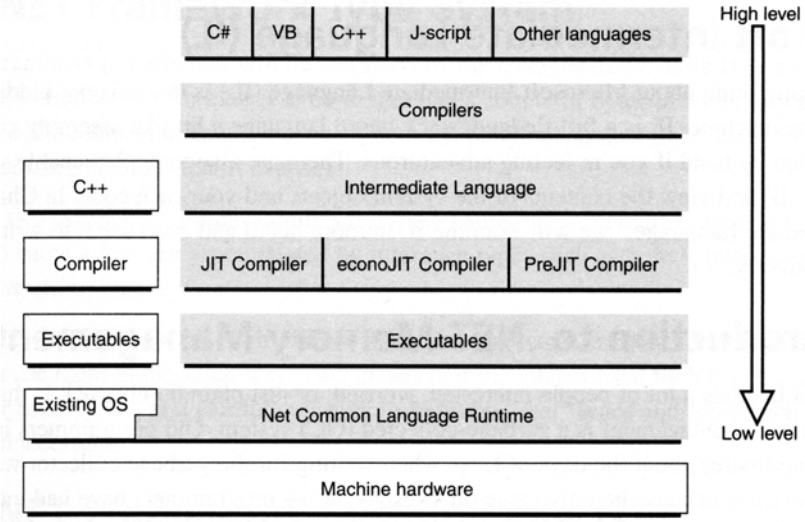
FCL on laaja yli 3 500 luokkaa sisältävä kirjasto, joka tarjoaa rajapinnat kaikkiin CLR:n tarjoamiin palveluihin. Nämä palvelut voidaan ryhmitellä seuraaviin kategorioihin [DAN03]:

- Ydintoimintojen tukeminen; esimerkiksi perustietotyyppien sekä kokoelmien käyttö ja hallinta, konsoliympäristön tuenta, verkkoympäristön palvelut, ja tiedostojen käsittely ja vuorovaikutus muiden suoritusympäristön toimintojen kanssa.
- Tietokantojen käsittely, XML –tiedostojen tulkitseminen ja tuottaminen sekä tabuloidun ja puu –rakenteisen tiedon käsittely.
- Verkkosovellusten teon tuenta, esim. tapahtumaohjatut asiakas - palvelin sovellukset.
- Graafisten sovellusten tuki.
- SOAP –pohjaisten XML-verkkosovellusten teon tuki.

2.3 Just-in-time -käännös

.Net -alustalla toimivat ohjelmat ovat täysin käännettyjä. Käännökset .Net -alustaa käytettäessä tehdään usemmassa vaiheessa. Just-in-time (JIT) kääntäjä huolehtii lopullisesta suoritettavasta käännöksestä. .Net -alustalla ei siten ole bytekoodia kuten yleensä virheellisesti luullaan. Tämä virheellinen käsitys perustuu siihen, että kunkin .Net -alustaa käytävän kielen ohjelmat ensin käännetään alustariippumattomalle välikielelle (CIL, common intermediate language). Tosin kuin perinteisten ohjelmien binääritiedostot, metadatan ja CIL:n yhdistelmä säilyttää melkein kaikki alkuperäisen kielen rakenteet. Kaiken lisäksi tämä yhdistelmä ei riipu käytetystä kielestä, jolloin ohjelmia voidaan rakentaa käyttämällä useampia ohjelmointikieliä (C#; C++, J++, Visual Basic, Delphi, Eiffel, Fortran, COBOL, Haskell, ML, Smalltalk, Scheme ym.), sillä jokainen välikielelle käännetty moduli sisältää täyden kuvauksen itsestään, sillä muun muassa rajapinnat, ominaisuudet, metodit, ja luokat ovat saatavilla metadatana muille moduleille tai IDE:lle (integrated development environment [PW01].

CIL: lle käännetty ohjelma käännetään edelleen JIT-kääntäjää käyttämällä tietokoneen käyttämälle konekielelle. Tämä vie tosin hiukan aikaa, joten järjestelmä tarjoaa mahdollisuuden käyttää preJIT -kääntäjää, joka kääntää CIL-perustaisen koodin pysyvästi konekielelle [PW01]. Alla on kuva .Net -kääntäjistä verrattuna vanhempaan C++ -kieleen.



Kuva 1. .Net -alustan kääntäjät

Edellä mainittuja JIT-kääntäjiä on itse asiassa kolme kappaletta: JIT, koneisiin, joissa on paljon muistia ja tehoa; econoJIT, joka on hyvin nopea, muttei tee optimointeja. EconoJIT perustuu esioptimoituun CIL:n osaan; sekä preJIT, joka kääntää koodin levitettävässä muodossa olevaksi konekieliohjelmaksi [PW01].

Kun CIL:lle tehtyä ohjelmaa suoritetaan, niin taustalle käynnistyy CLR, joka hoitaa ohjelman suorittamisen ja JIT -kääntämisen, muistinhallinnan (roskienkerääjä), poikkeusten hallinnan, debuggerin ja profiilien integroinnin sekä turvallisuuspalvelut (pinon käyttö ja luvitusten tarkistukset). Yleinen tyyppijärjestelmän (Common Type System) kautta CLR saa riittävästi tietoa, metadataa, suoritettavasta koodista, jotta edellä mainitut toiminnot onnistuvat. CTS määrä miten kaikki tyypit ilmaistaan, määritellään ja hallitaan lähdekielestä riippumattomasti. [DAN03].

Edellä kuvattu käännös ja suoritusmalli selittää, miksi C#:a kutsutaan hallituksi kieleksi (managed language) ja miksi koodia, jota CLR suorittaa sanotaan hallituksi koodiksi, ja miksi CLR:n sanotaan tarjoavan hallitun suorituksen (manged execution) [DAN03].

2.4 Yleinen tyyppijärjestelmä

Yleinen tyyppijärjestelmä (CTS) on suunniteltu riittävän monipuoliseksi ja joustavaksi, jotta se tukisi erilaisia ohjelmointikieliä. CTS on järjestelmänä olio-orientoitunut ja se tukee sekä proseduaalisia että funktionaalisia kieliä [Arc01]. .Net -alustaa tukevat ohjelmointikielet joko tukevat täysin yleistä tyyppijärjestelmää, käyttävät vain osaa tyyppijärjestelmästä tai laajentavat sitä, kuten C#. Koska kaikki tyypit esitetään CTS -tyyppeinä ja CTS tukee perintää, niin CLR tukee eri kielten välistä perintää [DAN 2003]. CLR valvoo ohjelmien turvallisuutta ja luotettavuutta muun muassa vahvan tyypityksen ja verifioinnin avulla. CLR käyttää vain CTS:n metadatan kautta tuntemiaan tyyppejä ja varaa niille muistia muistinhallinnan kautta. CLR ei salli käytettävän näkyvyysalueen ulkopuolisia tietoja, metodeja tai prosesseja ilman ekspisiittistä viittausta. Tyyppiturvallinen ohjelma ei voi siten kaatua menemällä ulos sille varatusta muistialueesta tai kirjoittaa toisen prosessin varaamaan muistialueesseen. Tyypityksen kannalta ongelmallisia ovat tilanteet, joissa CLR käyttää hallitun olion (managed object) hallitsematonta koodia (unmanaged code). Tämä tilanne tulee esille, kun käytetään ”vanhoja kirjastoluokkia” [PW01].

Tyyppijärjestelmään kuuluvat arvotyypit ja viittaustyypit sekä osoitintyypit. Koska kaikki tyypit ovat itseasiassa CTS tyyppejä, niin on mahdollista yhdistellä eri kielillä toteutettuja tyyppejä uudella ja mielenkiintoisella tavalla [DAN03].

2.4.1 Arvotyypit

Arvotyyppejä käsitellään ohjelmoijan kannalta C#:ssa aivan kuten C++:ssa tai Javassa. Kuitenkin kaikille primitiivityypin muuttujille täytyy antaa arvo ennen niiden käyttöä aivan kuten Javassa. Arvotyypit ovat .Net -alustassa toteutettu tietueina, joten ne sijoitetaan suoritusaikana pinoon. Näin niiden elinaika liittyy niiden luoneen rakenteen näkyvyysalueeseen [PW01]. Toisin kuin C/C++:ssa ohjelmoija voi luoda ”uusia” primitiivityyppejä. Tässä mielessä C#:n luokat ja tietueet ovat samanlaiset. Tietueet ja luokat eivät kuitenkaan ole sukua toisilleen ja niissä on monia eroavuuksia, sillä tietue ei voi periytyä toselta luokalta tai tietueelta. Toisaalta tietue voi toteuttaa yhden tai useamman rajapinnan. [PW01].

Arvotyypit ovat helpoimmat ymmärtää, koska ne sisältävät suoraan arvonsa. Esim. int -tyyppinen muuttuja sisältää kokonaisluvun ja bool –tyyppinen muuttaja sisältää boolean arvon *true* tai *false*. Arvotyypisiä muuttujia sijoitettaessa muuttujan arvo kopioidaan kohdemuuttujan arvoksi. Viittaustyypin muuttujien kohdalla kopioidaan arvoa koskeva viittaus kohdeolioon [DAN03]. C++:aan ja Javaan verrattuna uutena arvotyyppinä esitetään *desimal* –tyyppi, joka ilmasee desimaaliluvun 128 bitillä ja näyttää sen tarvittaessa 28 merkitsevän numeron tarkkuudella [ECMA-334].

```
using System;
using IO = System.Console;     // Alias nimi

class Test
{
  static void main()
  {
    int x = 3;
    int y = x;      // sijoitetaan x y:hyn
    x++;
// kasvatetaan x:ää yhdellä
    WriteLine(y);  // Tulostaa 3
  }
}
```

Kokonais- ja liukulukutyyppejä sanotaan yksinkertaisiksi, koska niiden esitystapa vastaa yleensä konekielistä esitysmuotoa. Niillä on kaksi käyttökelpoista ominaisuutta: ne ovat tehokkaista ja arvon kopiointisemantiikka on helppo ymmärtää. Javassa näitä tietotyypejä kutsutaan primitiivisiksi tietotyypeiksi. C#:ssa toisin kuin Javassa näitä yksinkertaisia tietotyyppejä voidaan luoda myös itse määrittelemällä ne tietueiksi [DAN03].

Alla on esimerkki siitä kuinka C#:ssa voidaan luoda uusi primitiivityyppi ja kuinka sitä käytetään.

```
using System;

namespace Struct
{
  /// <summary>        //<- C# Tuottaa HTML:ää suoraa koodista!!!
  /// Murtoluku
  /// </summary>
  struct Murtoluku
  {
    public int osoittaja;
    public int nimittäjä;

    public void Print()
    {
```

```
      Console.WriteLine("{0}/{1}", osoittaja, nimittäjä );
    }
  }

  class Class1
  {
    /// <summary>        // Jos omistaa IDE:n niin nämä saa piiloon!!
    /// Tietue testi
    /// </summary>
    [STAThread]          // <-  attribuutti, jonka voisi jättää myös pois
    static void Main(string[] args)
    {
      Murtoluku f;
      f.osoittaja = 6;
      f.nimittäjä = 12;
      f.Print();          // 6/12

      Murtoluku f2 = f;
      f2.Print();         // 6/12

      // Muokataan vähän f2:ta
      f2.osoittaja = 3;

      f.Print();          // 6/12
      f2.Print();         // 3/12
    }
  }
}// namespace Struct
```

Jotta jokainen operaatio voitaisiin toteuttaa sekä viittaus- että arvotyypeillä, niin jokaiseen arvotyyppiin liittyy vastaava piilotettu viittaustyyppi, joka sisältää varastoidun (”boxed”) arvon. Tällä menettelyllä pyritään estämään se epätietoisuus, mitä tehdään, kun viittattu arvo viedään pinoon ja viittauksen näkyvyysalue loppuu. .Net -alustassa CRL kopioi arvon kekoon ja viittaus osoittaa keossa olevaan varastoituun (”boxed”) kopioon. Näkyvyysalueelta poistuttaessa arvo kopioidaan (”unboxed”) takaisin keosta. Tämä menettely sallii arvo- ja viittaustyyppien yhdenmukaisen käyttämisen, mutta ei aiheuta viittaustyyppeihin liittyvää kuormitusta [ECMA-334].

2.4.2 Viitaustyypit

Viittautyypin muuttujissa tieto sisältyy viitattavaan olioon. C#:ssa kaikki oliot sijoitetaan hallittuun kekoon (”managed heap”). Viittaustyypiä ovat mm. luokat, taulukot, delegaatit ja rajapinnat. Viittaustyyppi määrittelee itseasiassa kaksi erillistä asiaa: olion ja siihen liittyvän viittauksen. Alla oleva esimerkki valaisee tätä asiaa.

```
using System;
using System.Text;

class Class1
{
  static void Main()
  {
    StringBuilder x = new StringBuilder("Hei,");
    StringBuilder y = x;
    x.Append(" siellä!");
    Console.WriteLine(y);   // Tulostaa:  Hei, siellä!

    String x2 = "Hei,";
    String y2 = " siellä2!";
    Console.WriteLine("{0}", x2 + y2); // Tulostaa:  Hei, siellä2!
  }
}// class Class1
```

2.4.3 Osoitintyypit

Osoitintyyppejä ei pidetä varsinaisesti C#:n perustyypeinä, mutta ne on sisällytetty kieleen, jotta yhteensopivuus C/C++ -ohjelmien kanssa säilytettäisiin. Niinpä osoitintyyppi on esitelty spesifikaation liitteessä ikään kuin välteltävänä ominaisuutena. C# sallii muistin suoran käsittelyn *unsafe*-määreellä varustetun koodilohkon sisällä koodissa, joka on käännetty käyttäen kääntäjän ”/unsafe” valintaa.

C#:ssa unsafe, turvaton koodi, on itseasiassa turvallinen sekä ohjelmoijien että käyttäjien kannalta, sillä turvaton koodi on selvästi merkittävä eikä tätä ominaisuutta voi käyttää vahingossa. Lisäksi suoritusympäristö pitää huolen, ettei turvatonta koodia voida käyttää oman luotetun ympäristönsä ulkopuolella [ECMA-334].

Joitakin esimerkkejä osoittimista:

byte*	osoitin tavuun
char*	osoitin merkkiin
int**	osoitin int –tyypin osoittimeen
int*[]	yksidimenssioinen taulukko osoittimia kokonaislukuun
void*	osoitin tuntemattomaan tyyppiin

Joissain tapauksissa olio, esimerkiksi taulukko, on kiinnitettävä, jotta roskien kerääjä ei siirtele sitä pitkin hallittua kekoa, kun käytetään osoittimia. Tämä tehdään käyttäen *fixed* – määrettä.

```
using System;
using IO = System.Console;

class Test
{
  static void Main()
  {
    int[,,] a = new int[2,3,4];
    unsafe
    {
      fixed (int* p = a)
      {
        for (int i = 0; i < a.Length; ++i)   // treat as linear
          p[i] = i;
      }
    }
    for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 3; ++j)
      {
        for (int k = 0; k < 4; ++k)
          Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
        WriteLine();
      }
  }
}
```

```
[0,0,0] = 0   [0,0,1] = 1   [0,0,2] = 2   [0,0,3] = 3
[0,1,0] = 4   [0,1,1] = 5   [0,1,2] = 6   [0,1,3] = 7
[0,2,0] = 8   [0,2,1] = 9   [0,2,2] = 10  [0,2,3] = 11
[1,0,0] = 12  [1,0,1] = 13  [1,0,2] = 14  [1,0,3] = 15
[1,1,0] = 16  [1,1,1] = 17  [1,1,2] = 18  [1,1,3] = 19
[1,2,0] = 20  [1,2,1] = 21  [1,2,2] = 22  [1,2,3] = 23
```

3 C# -ohjelmointikieli

C# on vahvasti ja staattisesti tyypitetty olio-orientoitunut ohjelmointikieli, joka on suunniteltu antamaan optimaalisen sekoituksen yksinkertaisuutta, ilmaisuvoimaa ja suorituskykyä. Staattinen tyypitys tarkoittaa, että nimisidonnat tehdään käännösaikana [Sco00]. Syntaktisesti ja ohjausrakenteiltaan C# muistuttaa C/C++:ta. Tämä ei ole vahinko, sillä yksi kielen suunnittelun päämääristä oli säilyttää C/C++:n hyvät ominaisuudet ja muuttaa tai laajentaa kielen syntaksia vain, jos siihen oli tarvetta [Arc01]. Tehdyistä muutoksista tässä voisi mainita sen, että kaikki muuttujat pitää alustaa ennen käyttöä, *if* - ja *while* –käskyt tarvitsevat boolean arvon, jolloin '=' -operaattorin virheellinen käyttö estyy, ja *switch* –rakenteen läpijuoksu on estetty, jolloin puuttuva *break* -komento ei aiheuta ylimääräisten rakenteiden läpikäyntiä [Arc01, PW01]. Semanttisesti C# muistuttaa Javaa, sillä esimerkiksi operandit evaluoidaan vasemmalta oikealle. Poikkeuksen tästä tekevät sijoitusoperaattori ja ehto-operaattori (? :), jotka evaluoidaan oikealta vasemmalle [ECMA-334].

C# -ohjelma koostuu yhdestä tai useammasta lähdetiedostosta, jotka ovat järjestettyjä Unicode sarjoja. Varatut sanat kirjoitetaan C/C++:n ja Javan tapaan pienillä kirjaimilla. Isot ja pienet kirjaimet tulkitaan eri merkeiksi kuten Modula-2/3, C, C++ ja Javassa [Sco00]. Kuten monissa muissa moderneissa kielissä "valkoiset merkit" jätetään huomioimatta.

C# tukee jossain määrin C++:n tapaisia esiprosessorikäskyjä. Erona on mm. se, että esiprosessorikäskyt suoritetaan vasta kielioppia tarkistettaessa. Esiprosessorikäskyt eivät kuitenkaan varsinaisesti kuulu C# -syntaksiseen kielioppiin. Esiprosessorikäskyillä voidaan kuitenkin liittää tai poistaa valinnaisesti koodia ja siten vaikuttaa varsinaiseen ohjelmaan [ECMA-334].

Tarkasti ottaen, C#:ssa ei itsessään ole muistinhallintaa, vaan sen suorittaa .Net –alustalle yhteinen välikieli (Intermediate Language, IL), jolle C# -ohjelmat käännetään. Olioita luotaessa IL:n roskienkerääjä varaa niille tilaa "hallitusta keosta" (managed heap) tai tietueiden osalta suorituspinosta. Hallittu keko eroaa tavallisesta keosta pääosin siinä, että siinä roskienkerääjä varaa muistia jatkuvista muistialueista ja roskienkeräys sekä keon tiivistäminen tehdään optimoidusti vaiheittain, jolloin varsinaisen ohjelman suoritus kärsii roskien keruusta mahdollisimman vähän [ECMA-334].

3.1 C# -kielen uudet ominaisuudet

Kielenä C# tukee kaikkia kolmea Bjarne Stroustrupin mainitsemaa olio-ohjelmointikielten periaatetta; objekteja, luokkia ja periytymistä. Lisäksi se tukee vielä kapselointia ja monimuotoisuutta, joten se toteuttaa kaikki modernin olio-ohjelmointikielten tunnusmerkit [Arc01]. Koska Javan voidaan olettaa olevan tuttu kaikille tämän seminaariesityksen lukijoille, niin niitä ominaisuuksia, joilla Javaa ja C#:a on parannettu C++:n nähden, ei enää käsitellä. Niistä tärkeimpiä on mainittu lyhyesti liiteessä A. Siten tässä esityksessä keskitytään vain joihinkin ominaisuuksiin, jotka voidaan katsoa olevan parannuksia Javaan verrattuna tai jotka puutuvat Javasta, vaikka ne ovat C++:ssa.

3.1.1 Ominaisuudet

Ominaisuudet (properties) ovat nimettyjä luokkien, tietueiden tai rajapintojen jäseniä. Ne tarjoavat joustavan tavan lukea, kirjoittaa tai laskea ”private” –kenttien arvoja käyttämällä ”get” ja ”set” aksessoreita. Ominaisuudet C#:iin tulevat Delphi- ja Visual Basic –kielistä [PR03].

```
public int Size
{
    get {  return size; }
    set  { size = value; }
}
```

3.1.2 Indeksoijat

Indeksoijat (indexers) tarjoavat luonnollisen tavan indeksoida luokkien tai tietueiden elementtejä siten, että kokoelma kapseloidaan taulukkosyntaksilla. Indeksoijat ovat kuten ominaisuudet, mutta viittaukset tehdään indeksien kautta, kun ominaisuuksissa käyteään nimeä [DAN03]. Alla esimerkki, joka valaisee myös rajapintojen käyttöä.

```
using System;
using System.Collections;
using IO = System.Console;


class MyListBox
{
  protected ArrayList data = new ArrayList();

  public object this[int idx]
  {
    get
    {
      if ( -1 < idx && idx < data.Count )
        return (data[idx]);
      else
        return null;
    }
    set
    {
      if ( -1 < idx && idx < data.Count )
         data[idx] = value;
      else
        data.Add(value);
    }
  }
}// class MyListBox


class Indeksoija
{

  [STAThread]
  static void Main(string[] args)
  {
    MyListBox lbx = new MyListBox();
    lbx[0] = 123;
    lbx[1] = "fool";
    lbx[2] = "wise";


    IO.WriteLine("{0} {1} {2}", lbx[0], lbx[1], lbx[2]); // 123 fool wise
  }
}
```

3.1.3 Delegaatit (delegates)

Delegaatit ovat olioita, jotka toimivat kuten Delphi -kielen tyyppisuojatut funktio-osoittimet. Delegaateilla käsitellään ongelmia, jotka C++:ssa ratkaistaisiin funktio-osoittimien avulla. Delegaatti on tyyppiturvallisena parempi kuin funktio-osoitin [Col04]. Delegaattien avulla vältytään ylimääräisen koodin kirjoittamiselta, kun kutsutaan eri metodeja, jolla on samanlaiset argumentit ja palautustyypit. Esimerkissä kuvataan myös tietueen tekoa.

```
using System;
using IO = System.Console;

public struct Person
{
  public string FName;
  public string LName;
}

public delegate void OnNewHire(Person person);

public class HR
{
  public void OnNewHire( Person person )
  {
    IO.WriteLine("Palkataan {0}", person.FName );
  }
}

public class Department
{
  private OnNewHire m_OnNewHireDelegate = null;

  public void AddOnNewHireDelegate( OnNewHire onh )
  {
    m_OnNewHireDelegate = onh;
  }

  public void HirePerson( Person p )
  {
    if ( m_OnNewHireDelegate != null )
      m_OnNewHireDelegate( p );
  }
}

class Delegaatti
{

[STAThread]
  static void Main(string[] args)
  {
    HR hr = new HR( );
    Department dept = new Department( );

    dept.AddOnNewHireDelegate( new OnNewHire( hr.OnNewHire ) );

    Person me;
    me.FName = "Veli-Matti";
    me.LName = "Sivonen";

    dept.HirePerson( me );
  }
}// Delegaatti

// Tulostus: Palkataan Veli-Matti
```

3.1.4 Tapahtumat (events)

C# tukee suoraan tapahtumia. Tapahtumat ovat prosesseja, joissa olio voi ilmoittaa toiselle oliolle, että tapahtuma on tapahtunut [DAN03].

```
using System;
using IO = System.Console;

// FireEventArgs: a custom event inherited from EventArgs.

public class FireEventArgs: EventArgs
{
  public FireEventArgs(string room, int ferocity)
  {
    this.room = room;
    this.ferocity = ferocity;
  }

  // The fire event will have two pieces of information--
  // 1) Where the fire is, and 2) how "ferocious" it is.

  public string room;
  public int ferocity;

}    //end of class FireEventArgs


// Class with a function that creates the eventargs and initiates the event
public class FireAlarm
{
    // Events are handled with delegates, so we must establish a FireEventHandler
    // as a delegate:
  public delegate void FireEventHandler(object sender, FireEventArgs fe);

    // Now, create a public event "FireEvent" whose type is our FireEventHandler delegate.
  public event FireEventHandler FireEvent;

    // This will be the starting point of our event-- it will create FireEventArgs,
    // and then raise the event, passing FireEventArgs.
  public void ActivateFireAlarm(string room, int ferocity)
  {
    FireEventArgs fireArgs = new FireEventArgs(room, ferocity);
      // Now, raise the event by invoking the delegate. Pass in
      // the object that initated the event (this) as well as FireEventArgs.
      // The call must match the signature of FireEventHandler.
    FireEvent(this, fireArgs);
  }
}  // end of class FireAlarm


// Class which handles the event
class FireHandlerClass
{
    // Create a FireAlarm to handle and raise the fire events.
  public FireHandlerClass(FireAlarm fireAlarm)
  {
      // Add a delegate containing the ExtinguishFire function to the class'
      // event so that when FireAlarm is raised, it will subsequently execute
      // ExtinguishFire.
    fireAlarm.FireEvent += new FireAlarm.FireEventHandler(ExtinguishFire);
  }

    // This is the function to be executed when a fire event is raised.
  void ExtinguishFire(object sender, FireEventArgs fe)
  {

    IO.WriteLine("\nThe ExtinguishFire function was called by {0}.", sender.ToString());
    // Now, act in response to the event.

    if (fe.ferocity < 2)
      IO.Write("This fire in the {0} is no problem.", fe.room);
      IO.WriteLine("I'm going to pour some water on it.");
    else if (fe.ferocity < 5)
      IO.WriteLine("I'm using FireExtinguisher to put out the fire in the {0}.",  fe.room);
    else
      IO.Write("The fire in the {0} is out of control.",  fe.room);
      IO.WriteLine("I'm calling the fire department!");
  }
} //end of class FireHandlerClass


public class FireEventTest
{
  public static void Main ()
  {
      // Create an instance of the class that will be firing an event.
    FireAlarm myFireAlarm = new FireAlarm();

      // Create an instance of the class that will be handling the event. Note that
      // it receives the class that will fire the event as a parameter.
```

```
    FireHandlerClass myFireHandler = new FireHandlerClass(myFireAlarm);

      //  use our class to raise a few events and watch them get handled
    myFireAlarm.ActivateFireAlarm("Kitchen", 3);
    myFireAlarm.ActivateFireAlarm("Study", 1);
    myFireAlarm.ActivateFireAlarm("Porch", 5);

    return;

  } //end of main
} // end of FireEventTest

/* The ExtinguishFire functon was called by FireAlarm.
   I'm using FireExtinguisher to put out the fire in the Kitchen.

   The ExtinguishFire functon was called by FireAlarm.
   This fire in the Study is no problem. I'm going to puor some water on it.

   The ExtinguishFire functon was called by FireAlarm.
   The fire in the Porch is out of control. I'm calling the fire department!
*/ PR03
```

3.1.5 ref-, out- ja params -parametrit

Tosin kuin Java, C# sallii muuttujien välityksen viitteinä. Tämä tapahtuu ref –parametrin avulla. Kun muuttujalla ei ole metodikutsussa arvoa, se voidaan välittää kutsuttavalle metodille out-parametria käyttäen. Tällöin kutsuttu metodi asettaa viitatulle muuttujalle arvon. ”params”-parametriä käytettäessä metodiin voidaan lisätä useampia samantyyppisiä muuttujia.

```
using System;
using IO = System.Console;

class refout
{
  public static void swap(ref int a, ref int b, out int c)
  {
    int temp = a;
    a = b;
    b = temp;
    if ( a < b )
      c = b;
    else
      c = a;
  }

  [STAThread]
   static void Main(string[] args)
   {
    int a = 1;
    int b = 2;
    int suurempi;
    swap(ref a, ref b, out suurempi);
    IO.WriteLine(" {0} {1} {2}", a, b, suurempi);  // 2 1 2
   }
}// refout
```

3.1.6 Säännölliset taulukot

C#:ssa voidaan luoda sekä säännöllisiä että perinteisiä taulukoita. Säännöllisten taulukoiden etuna on mm. niiden tehokkuus ja selkeys.

```
int [,,] array = new int[3,4,5];        // luo vain yhden taulukon
int[][][] arrau = new int [3][4][5];   // luo 16 taulukkoa.
```

Liite A

Lista Javan ja C#:n ominaisuuksista, jotka on tarkoitettu kehittämään C++:aa:

- ohjelmat käännetään laitteistoriippumattomaksi koodiksi, joka suoritetaan hallitussa suoritusympäristössä (JVM ja CLR)

- roskienkerääjä yhdistettynä osoittimien käytön estoon (C# sallii tosin osoittimien käytön rajoitetusti koodissa, joka on merkitty avainsanalla ”unsafe”)

- olioiden ja luotujen tyyppien tietoisuus tyypistään (reflection)

- ei käytetä erillisiä otsikkotiedostoja (header files).

- näkyvyysalueet

- luokat periytyvät object –luokasta ja uudet oliot luodaan avainsanaa ”new” käyttäen

- säikeitten tuki lukitsemalla oliot, kun koodissa on määre ’locked’ tai ’synchronized’

- rajapinnat, rajapintojen moniperintä, toteutuksen yksittäinen perintä (ei moniperintää!)

- sisäluokat

- luokkaa ei voida periä tiettyllä määritetyllä tavalla

- ei globaaleja funktioita tai vakioita; kaikki kuuluvat luokille

- taulukot ja merkkijonot, jossa niiden pituus on sisäänrakennettuna, rajojen tarkistus

- käytetään aina ”.” operaattoria, ei enää “->” ja “::” operaattoreita viittauksissa

- “null” ja “Boolean/ Bool” ovat avainsanoja

- kaikki arvot on alustettava ennen käyttöä

- “if” lause tuottaa totuusarvoksi Boolean tyypin “true” tai ”false”
  eikä ”0” ja jokin muu kokonaisluku

- ”try” lohkolla voi olla ”finally” lause.

(Yllä oleva lista perustuu Javan version 1.3 ja C# versioon 1.1, joita esitelmänkirjoittaja on käyttänyt. Javasta on julkaistu myöhemmin versiot 1.4 ja 1.5.)

Lähdeluettelo:

Arc01

Arcer, Tom, Inside C#, Edita Oyj, Helsinki 2001.

Col04

Collingbourne, Huw, C# Programmers' World, PCPlus, Issue 215, May 2004 Future Publishing, United Kingdom, 2004.

DAN03

Peter Drayton, Ben Albahari & Ted Neward, C# in a Nutshell, Second Edition, O'Reilly & Associates, Inc. USA, 2003.

D03

Deitel, H. M.; Deitel, P. J. Listfield, J. A.; Nieto, T. R.; Yager, C. H. & Zlatkina, M. C#: A Programmer's Introduction, Pearson Education, Inc. Upper Saddle River, New Jersey, USA, 2003

ECMA

ECMA Standardonti järjestön verkkosivut
http://www.ecma-international.org

ECMA-334

C# Languge Specification

PR03

C# Programmer's Reference, Microsoft Visual Studio .NET 2003.

PW01 Robert Powell & Richard Weeks, C# and the .NET Framework, Sams Publishing, USA, 2001.

Sco00

Michael Lee Scott, Programming language pragmatics, Morgan Kaufmann Publishers, USA. 2000.

SJ03

Sharp, John & Jagger, Jon, Microsoft Visual C#.NET Step by Step: Version 2003, Microsoft Press, Redmond, Washington, USA, 2003

Lähdeviitteiden käyttö noudattelee Matti Mäkelän opasta kirjoitelman laatimiseksi. Matti Mäkelä: Kirjoitelman laatiminen, http://www.cs.helsinki/u/mamakela/kirjlaat.html (23.1.2004)