Paavo Räisänen

C# olio-ohjelmointi perusopas

www.ohjelmoimaan.net

Sisällysluettelo

1: Johdanto

C# on täysin olio-ohjelmointikieli. Oliot eli objektit vastaavat perinteisen ohjelmoinnin aliohjelmia eli funktioita. Olio-ohjelmointi on kuitenkin huomattavasti kehittyneempi, ja myös monipuolisenpi, mutta valitettavasti myös monimutkaisenpi tekniikka. Kun olio-ohjelmoinnin oppii kuitenkin sisäistämään, niiden käyttö sujuu siinä kuin funktioidenkin. Esim. C kielessä käytetään olioiden sijaan funktioita, ja taasen C++ ja PHP ovat hydridikieliä, eli niissä on sekä oliot, että funktiot. C#:in lisäksi esim. Java on puhdas olio-ohjelmointikieli. Olio-ohjelmointi on englanniksi ”object-oriented programming”.

Tässä oppaassa oletetaan, että olet lukenut C# perusoppaan näiltä sivuilta.

2: Mikä luokka on?

Luokka eli class on tietynlainen kokonaisuus, joka sisältää informaatiota, esim. muuttujia (ominaisuuksia) ja toimintoja (metodeita).
Olio-ohjelmoinnissa puhutaan kapseloinnista, eli luokka sisältää tietoa, ja sitä käyttävät metodit (ikäänkuin aliohjemat). Metodeita käyttävän ohjelman ei tarvitse tietää, miten luokan sisälle metodit on toteutettu (implementoitu). Vain sillä on merkitystä esim. pääohjelmaa tehtäessä, mitä metodi tekee. C#:ssa on paljon valmiita metodeita, kuten esim. Console.WriteLine. Käytettäessä tätä metodia, sinun ei tarvitse tietää miten tämä metodi on toteutettu, vaan merkitystä on vain sillä, mitä tämä metodi tekee. Luokkia ja metodeita käyttämällä voit siis ikäänkuin luoda omia käskyjä. Olio-ohjelmointi voi aluksi tuntua konstikkaalta varsinkin C tai C++ tyyppisiin funktioihin tottuneesta, mutta luokat ja oliot ovat hyvin käytännöllisiä, kun ne oppii. Niiden käytännöllisuus tulee ilmi eritoten pitkissä, ehkä monimutkaisissakin ohjelmissa. Tiedon kapselointi ja monet muut toiminnos luovat selkeän rakenteen, jossa esim. ohjelman jatkotuotekehitys on helppoa ohjelman selkeän rakenteen ansiosta. Juuri esim. se, että ohjelmoijan ei tarvitse välttämättä lainkaan olla selvillä, miten luokka toimii, kunhan tietää miten sen metodi toimii, ja mitä se tekee.

Luokan muuttujia, esim. ”int lukumaara”, kutsutaan kentiksi. Kentät ja metodit muodostavat luokan rungon.

Esim. kirjastojärjestelmästä voisi tehdä luokan:

```
class Kirjasto
{
  //kentät
  private int erapaiva;
  private int tyyppi;
  private string nimi;

  //metodit
  int Erapaiva();
  int Tyyppi();
  string Nimi();
}
```

Nyt voisit tehdä metodin Erapaiva(), joka tallentaa kenttään erapaiva eräpäivän, metodin Tyyppi(), joka tallentaa kenttään tyyppi sen, onko kyseessä kirja, lehti vai äänite (numeroilla esim. kirja : 1, lehti: 2, äänite: 3) ja metodin Nimi(), joka tallentaa kenttään nimi lainauksen nimen.

Uusi olio luodaan new avainsanalla. Esim. Kirjasto luokkaan:

```
Kirjasto tallenne;
tallenne = new Kirjasto();
```

tai samalla rivillä:

```
Kirjasto tallenne = new Kirjasto();
```

Nyt voit kutsua luokan metodeita seuraavasti:

```
int uusi_erapaiva = tallenne.Erapaiva();
int tyyppi = tallenne.Tyyppi();
string nimi = tallenne.Nimi();
```

Esim. metodiin Erapaiva() vain kirjoitettaisiin sitten ohjelma, joka kysyy lainausajan (tai katsoo sen tietokoneen kellosta) ja laskee sitten jäljellä olevan lainausajan, eräpäivän (esim. 30 päivää).

3: Luokan määrittely ja käyttö

C#:ssa käytetään class avainsanaa, kun määritellään luokka. Luokassa ohjelman data ja metodit sijaitsevat aaltosulkeiden {} välissä luokan sisällä.
Yllä tehtiin jo luokka Kirjasto jossa oli kentät (datapalaset) ja metodit.
Oliot määritellään kuten yllä new avainsanalla seuraavasti, ja ne toimivat vain saman nimisessä luokassa.
Kirjasto tallenne = new Kirjasto(); //luodaan tallenne olio
Kirjasto toinentallenne;
tallenne = toinentallenne; //hyväksyttävää, kun on sama luokka

Termejä luokka (eli class) ja olio (eli objekti) ei saa sekoittaa. Luokka on tyypin määrittely, ja olio on siitä ajon aikana luotava instanssi, ilmentymä. Ilmentyma eli instanssi luodaan luokasta new avainsanalla. Ilmentymä, instanssi, olio ja objekti ovat käytännössä sama asia. Jos aivan tarkkoja ollaan, instanssi on kuitenkin muistin hallinnallisesti hieman poikkeava oliosta, vaikka useissa ohjelmointioppaissa ne esitellään samana asiana. Tätä eroa ei tässä oppaassa käsitellä, eikä sen osaaminen ole välttämätöntä.

4: Saannin kontrollointi

Kun luokan metodit ja oliot kapseloidaan luokan sisälle, niitö ei voi noin vain käyttää luokan ulkopuolella. Luokalla on ikäänkuin suojakuori ulkopuolisia yhteydenottoja varten, ja ne ikäänkuin salaavat tietoaan.

Muutetaan nyt Kirjasto luokkaa niin, että siihen pääsee käsiksi myös luokan ulkopuolelta.

```
class Kirjasto
{
  //kentät
  private int erapaiva;
  private int tyyppi;
  private string nimi;

  //metodit
  public int Erapaiva();
  public int Tyyppi();
  public string Nimi();
}
```

Nyt tässä ohjelmassa pääsee eri ohjelmakohkoista käsiksi Kirjasto luokan metodeihin. Sensijaan kentät on private tyyppisinä kapseloitu luokkaan, eli niihin ei suoraan pääse käsiksi. Voisi tietenkin tehdä kentän public int tyyppi, johon pääsisi muualtakin käsiksi, mutta se ei ole suotavaa kapseloinnin periaatteiden mukaan.

Kannattaa suosia mahdollisimman paljon kapselointia, vaikka joitain kenttiä joutuu ehkä joskus laittamaan julkisiksi eli public kentiksi.
Jos kentän tai metodin näkyvyyden jättää määrittelemättä, se on oletusarvoisesti privat, mutta on hyvä tapa määritellä myös privat muuttujat niin, että määrittely näkyy. privat muuttujat eivät näy kuin luokan sisällä. Muualta niihin ei pääse käsiksi kuin julkisten public metodien kautta.
C#:ssa luokkien kentät alustetaan aina oletusarvoisesti joko arvoksi 0 (int, float,double jne.), tai false (bool), tai null (string). Huomaathan, että vaikka muuttujien automaattinen alustus toimii luokissa, yleensä ohjelmassa muuttujat on jotenkin alustettava ennen ensimmäistä käyttöä.

5: Konstruktorit

Jokaisella luokalla on oltava konstruktori, joka tekee kentistä instanssit. Jos et itse kirjoita konstruktoria, kääntäjä tekee sen, ja antaa kentille arvot 0, false tai null tietotyypistä riippuen. Nimittäin, kun luot new avainsanalla olion, ohjelman täytyy pystyä rakentamaan olio, alustamaan kentät ja varata tilaa muistista.

Voit myös itse kirkoittaa oletuskonstruktorin, mikä on suositeltavaa. Kirjasto luokkaan se luotaisiin seuraavasti:

```
class Kirjasto
{
  //kentät
  private int erapaiva;
  private int tyyppi;
  private string nimi;

  //metodit
  public int Erapaiva();
  public int Tyyppi();
  public string Nimi();

  //Oletuskonstruktori
  Public Kirjasto()
  {
    erapaiva = 0; //oletetaan, että tässä on lainausaika, esim. 30 päivää.
                  //ei koko päiväystä, joka vaatisi useamman eri muuttujan
    tyyppi = 0;
    nimi = ””; //kaksi lainausmerkkiä alustaa merkkijonon arvoksi null
  }
}
```

5.1: Konstruktorien ylikuormitus

Edellisen oletuskonstruktorin luomisen jälkeen olio voidaan kyllä luoda, mutta sen sisältö on tyhjiä arvoja. Muokataan vielä konstruktoria.

```
class Kirjasto
{
  //kentät
  private int erapaiva;
  private int tyyppi;
  private string nimi;

  //metodit
  public int Erapaiva();
  public int Tyyppi();
  public string Nimi();

  //Oletuskonstruktori
  Public Kirjasto()
  {
    erapaiva = 0; //oletetaan, että tässä on lainausaika, esim. 30 päivää.
                  //ei koko päiväystä, joka vaatisi useamman eri muuttujan
                  //eli päivät ja kuukaudet
    tyyppi = 0;
    nimi = ””; //kaksi lainausmerkkiä alustaa merkkijonon arvoksi null
  }

   //Ylikuormitettu konstruktori
   Public Kirjasto(int u_erapaiva, int u_tyyppi, string u_nimi)
   {
     erapaiva = u_erapaiva;
     tyyppi = u_tyyppi;
     nimi = u_nimi;
}
```

Voit nyt luoda ”Kirjasto” tyypin muuttujan ja antaa sille arvot (parametrit) seuraavasti:

u_nimi=”Aitta”;
Kirjasto arvot = new Kirjasto(30,1,u_nimi); //30 päivää lainausaikaa, tyyppi 1, nimi

Huomaa, että kaikki kolme parametriä on annettava, koska ne on määritelty oletuskonstruktorin määrittelyssä. Tämä olisi virhe:
Kirjasto arvot = new Kirjasto(27,1); //ei toimi tässä oletuskonstruktorissa

Jos kirjoitat konstruktorin, jossa on parametrejä, sinun on myös luotava oletuskonstruktori itse, sillä siinä tapauksessa kääntäjä ei itse luo oletuskonstruktoria.

6: Partial luokat

Luokasta tulee kaikkine konstruktoreineen helposti suuri. C#:ssa voidaan luokka jakaa osiin partial avainsanalla. Seuraavassa esimerkki:

```
//Kirjasto osa 1
partial class Kirjasto
{
  //Oletuskonstruktori
  Public Kirjasto()
  {
    erapaiva = 0; //oletetaan, että tässä on lainausaika, esim. 30 päivää.
                  //ei koko päiväystä, joka vaatisi useamman eri muuttujan
    tyyppi = 0;
    nimi = ””; //kaksi lainausmerkkiä alustaa merkkijonon arvoksi null
  }

   //Ylikuormitettu konstruktori
   Public Kirjasto(int u_erapaiva, int u_tyyppi, string u_nimi)
   {
     erapaiva = u_erapaiva;
     tyyppi = u_tyyppi;
     nimi = u_nimi;
}

//Kirjasto osa 2
partial class Kirjasto
{
  //kentät
  private int erapaiva;
  private int tyyppi;
  private string nimi;

  //metodit
  public int Erapaiva();
  public int Tyyppi();
  public string Nimi();
}
```

Tässä osa 1:ssä ovat konstruktorit, ja osa 2:ssa kentät ja metodit.

7: Ominaisuusfunktiot

Ominaisuusfunktioilla voidaan lukea ja kirjoittaa luokan yksityisten (private) jäsenten dataa. Ominaisuusfunktioita set ja get sanotaan myös settereiksi ja gettereiksi. set toiminnan avulla voidaan tallentaa yksityisen luokan kenttään arvo ja get toiminnon avulla voidaan lukea luokan ulkopuolelta yksityisen kentän arvo. value on avainsana, jossa on sijoitettava tai palautettava arvo.
Kolmion pinta-alan voi kyllä laskea paljon yksinkertaisemmallakin ohjelmalla, mutta tämä on harjoitusohjelma, jolla on helppo opetella olioiden käyttöä. Olioiden edut kasvavat, mitä suurenpia ohjelmia tehdään.

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

namespace SuorakulmainenKolmio
{
    class Program
    {
        class Kolmio
        {
            private double kanta, korkeus;

            public double Laske()
            {
                return kanta * korkeus / 2; //palauttaa kolmion pinta-alan
            } //huom. ei puolipistettä lopussa { merkin jälkeen

            public Kolmio() //oletuskonstruktori
            {
                kanta = 0;
                korkeus = 0;
            }

            public double Kanta
            {
                set
                {
                    kanta = value;
                }

                get
                {
                    return kanta;
                }
            }

            public double Korkeus
            {
                set
                {
                    korkeus = value;
                }
                get
                {
                    return korkeus;
                }
            }
        }
```

```
        static void Main(string[] args)
        {
            double u_kanta, u_korkeus;

            try
            {
                Console.Write("Anna kolmion kannan pituus: ");
                u_kanta = double.Parse(Console.ReadLine());
                Console.Write("\nAnna kolmion korkeus: ");
                u_korkeus = double.Parse(Console.ReadLine());

                Kolmio kolmio = new Kolmio();
                //siirretään arvot luokalle setterien avulla
                kolmio.Kanta = u_kanta; //aktivoidaan Kanta setterin arvo
                kolmio.Korkeus = u_korkeus; //aktivoidaan Korkeus setterin arvo

                //nyt kun arvot ovat luokalla, Laske() metodi osaa laskea
                //ja palauttaa tuloksen
                Console.Write("\nKolmion pinta-ala on: " + kolmio.Laske());
                //laitetaan luokan getteri palauttamaan kolmion korkeus
                Console.Write("\nKolmion korkeus on: " + kolmio.Korkeus);

            }
            catch
            {
                Console.Write("Virheellinen syöte.");
            }

             Console.Write("\n\nPaina jotain näppäintä jatkaaksesi...");
             Console.ReadKey(true);
        }
    }
}
```

8: Staattiset jäsenet ja luokat

Kun luokan metodi määritellään staattiseksi (static), siitä ei käytettäess luoda instanssia. Pääset metodiin käsiksi suoraan käyttämällä luokan nimeä. Seuraavassa esimerkkitapaus C#:in Math luokan metodista neliöjuuren laskennasta.

```
class Math
{
  public static double Sqrt(double d) { . . . }
}
```

Pystyt käyttämään metodia näin. Huomaa, että mitään new määritteitä ei tarvita.

```
double neliojuuri = Math.Sqrt(123.456); //Kutsu tapahtuu suoraan luokasta
```

8.1: Jaettu kenttä

Static määreellä voidaan myös luoda jaettuja kenttiä, joissa kaikki luokan oliot jakavat saman kentän. Seuraavassa esimerkki.

```
class Laskuri
{
 private string kavijanNimi;
 private int laskuri;

 public static int KavijaLaskuri=0;

 public Laskuri() //oletuskonstruktori
 {
   kavijanNimi="";
   KavijaLaskuri++;
  }

  public Laskuri(string nimi) //ylikuormitettu konstruktori
  {
   kavijanNimi=nimi;
   KavijaLaskuri++;
   }
}
```

Tässä KavijaLaskuria kasvatetaan aina, kun uusi instanssi luodaan. Staattiseen muuttujaan päästään käsiksi suoraan luokan nimellä, eikä tarvitse luodan objektia, oliota, eli esim. näin:

```
Console.WriteLine("Kävijöiden lukumäärä: " + Laskuri.KavijaLaskuri);
```

8.2: const avainsana

const avainsanalla kenttä voidaan määritellä staattiseksi, mutta sen arvoa ei pysty sen jälkeen muuttamaan, vaan se on vakio, esim. Math luokan Pi (pii) on tällainen, eli sen arvo on aina 3.14159...

C#:ssa myös luokka voidaan määritellä staattiseksi. Staattinen luokka voi sisältää vain static jäseniä, eikä static luokassa luoda mitään instansseja, eikä static luokkaa käytettäessä käytettä new avainsanaa.

9: Arvot ja viittaukset

int, float jne. tyyppejä sanotaan arvotyypeiksi. Jos määrittelet esim. int tyyppin, ohjelma varaa muistista neljä tavua (32 bittiä) muistia. Jos sijoitat int muuttujaan esim arvon kaksi, ohjelma varaa tuon neljä tavua muistia, vaikka sen pystyisi tallentamaan yhteenkin tavuun.
Luokka tyyppejä käytettäessä taasen, kun luokka määritellään ohjelma ei varaa koko luokalle tilaa, vaan ainoastaa tilan osoitteelle tai viitaukselle sinne muistialueelle minne luokka tallennetaan. Muisti oliolle varataan, kun käytät new avainsanaa.

Huomionarvoista on, että useimmat C#:in tyypit ovat primitiivi tyyppejä, paitsi että string on viittaustyyppi, joka toimii, kuten luokat. string on itse asiassa avainsana, ja tarkoittaa samaa kuin System.String luokka. Kun tässä opetellaan viitauksia, ne koskevat siis myös string tyyppiä.

Luokat siis käyttävät viittauksia. Kun käytät arvotyyppejä (primitiivejä tyyppejä), muisti toimii seraavasti:

```
int a=4;
int a2=a;
a=7;
```

Jos nyt tulostetaan a ja a2, niissä on eri arvo, koska a ja a2 ovat eri muuttujia, ja niitä säilytettään omassa eri muistilohkossa.

Sensijaan jos luodaan olio seuravasti esim. Kirjasto luokkaan:

```
Kirjasto erapaiva = new Kirjasto(30);
Kirjasto erapaiva_uusi = erapaiva;
erapaiva.Aseta = 60; //käytetään Aseta setteriä asettamiseen
```

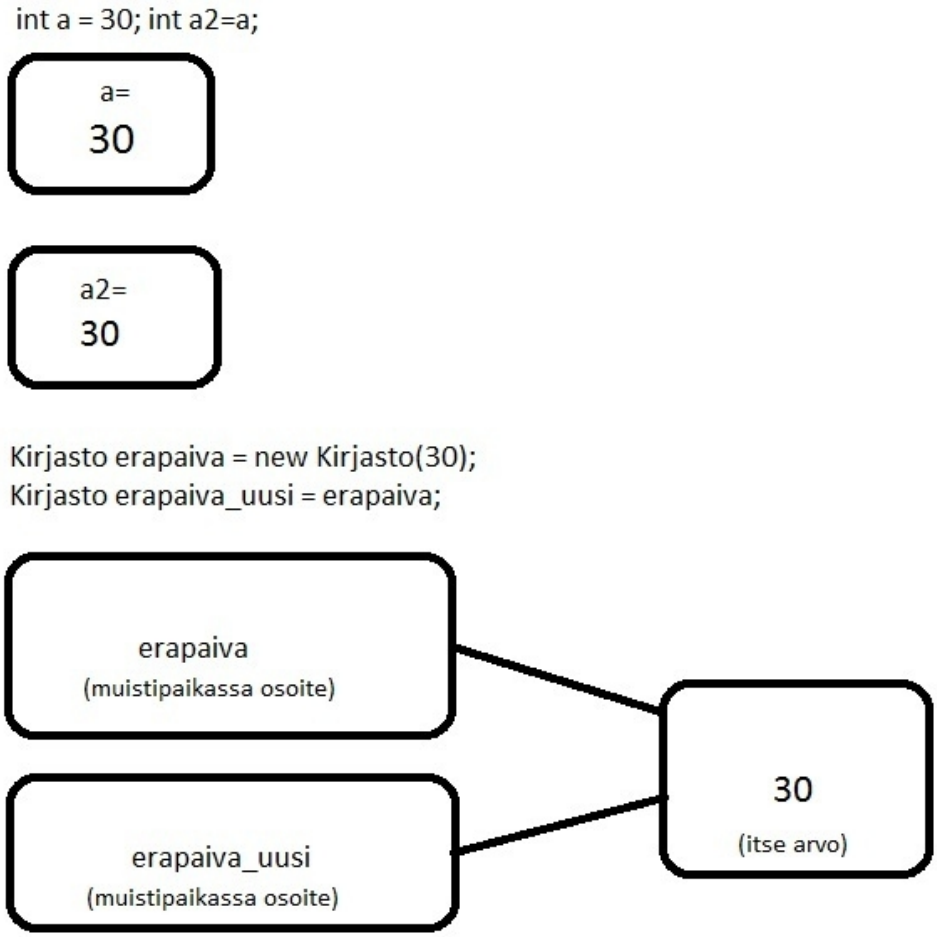
Nyt muisti varaa muistilohkon olioille erapaiva ja erapaiva_uusi niin, että onkin vain yksi muistipaikka, jossa on muuttuja, eli tässä tapauksessa 30. Sitten on kaksi osoitetta, eli erapaiva ja erapaiva_uusi, jotka viittaavat samaan muistipaikkaan. Kun nyt muutat olioon erapaiva 60, muutos näkyy myös erapaiva2 muuttujassa, koska

muuttujalla on vain yksi muistipaikka, minne data tallennetaan, ja kaksi osoitetta, mistä sinne viitataan.

Samoin, jos kirjoitat:
Kirjasto erapaiva = new Kirjasto(30);
Kirjasto erapaiva_uusi = erapaiva;
erapaiva_uusi.Aseta = 90; //käytetään Aseta setteriä asettamiseen

Nyt kummankin olion arvo on 90.

Seuraava kuva selventää asiaa.



Tämä ero on hyvin tärkeä, koska metodien käyttäytyminen riippuu siitä, ovatko ne arvotyyppejä vai viitaustyyppejä.

10: Periytyvyys (inheritance)

Periytymisen avulla voit luoda luokan, jolle luot periviä luokkia. Periytyvät luokat perivät yläluokalta joitain arvoja, ja sisältävät omia arvoja. Ne voivat myös ylikirjoittaa yläluokan arvoja omassa luokassaan.
Otetaan esimerkiksi kirjasto -luokka. ”Kirjasto” luokalla voisi olla alaluokka ”Teostiedot”. ”Kirjasto” luokassa on kaikille teoksille yhteisiä tietoja. Tällainen on esim. teoksen nimi. ”Teostiedot” luokalla on mm. eräpäivä (oletetaan, että tässä on lainattavia teoksia), kustantaja ja julkaisuvuosi. Tällaisen järjestelmän voisi toteuttaa tekemällä suuren joukon erillisiä luokkia. Olio-ohjelmoinnissa on kuitenkin mahdollisuus luoda yläluokka, jonka ominaisuuksia aliluokat perivät, ja voivat myös korvata niitä, kun sen sallii. Esim. alla myöhemmin tässä oppaassa käytetään esimerkkiä, jossa teoksen ”tyyppi” muuttujalla on oletusarvo, jonka voi ylikirjoittaa.

Voitaisiin tehdä seuraava luokkahierarkia:

```
class Kirjasto
{
  private nimi;

  public void Nimi()
  {
  }
}

class Teostiedot : Kirjasto
{
  private erapaiva;
  private kustantaja;
  private julkaisuvuosi;

  public void Erapaiva()
  {
  }

   public void Kustantaja()
  {
  }

  public void Julkaisuvuosi()
  {
  }
}
```

Tässä on ”Kirjasto” luokasta periytetty ”Teostiedot” luokka. ”Teostiedot” luokka perii ”Kirjasto” luokalta ”nimi” muuttujan, ja sillä on omia tietoja.

10.1: Perusluokan konstruktorin kutsuminen

Voit nyt tehdä ”Kirjasto” luokalle konstruktorin

```
class Kirjasto
{
  private nimi;

  public Nimi(string nimi)
  {
  }
}
```

Kun määrittelet aliluokan, on hyvä tehdä myös sille ”nimi” konstruktori, vaikka ohjelma toimii ilmankin, jos yläluokalla on public oletuskonstruktori. Esim. tässä ohjelmassa public oletuskonstruktoria ei ”kirjasto” luokalla ole. ” ”Teostiedot” luokan konstruktori ”Nimi” tehdään seuraavasti:

```
class Teostiedot : Kirjasto
{
  public Nimi(string nimi) : base(nimi) //kutsuu Kirjasto (nimi)
  {
  }
}
```

Voit nyt periyttää ”Teostiedot” luokasta eri luokkia.

```
class Kirja : Teostiedot
{
}

class Lehti : Teostiedot
{
}

class Aanite : Teostiedot
{
}

class Muu : Teostiedot
{
}
```

Voit tietenkin periyttää ”Kirjasto” luokasta myös muita luokkia, jos ajatellaan, että ”Teostiedot” luokassa on lainattavat teokset, voisit luoda vaikka käsikirjasto luokan ”Kasikirjat”.

```
class Kasikirjat : Kirjasto
{
}
```

Ja periyttää siitä esim. sanakirjat, tietosanakirjat jne.

Pääohjelmassa esim. ”Teostiedot” luokan olio luodaan esim. seuraavasti:
Teostiedot uusiTeos = new Teostiedot(”kirjan_nimi”);

Vastaavasti ”Lehti” luokan olio luodaan seuraavasti:
Lehti uusiLehti = new Lehti(”lehden_nimi”);
Jos luokka sijaitsee yläpuolella, myös siihen voidaan viitata, eli seuraava on oikein:
Lehti uusiLehti = new Lehti(”lehden_nimi”);
Teostiedot teos = uusiLehti;

Sensijaan tämä olisi virhe:
Lehti uusiLehti = new Lehti(”lehden_nimi”);
Kirja uusiKirja = uusiLehti; //Virhe! Samalle tasolle tai alaspäin ei eri luokkaan voi
//viitata

10.2: Metodin ylikirjoittaminen (virtual ja override)

virtual avainsanalla voidaan metodi määritellä ylikirjoitettavaksi, ja override avainsanalla toteutetaan ylikirjoittaminen. Esim. ”Teostiedot” luokassa ja sen aliluokissa. Tässä muutetaan luotua ”tyyppi_nimi” tietoa.

```
class Teostiedot : Kirjasto
{
  private string tyyppi_nimi;

  public virtual string TyyppiNimi()
  {
    tyyppi_nimi = ”Määrittelemätön teos”
  }
}
```

```
class Kirja : Teostiedot
{
  public override string TyyppiNimi()
   {
    tyyppi_nimi= ”Kirja”;
   }
}

class Lehti : Teostiedot
{
  public override string TyyppiNimi()
   {
    tyyppi_nimi = ”Lehti”;
   }
}

class Aanite : Teostiedot
{
  public override string Tyyppi()
   {
    tyyppi_nimi = ”Äänite”;
   }
}

class Muu : Teostiedot
{
}
```

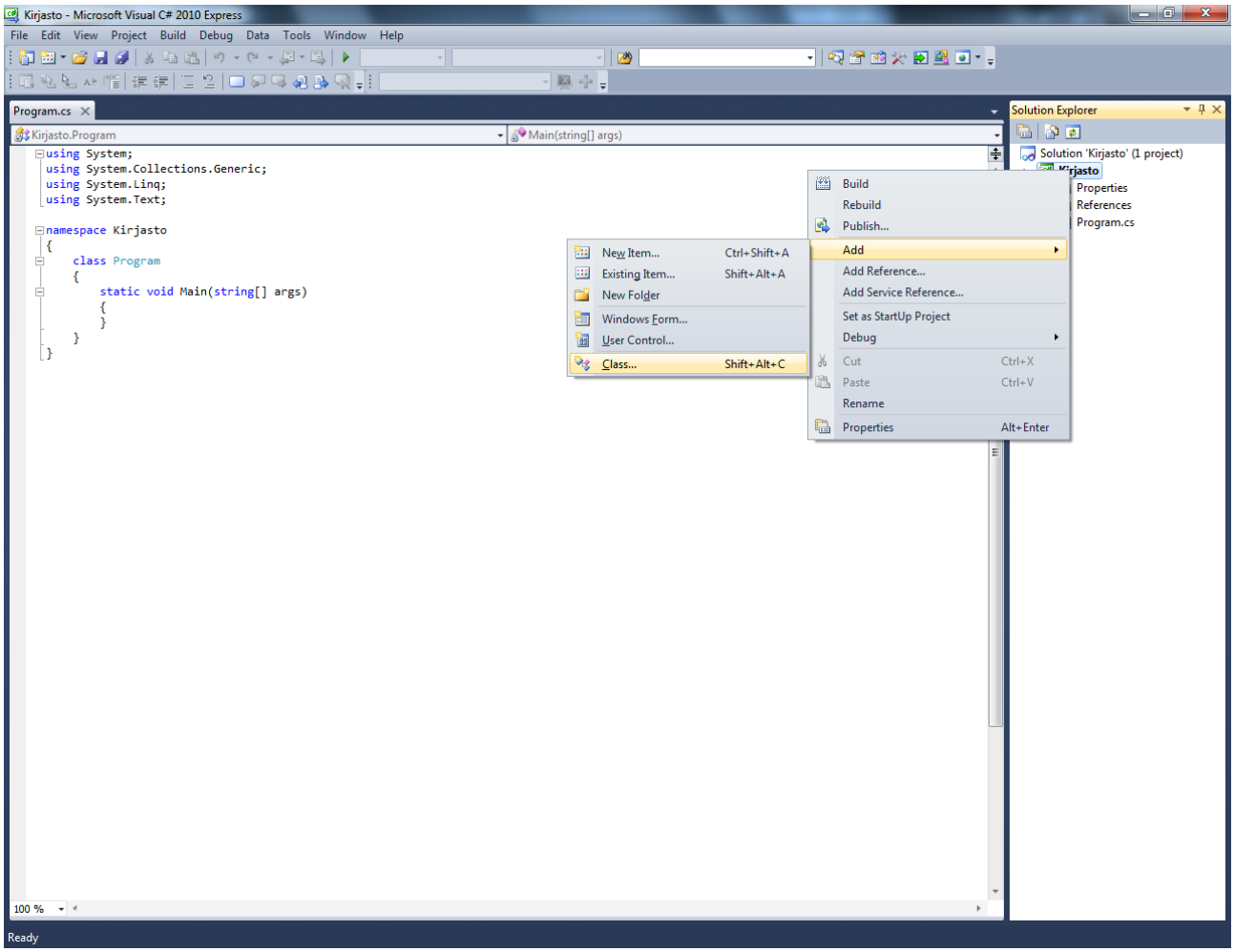
”Muu” luokassa ei ole ylikirjoitettu ”TyyppiNimi” metodia, joten se tulee ”Teostiedot” luokasta arvoksi ”Määrittelemätön teos”. Muissa aliluokissa ”tyyppi_nimi” ylikirjoitetaan, eli esim. ”Aanite” luokassa ”tyyppi_nimi” muuttuja saa arvon ”Äänite”.

11: Tietokoneen muistin organisoituminen (stack ja heap)

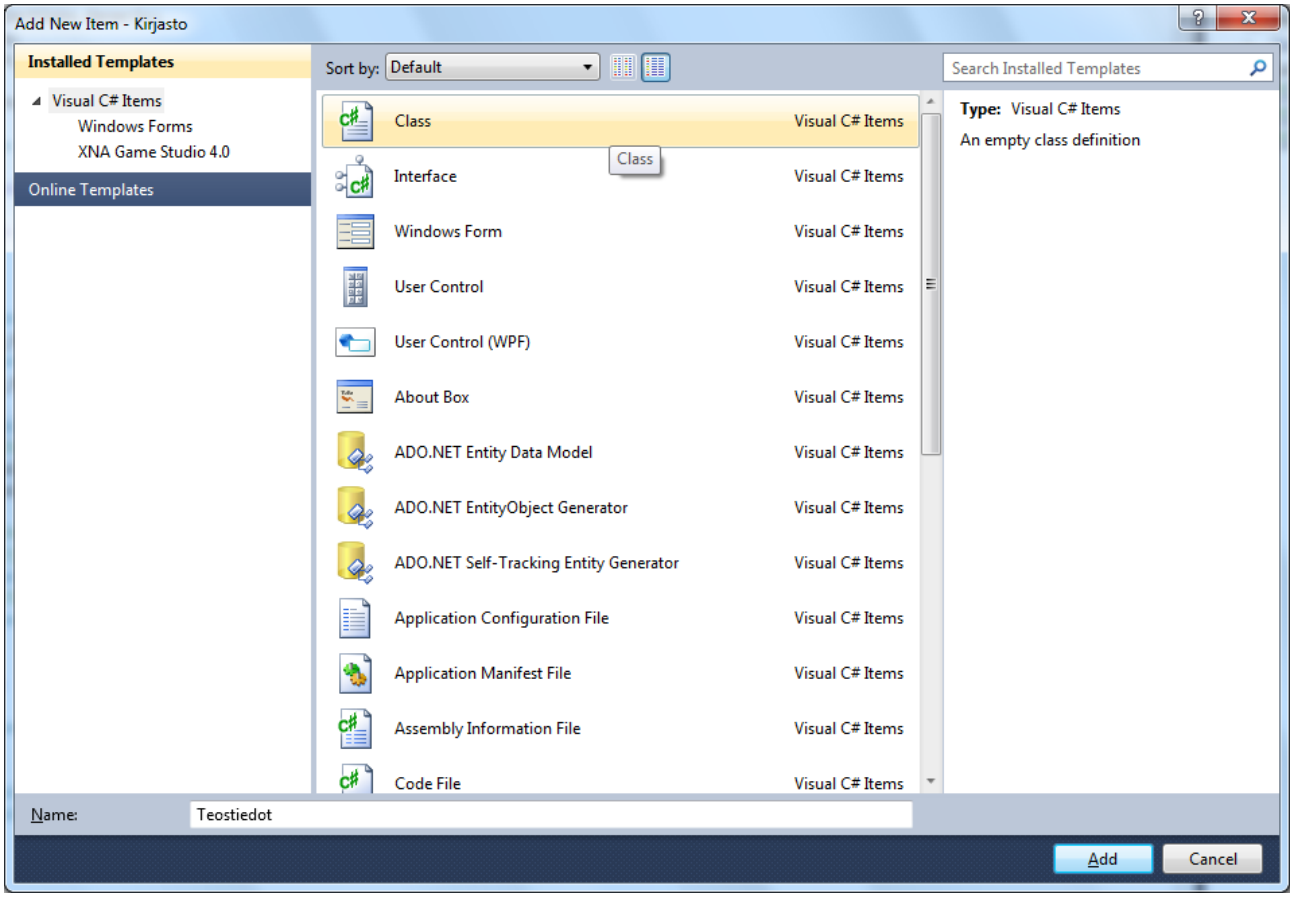
Kun metodia kutsutaan, sen parametrien tarvitsema muisti varataan stack muistista. Kaikki arvomuuttujat varataan stackistä. Stack on ikäänkuin pino, jonne laitetaan päällekäin muuttujia, ja poimitaan sieltä. Kun metodin suoritus loppuu, vapautetaan stack muistin tila.
Olion (jotka varataan new avainsanalla) ja kaikkien viitausmuuttujien tarvitsema muisti jonne viitataan, varataan heapista. Osoitteet, jotka osoittavat heapiin varataan stackistä. Yhteen heap muistin paikkaan voi osoittaa useasta eri stacikin osoitteesta. Luvussa 9 oleva kaavio selventää tätä viitausmuuttujista. Heap on tavallaan hyvin hajallaan muistiavaruudessa oleva laatikkopinojen kokoelma, ja sen varaama tila vapautetaan joskus, kun viimeinen osoitus siihen poistuu. Heapia ei siis automaattisesti vapauteta, kun sitä ei enään tarvita, kuten stackille tehdään.

12: Uuden luokkaikkunan luonti

Kun luokkia on samassa nimiavaruudessa (namespace) paljon, on helpompi, jos luodaan eri luokille oma ikkunansa. Se tapahtuu Solution Explorerissa klikkaamalla luokan nimeä hiiren oikealla näppäimellä, ja valitsemalla ”Add” ja sitten ”Class”. Suraavassa kuvassa kuva näytöstä.



Avautuu ikkuna, johon alle kirjoitetaan aliluokan nimi, esim. ”Kirjasto” luokassa ”Teostiedot”. Tuplaklikkaa ”Class” toimintoa, niin saat uuden luokan luotua.



”Teostiedot” luokka aukeaa näkyviin seuraavasti:

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

namespace Kirjasto
{
    class Teostiedot
    {
    }
}
```

Kuten huomaat, namespacena on ”Kirjasto” jonka perusteella kääntäjä osaa linkittää ”Teostiedot” luokan samaan projektiin ilman sen kummempia lisäkoodeja.

Esim. aiemmin ollut ”Kolmio” ohjelma näyttäisi tältä, kun se jaetaan eri ikkunoihin.

Pääohjelma ”Program.cs”:

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

namespace SuorakulmainenKolmio
{
    class Program
    {
        static void Main(string[] args)
        {
           double u_kanta, u_korkeus;

           try
           {
               Console.Write("Anna kolmion kannan pituus: ");
               u_kanta = double.Parse(Console.ReadLine());
               Console.Write("\nAnna kolmion korkeus: ");
               u_korkeus = double.Parse(Console.ReadLine());

               Kolmio kolmio = new Kolmio();
               //siirretään arvot luokalle setterien avulla
               kolmio.Kanta = u_kanta; //aktivoidaan Kanta setterin arvo
               kolmio.Korkeus = u_korkeus; //aktivoidaan Korkeus setterin arvo

               //nyt kun arvot ovat luokalla, Laske() metodi osaa laskea
               //ja palauttaa tuloksen
               Console.Write("\nKolmion pinta-ala on: " + kolmio.Laske());
               //laitetaan luokan getteri palauttamaan kolmion korkeus
               Console.Write("\nKolmion korkeus on: " + kolmio.Korkeus);

           }
           catch
           {
               Console.Write("Virheellinen syöte.");
           }

            Console.Write("\n\nPaina jotain näppäintä jatkaaksesi...");
            Console.ReadKey(true);
        }

    }
}
```

Ja luokka ”Kolmio”:

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

namespace SuorakulmainenKolmio
{
    class Kolmio
    {
        private double kanta, korkeus;

        public double Laske()
        {
            return kanta * korkeus / 2; //palauttaa kolmion pinta-alan
        } //huom. ei puolipistettä lopussa { merkin jälkeen

        public Kolmio() //oletuskonstruktori
        {
            kanta = 0;
            korkeus = 0;
        }

        public double Kanta
        {
            set
            {
                kanta = value;
            }

            get
            {
                return kanta;
            }
        }

        public double Korkeus
        {
            set
            {
                korkeus = value;
            }
            get
            {
                return korkeus;
            }
        }
    }
}
```

Tällainen ohjelman jakaminen eri ikkunoihin on välttämätöntä, kun ohjelma kasvaa suureksi, ja on erilaisia periytyviä, ym. luokkia. Vaikka ohjelma sinällään toimii, vaikka kaikki luokat kirjoittaisi ”Program.cs” ikkunaan, ohjelman hallinnan takia vähänkin pidemmät ohjelmat kannattaa jakaa eri ikkunoihin.

Lähteet: Oulun Seudun Ammattiopisto (OSAO), peliohjelmointi verkkokurssi