C# ja .NET Framework

Olio-ohjelmointi

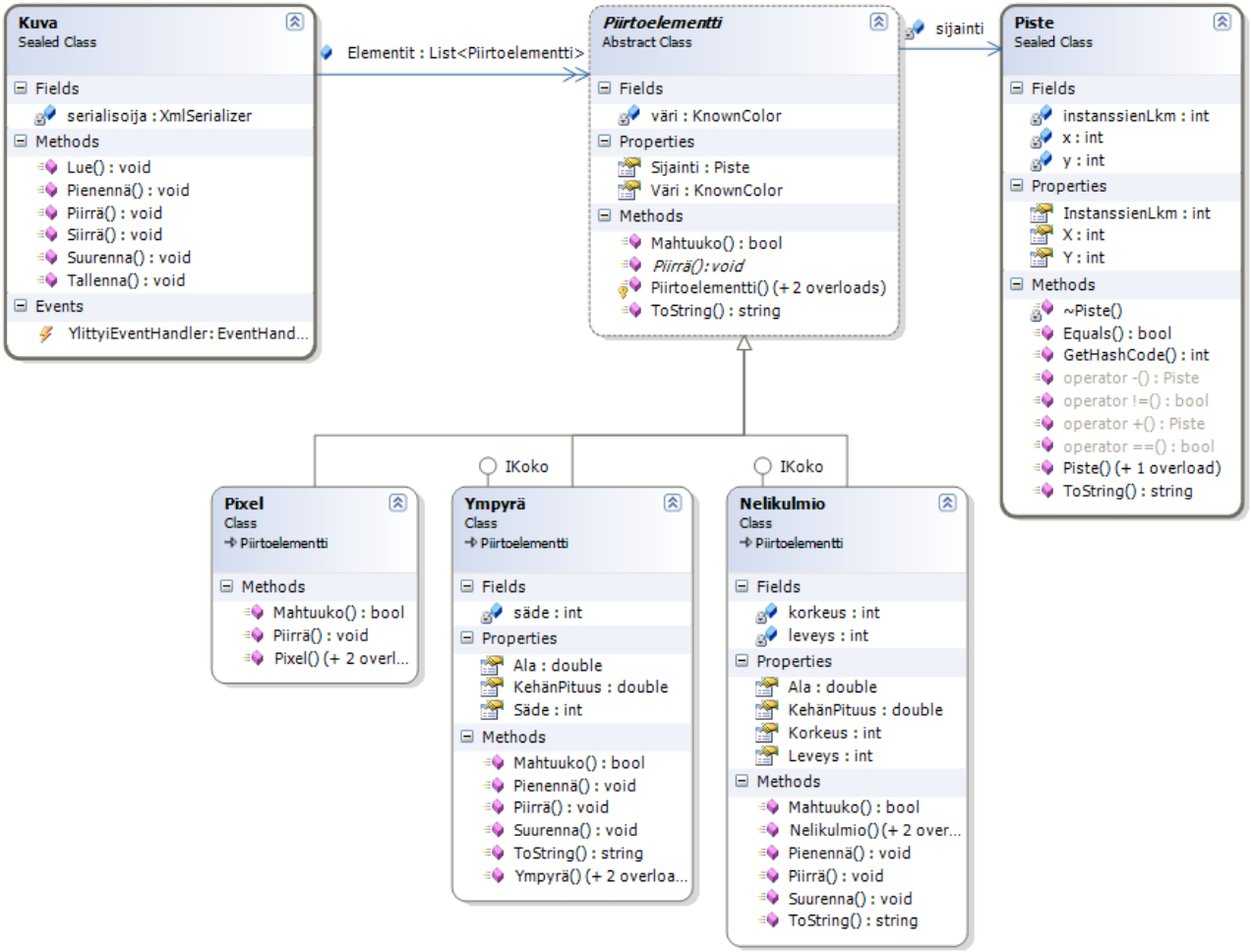
Sisällys

Harjoitus 1: **Harjoitusprojektin tekeminen**

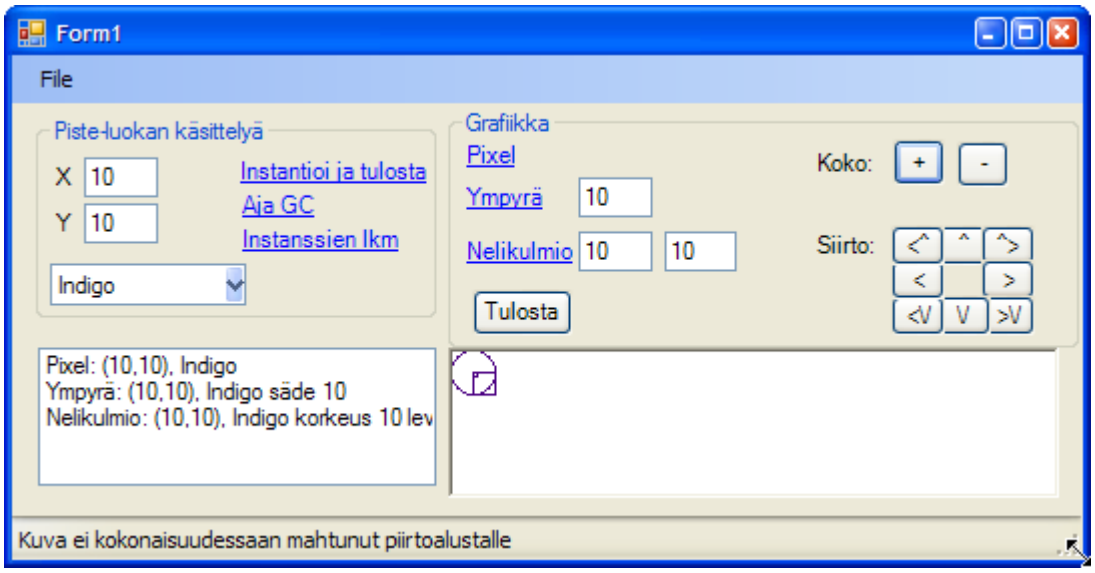
Tausta

Seuraavissa harjoituksissa käydään läpi yksityiskohtaisesti C# olio-ohjelmointi. Harjoitusten aiheena on tuottaa graafisia palveluita tarjoava `PiirtoAvut.dll` komponentti. Harjoitustehtäväaihe on valittu siksi, että käsiteltävät luokat ovat suhteellisen helposti konkretisoitavissa niin käsitteinä kuin käyttöliittymässäkin. Toisaalta voidaan myös todeta, että harjoitusten oliot ovat niin yksinkertaisia, että vastaavat palvelut ovat saatavilla huomattavasti laajempina suoraan `System.Drawing` palveluista.

Harjoitussarjassa tullaan tekemään seuraavan luokkakaavion mukainen luokkahierarkia sekä testeri-sovellus, jossa näitä luokkia tullaan käyttämään.



Harjoituksen lopussa luokkarakenne on tällainen.



Sovelluksen testerilomake tulee olemaan tällainen.

Tässä ensimmäisessä harjoituksessa käsitellään

- *Solution- ja projektirakenne*
- *referointi*

Tehtävä

Tee seuraavankaltainen solution- ja projektirakenne.

- Solution: `Olioharjoitus` hakemistoon `C:\user\Olioharjoitus`

jonne sijoitat projektit:

- Class Library: `PiirtoAvut` hakemistoon `C:\user\olioharjoitus\PiirtoAvut`
- Windows Application: `Tester` hakemistoon `c:\user\Olioharjoitus\Tester`

Tester-projekti referoi Piirtoavut. Käännä ja testaa sovellus, vielä ei ole siis mitään toiminnallisuutta.

Toimenpiteet

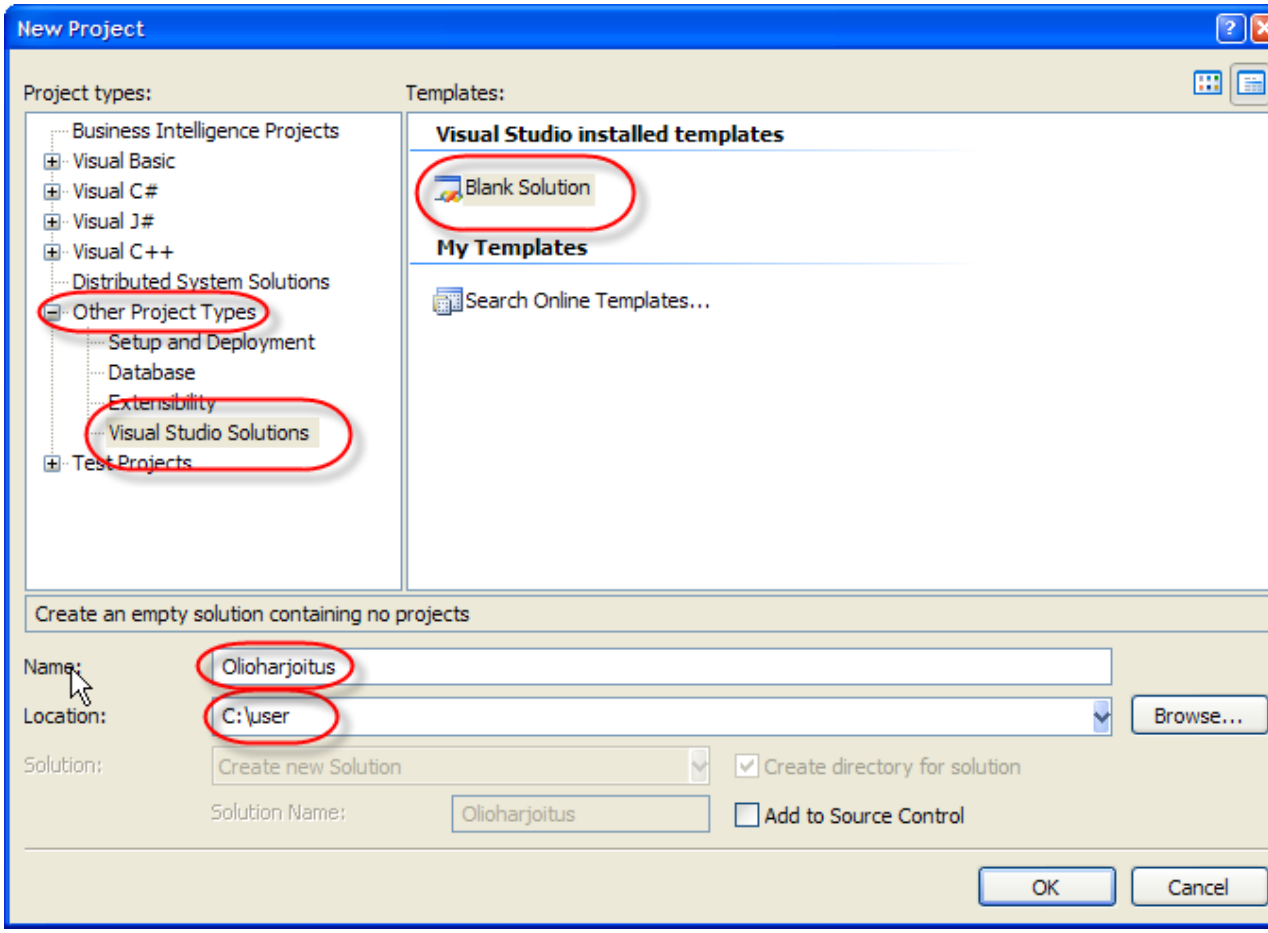
1. Tee uusi ’Blank solution’ valikkokomennolla **File|New Project**. Valitse/aseta

   1.1. Project types: Other Project Types -> Visual Studio Solutions

1.2. Templates: Blank Solution

1.3. Name: Olioharjoitus

1.4. Location: C:\User



2. Lisää solutioniin `Piirtoavut`-projekti valikkokomennolla: **File | Add | New Project**, ja valitse

   2.1. Project types: Visual C# -> Windows

   2.2. Templates: Class Library

   2.3. Name: Piirtoavut

   2.4. Location: `C:\User\Olioharjoitus` (on oletuksena oikein)

3. Poista `Piirtoavut`-projektissa oleva `Class1.cs` –tiedosto. Tämä tehdään painamalla hiiren oikeaa Solution Explorer-ikkunassa tiedostonimen päällä ja valitesemalla **Delete**.

4. Lisää solutioniin `Tester`-projekti valikkokomennolla: **File | Add | New Project**, ja valitse

    4.1. Project types: Visual C# -> Windows

    4.2. Templates: Windows Application

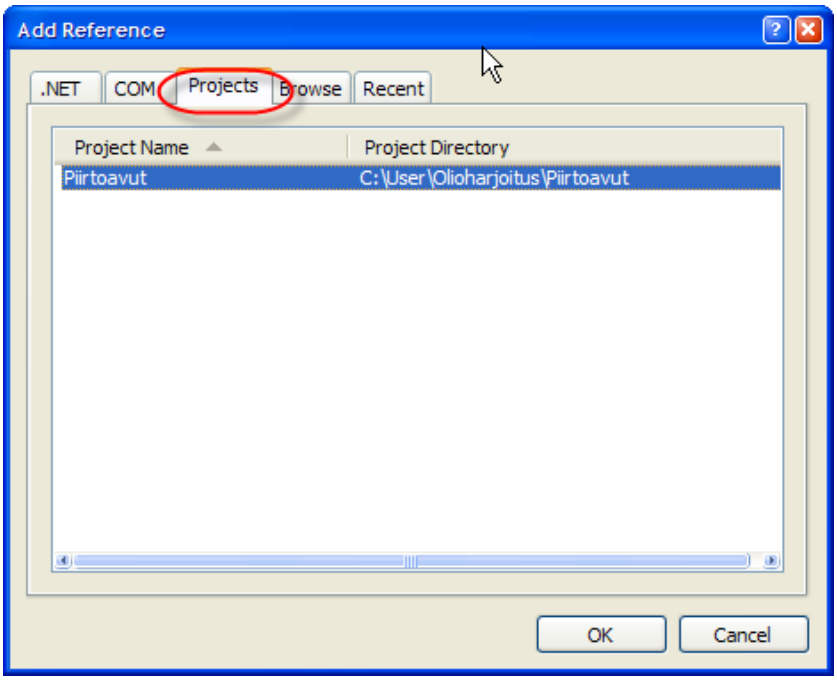
    4.3. Name: Tester

    4.4. Location: `C:\User\Olioharjoitus` (on oletuksena oikein)

5. Referoi Tester-projektiin Piirtoavut.

    5.1. Paina hiiren oikeaa Solution Explorerissa Tester|References –kohdassa, ja valitse **Add Reference…**

    5.2. Valitse Projects-välilehdeltä Piirtoavut –projekti.



6. Käännä solution (Shift + Ctrl + B).

7. Tarkista, että Tester-projekti on StartUp projektina (näkyy vahvennettuna projektinimenä Solutions-ikkunassa). Ellei ole, niin hiiren oikeaa Solution Explorer-ikkunassa projetinimen päällä **| Set as StartUp Projcet**.

8. Aja sovellus, nyt on vielä pelkkä tyhjä lomake.

Mallivastaus on VSS:n projekti **Olioharjoitus_01**

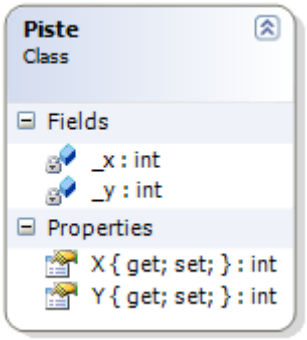
Harjoitus 2: **Luokan tekeminen ja käyttäminen**

Tausta

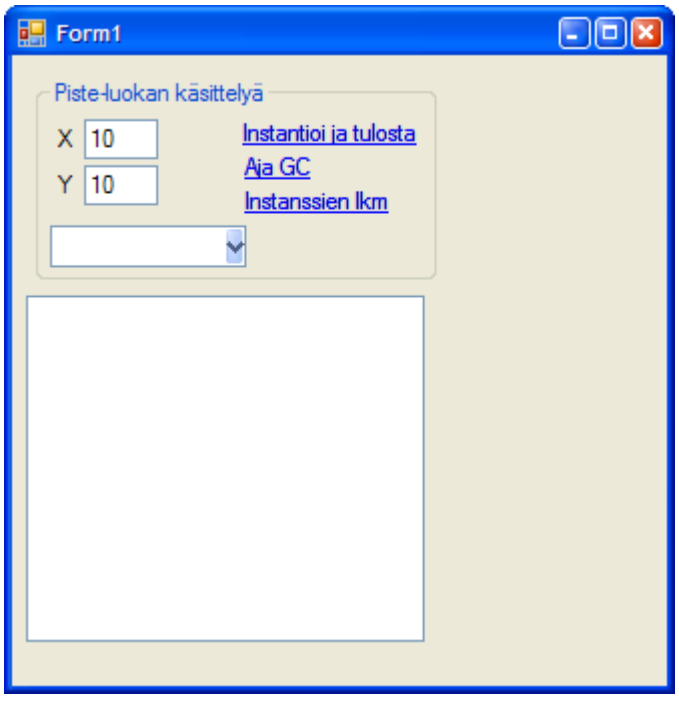
Tässä harjoituksessa tehdään luokka ja testataan sitä testerissä.

Tehtävä

Tee `PiirtoAvut`-projektiin luokka `Piste`. Luokan `Piste` tarkoituksena on määritellä jokin kohta piirtokoordinaatistossa. Sillä on private kentät (field) `int _x, _y` sekä ominaisuudet (property) `X` ja `Y` (tyyppiä `int`), joilla kyseiset kentät voidaan asettaa ja kysyä (= kenttien aksessorit). Luokalla ei ole alkuvaiheessa itse tehtyä konstruktoria. Luokkakaavio on:



Tee Testeriin toiminnat, jolla voi testata Piste-luokan toimintoja.



Toimenpiteet, Piirtoavut

1. Lisää Piirtoavut-projektiin luokka `Piste.` Tämä tehdään painamalla hiiren oikeaa Piirtoavut-projektirivin päällä Solution Explorerissa ja valitsemalla **Add | Add Class…** Anna nimeksi: `Piste`

2. Muuta luokka julkiseksi luokaksi (=lisää public määre)

```
public class Piste {
}
```

3. Lisää luokkaan kentät x ja y

```
private int _x;
private int _y;
```

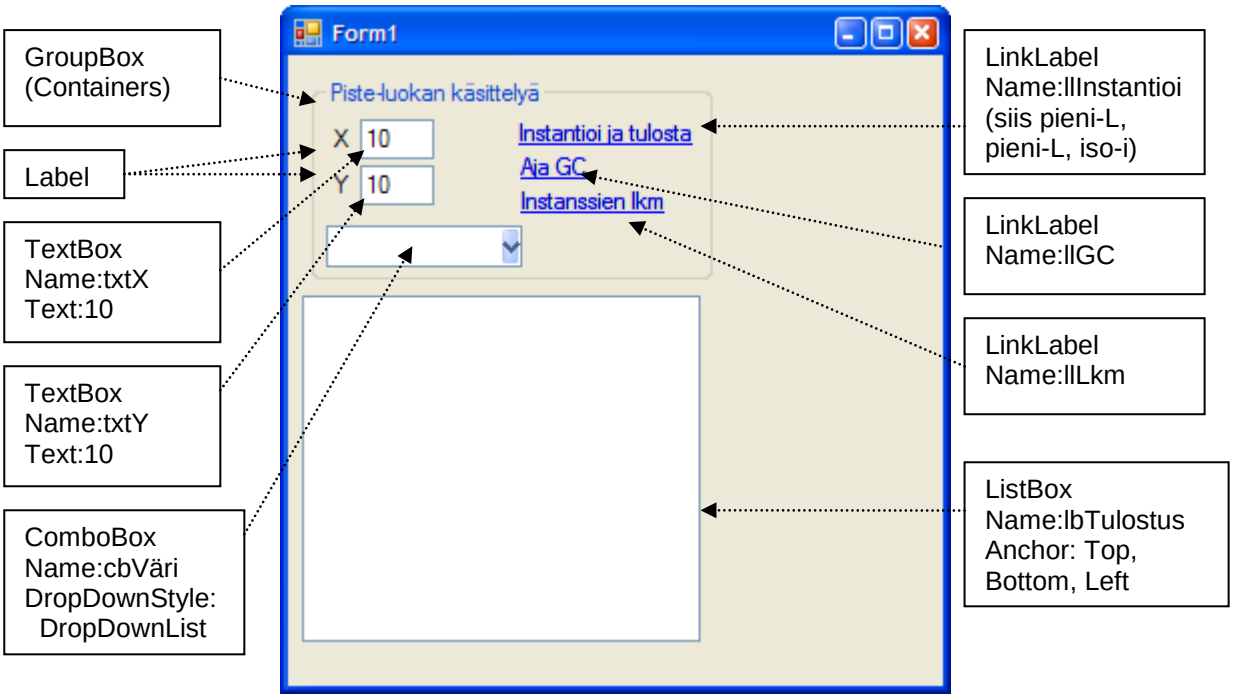
4. Tee ominaisuudet (properties) X ja Y. Tehdään näin ensimmäisellä kerralla koodaten – ilman velhojen apua. Ohessa X-propertyn koodi, vastaava on siis tehtävä myös Y:lle.

```
public int X {
  get { return _x; }
  set { _x = value; }
}
```

5. Käännä projekti.

Toimenpiteet, Tester

6. Maalaa Tester-projektin lomakkeelle Piste-luokan käsittelyssä tarvittavat kontrollit (ja samalla jo muutaman seuraavankin harjoituksen kontrollit). Ohessa kontrollit ja niille tehtävät property-asetukset:



7. Tee ”*Instantioi ja tulosta*” linklabeliin tapahtumakäsittelijä (=tuplaklikkaa kontrollia, jolloin tehdään tapahtumakäsittelijä oletustapahtumaan Click). Koodaa Piste-olion luonti ja tulostus `lbTulostus`-kontrolliin.

```
Piirtoavut.Piste p = new Piirtoavut.Piste();
p.X = int.Parse(txtX.Text);
p.Y = int.Parse(txtY.Text);
lbTulostus.Items.Add(p.ToString());
```

8. Testaa sovellus. Painonapin painalluksessa tulostuu Piste-luokan nimi (koko nimi) listaan. Minkä luokan ToString-metodi nyt suoritetaankaan?
   Vast: _____
   (Object-luokan, koska Piste-luokalla ei ole tätä metodia korvattu).

Mallivastaus on VSS:n projekti **Olioharjoitus_02**

Harjoitus 3: **ToString –metodin korvaaminen**

Tausta

Jokaiselle luokalle on syytä toteuttaa `ToString()`-metodi. Metodia tarvitaan vähintäänkin debuggaamisessa, ja sen avulla oliosta voidaan saada myös käyttöliittymässä käytettävä merkkijono.

Jos kentästä on tehty ominaisuus (property), on sitä syytä käyttää myös luokan sisällä, ei ainoastaan luokan ulkopuolelta.

Tehtävä

Toteuta Piste-luokkaan `ToString()` –metodi, joka tulostaa pisteen muodossa `(x,y)`, esimerkiksi (10,10).

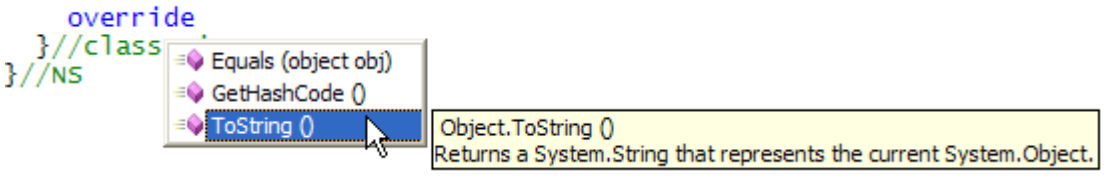
Toimenpiteet

1. Lisää Piste-luokkaan `ToString()`-metodi. Tässä siis korvataan (override) kantaluokasta `Object` peritty `ToString()`-metodi. Korvaaminen on syytä aina tehdä wizardin avulla seuraavasti

   1.1. Avaa `Piirtoavut` – projektin `Piste.cs` –tiedosto (tuplaklikkaa `Piste.cs` –riviä Solution Explorerissa)

   1.2. Vie kohdistin Piste-luokan ”sisälle”, esimerkiksi juuri ennen luokan sulkevaa aaltosulkua.

   1.3. Kirjoita `override` ja välilyönti. Valitse override-listasta `ToString`.



   Koodaa toiminta:

```
return string.Format("({0},{1})", X,Y);
```

2. Testaa sovellus. Nyt pitäisi tulostua pisteen koordinaatit "kauniisti", esim `(10,10)`.

   Mallivastaus on VSS:n projekti **Olioharjoitus_03**

Harjoitus 4: **Konstruktori**

Tausta

Jokaiselle luokalle on syytä tehdä konstruktorit. Yleensä konstruktoreita on useita, ne siis ylikuormitetaan (overload). Tämä tehdään siksi, että oliot olisivat mahdollisimman helppo instantioida, mutta toisaalta ovat heti instantioinnin jälkeen toimivia.

Mikäli konstruktoria ei ole lainkaan, tekee kääntäjä ns. oletuskonstruktorin, eli parametrittoman konstruktorin. Oletuskonstruktori jättää kaikki kentät tyhjiksi (=0, null, false). Tätä konstruktoria ei synny, mikäli luokassa on yksinkin konstruktori.

Konstruktorit toteutetaan usein delegointitavalla, eli koodi kirjoitetaan vain 'täydelliseen' konstruktoriin(se, joka saa kaikki parametrit). Muut overloadatut versiot kutsuvat tätä konstruktoria täydentäen puuttuvat parametrit oletusarvoilla.

Tehtävä

Tee Piste-luokkaan kaksi konstruktoria: sellainen, joka saa x- ja y-parametrit (käytetään tässä nimeä integer-konstruktori) sekä toinen, joka alustaa pisteen origoksi (0,0). Tämä 'origo'-konstruktori delegoi konstruktoinnin 'integer-konstruktorille'.

Toimenpiteet

1. Tehdään ensin parametreja saava "integer-konstruktori". Koodaa Piste-luokkaan seuraava konstruktori. Huomaa, että

   1.1. voit käyttää intellisenseä `public` ja `Piste`-sanojen täydentämiseen. Intellisenseä on syytä opetella tietoisesti käyttämään.

   1.2. Myös konstruktori käyttää `X` ja `Y` propertyjä kenttien asettamiseen, ei siis suoraan aseta arvoja `_x` ja `_y` –kenttiin.

   1.3. Selvyyden vuoksi käytetään `this.` viittausta. Konstuktorikoodissa on usein kolme lähes samannimistä muuttujaa, eli 1) parametri (tässä: `int x`), 2) kenttä (tässä: `int _x`) ja 3) property (tässä `int X`). Usein kenttä ja parametri ovat täsmälleen samannimisiä, silloin `this.` viittausta on käytettävä.

```
//'integer'konstruktori
public Piste(int x, int y) {
  this.X = x;
  this.Y = y; }
```

2. Käännä. Nyt tulee käännösvirhe Piste-olion konstruktoinnista (`Form1`-luokasta). Mieti ja kirjoita oheen, miksi tämä virhe tuli (mieti jonkin aikaa, vasta sitten katso vastaus):

   Vastaus: _____

   _____

   Vastaus: virhe tulee siksi, että nyt Piste-luokaan tehtiin konstruktori (joka saa kaksi integer-parametria). Kun luokassa on konstruktori, kääntäjä ei enää generoi oletuskonstruktoria. Siksi Piste-oliota ei enää voi instantioida antamatta x- ja y-parametreja.

3. Korjataan testerin koodi siten, että piste instantioidaan käyttäen lomakkeella annettuja X- ja Y-arvoja.

```
Piirtoavut.Piste p = new Piirtoavut.Piste(
                                    int.Parse(txtX.Text),
                                    int.Parse(txtY.Text));
lbTulostus.Items.Add(p.ToString());
```

4. Käännä ja testaa sovellus.

5. Toteutetaan "origo"-konstruktori delegoimalla konstruktointi "integer"-konstruktorille parameterilla 0,0.

```
//'origo'konstruktori
public Piste():this(0,0) {
}
```

6. Lisää testeriin toisenkin pisteen instantiointi (samaan tapahtumakäsittelijään). Piste p2 instantioidaan käyttäen "origo" –konstruktoria.

```
//instantioidaan toinen piste käyttäen
//parametritonta konstuktoria
Piirtoavut.Piste p2 = new Piirtoavut.Piste();
lbTulostus.Items.Add(p2.ToString());
```

7. Käännä ja testaa.

   Mallivastaus on VSS:n projekti **Olioharjoitus_04**

Harjoitus 5: **Olion elinkaari ja Trace-diagnostiikka**

Tausta

Sovellus voi tulostaa `System.Diagnostic.Trace`–luokan metodeilla kehittimen (tai vastaavan) debug-ikkunaan. Mikäli sovellus käännetään tuotantoversioksi, voidaan nämä komennot jättää pois.

.NET-luokkiin tehdään harvoin destruktori-metodia, koska sen kutsunta-ajankohta (ja järjestys) on satunnainen. Sellainen voidaan kuitenkin tehdä, ja destruktori on nimeltään `~Luokka()`. Destruktori-tarpeeseen ja varsinaiseen toteuttamiseen tutustutaan resursseja varaavan luokan yhteydessä. Nyt tutustutaan pelkkään destruktori-syntaksiin diagnostiikkamielessä.

Tehtävä

Tee `Piste`-luokalle destruktori. Lisää konstruktoriin ja destruktoriin `Trace.WriteLine` -tulostus, jotta nähdään kun niitä kutsutaan. Käytä tulosteessa myös `Object`-luokan `GetHashCode` -metodia olion identiteetin toteamiseksi.

Toteuta testerin ”*Aja GC*” –painonappi, jossa kutsutaan `GC.Collect()`. Tämän avulla voidaan tutkia destruktorien toiminta.

Toimenpiteet

1. Lisätään ensin konstruktoriin debug-tulostus. ”integer”-konstruktorin loppuun lisää seuraavat rivit:

```
string id = this.GetHashCode().ToString();
System.Diagnostics.Trace.WriteLine(".CTOR "
  + this.ToString(), id);
```

2. Lisää Piste-luokkaan desktruktori, jonka koodi on:

```
~Piste() {
  string id = this.GetHashCode().ToString();
  System.Diagnostics.Trace.WriteLine(".DTOR "
   + this.ToString(), id);
}
```

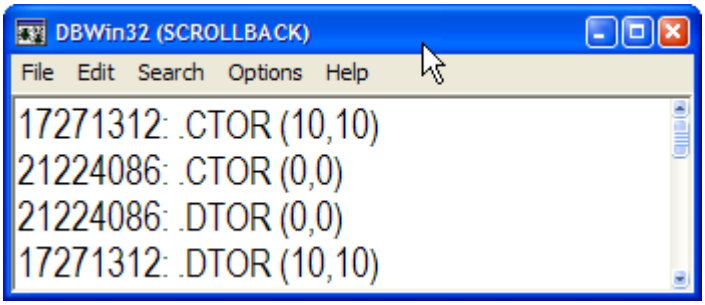
3. Lisää Testerin ”*Aja GC*”-painallukseen GC:n suoritus.

   3.1. Tuplaklikkaa lomakkeella linkLabelia, jolloin muodostetaan tapahtumakäsittelijämetodi

3.2. Koodaa toiminnallisuus

```
GC.Collect();
```

4. Käännä ja testaa sovellus. Debug-tulostukset menevät Visual Studion Output-ikkunaan mikäli ajat sovellusta debug-moodissa. Vaihtoehtoisesti voit käynnistää opettajan mainitsemasta paikasta `DBWin32.exe` -sovelluksen (ennen debug-session aloittamista). Debug-tulostukset tulevat tähän ikkunaan.



Mallivastaus on VSS:n projekti **Olioharjoitus_05**

Harjoitus 6: **Luokan yhteiset (static) jäsenet**

Tausta

Luokan static–jäsenet (kentät, ominaisuudet, metodit, indekserit) ovat luokan kaikille ilmentymille yhteisiä ja niitä kutsutaan suoraan luokan avulla. Static jäsenet voivat hyödyntää vain luokan muita static – jäseniä.

Tehtävä

Toteuta `Piste`-luokkaan staattinen `InstansienLkm`-laskuri. Laskurin arvoa kasvatetaan aina instantioinnin yhteydessä ja pienennetään destruktorissa. Laskuri on staattinen private-kenttä, ja sillä on julkinen get-aksessori.

Lisää testeriin laskurin kysyntä.

Toimenpiteet

1. Tee `Piste`-luokkaan laskuri-kenttä.

```
static private int instanssiLkm;
```

2. Ja sille julkinen get-ja private set- property

```
static public int InstanssiLkm {
  get { return Piste.instanssiLkm; }
  private set { Piste.instanssiLkm = value; }
}
```

3. Pisteen konstruktorissa (”integer-konstruktori”) kasvata instanssilaskuria

```
Piste.InstanssiLkm++;
```

4. Ja destruktorissa vähennä laskuria

```
Piste.InstanssiLkm--;
```

5. Lisää testerin ”*Instanssien lkm*” –linkLabelin painallukseen laskurin tulostus. Kokeile kirjoittaa tämä itse, vasta sitten katso esimerkkikoodia. Jookos:)

```
lbTulostus.Items.Add("Piste-instansseja on " +
                     Piirtoavut.Piste.InstanssiLkm + " kpl");
```

6. Testaa sovellus.

   Mallivastaus on VSS:n projekti **Olioharjoitus_06**

Harjoitus 7: **Perintä, luokat Piirtoelementti ja Pixel**

Tausta

.NETissä luokka periytyy vain yhdestä kantaluokasta (siis moniperintää ei ole, mutta rajapintoja voidaan kuitenkin implementoida useita). Perintä on tyypillinen yleistämiseen tai erikoistamiseen liittyvä ohjelmointitekniikka. Luokkien välistä yhteyttä kuvataan usein "on eräänlainen"–määreellä.

Seuraavissa harjoituksissa tehdään varsinaisia piirtoelementtejä sovellusten käytettäväksi. Tarkoituksena on tehdä luokat `Pixel`, `Ympyrä` ja `Nelikulmio`. Koska luokilla on kuitenkin paljon yhteistä toiminnallisuutta, kerätään ne yleisempään `PiirtoElementti`-luokkaan, josta sitten ko. luokat periytetään. Aluksi toteutetaan vain luokat `Piirtoelementti` ja `Pixel`.

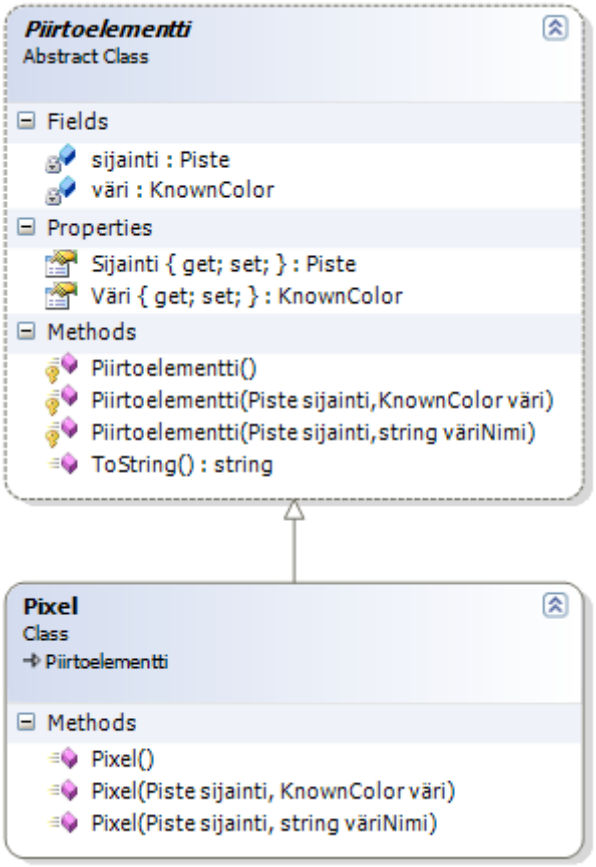
Tehtävä

Tee aluksi `PiirtoElementti` –luokka: Olkoon luokka sellaisenaan sovelluksille "käyttökelvoton", eli tee luokasta abstract –tyyppinen.

Toteuta `PiirtoElementtiin` kuitenkin toiminnallisuutta seuraavasti:

- Tee luokalle kenttä `System.Drawing.KnownColor väri` ja sille ominaisuus `Väri`.
- Samoin kenttä `Piste sijainti` ja ominaisuus `Sijainti`.
- Tee luokalle myös seuraavat konstruktorit:
  - Parametriton, jolloin Sijainti = (0,0) ja väri=musta
  - Parametreina Piste ja `System.Drawing.KnownColor`
  - Parametreina `Piste` ja `string värinimi`. Värinimi tulee muuttaa `System.Drawing.KnownColor`:iksi, siten, että mahdollisilla väärillä värinimillä käytetään väriä Black.
- Toteuta `PiirtoElementille` myös `ToString( )` –metodi, joka tulostaa ilmentymän luokan nimen, sijainnin ja värin, esim. *Pixel: (5,6),Red*.

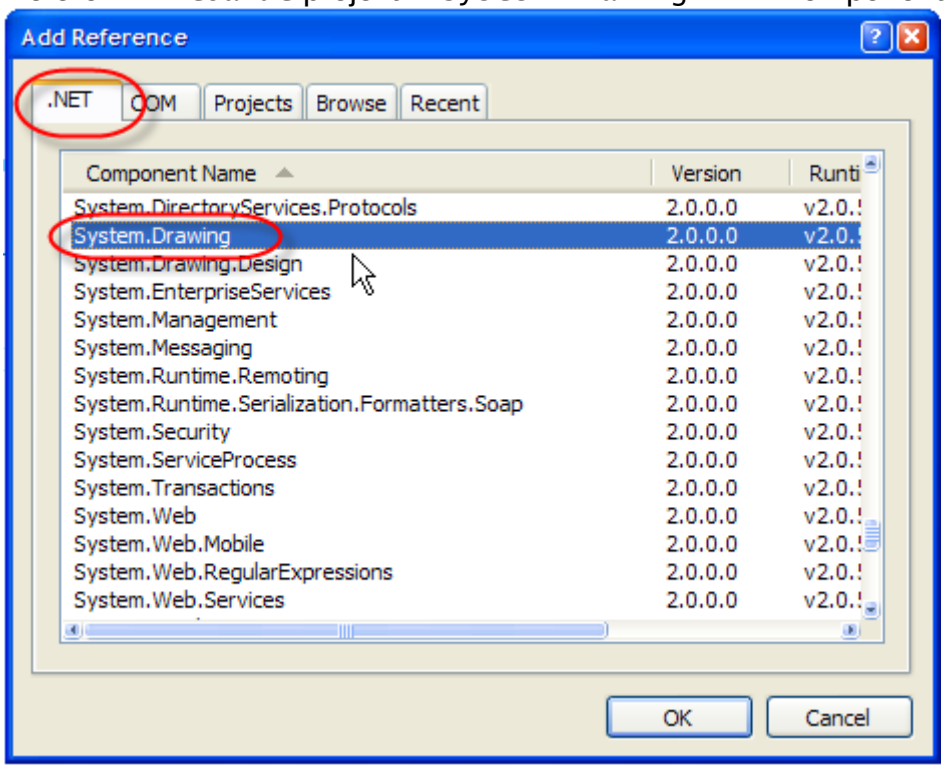
Tee myös luokka `Pixel`, joka periytyy `PiirtoElementti`-luokasta. Sen toteutus saa periytyä aluksi sellaisenaan PiirtoElementistä, eli toteuta tässä vaiheessa luokkaan vain konstruktorit.



Laajenna testeriä siten, että voit testata myös Pixelin instantiointia

Toimenpiteet, Piirtoavut

1. Referoi `Piirtoavut`-projektiin `System.Drawing.Dll` –komponentti.



Piirtoelementti:

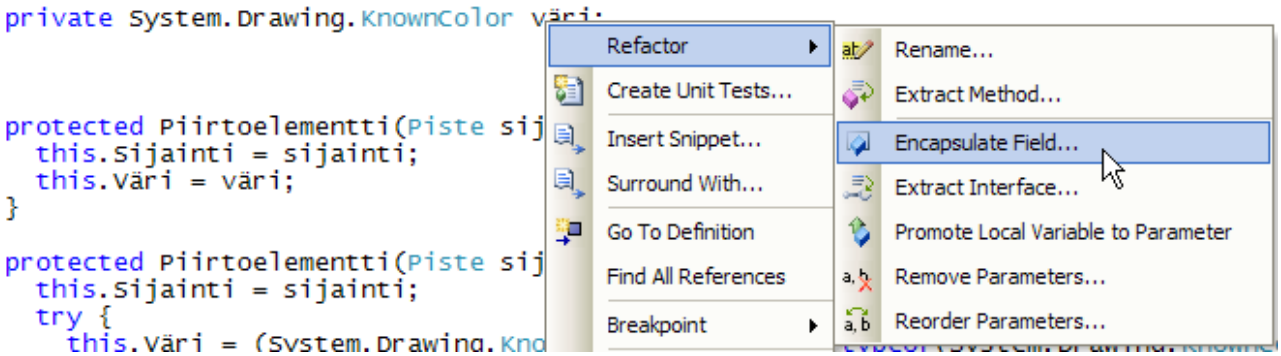
2. Lisää `Piirtoavut` projektiin uusi luokka `PiirtoElementti`. Muuta luokka julkiseksi abstraktiksi luokaksi

```
public abstract class Piirtoelementti {
```

3. Toteuta `väri`-kenttä ja sille property. Tehdään property tällä kertaa refactoroinnin avulla. Ensiksi lisää kyseinen kenttä

```
private System.Drawing.KnownColor väri;
```

4. Paina hiiren oikeaa `väri`-kenttänimen päällä, ja valitse: **Refactor | Encapsulate Field…**



Propertyn nimeksi saa tulla `Väri` (on oletus), ja sille generoidaan sekä `set` että `get` (jälleen oletus). Myös mahdolliset olemassa olevat viittaukset voidaan päivittää (eli paina vain Ok pari kertaa).

5. Tee vastaavasti kenttä `private Piste sijainti` ja siitä property `public Piste Sijainti`.

6. Tee `Piirtoelementin` konstruktorit. Konstruktorien näkyvyys on `protected`.

```
protected Piirtoelementti(Piste sijainti,
System.Drawing.KnownColor väri) {
  this.Sijainti = sijainti;
  this.Väri = väri;
}

protected Piirtoelementti(Piste sijainti, string väriNimi)
{
  this.Sijainti = sijainti;
  try {
    this.Väri = (System.Drawing.KnownColor)Enum.Parse(
              typeof(System.Drawing.KnownColor), väriNimi);
  }
  catch {
    this.väri = System.Drawing.KnownColor.Black;
  }
}

protected Piirtoelementti() : this(new Piste(),
System.Drawing.KnownColor.Black) { }
```

7. Toteuta luokkaan `ToString()`-metodi (muista käyttää override –wizardia, siis kirjoitat sanan `override` + välilyönti, ja valitset korvattavan metodin listasta)

```
public override string ToString() {
  return this.GetType().Name + ": " + Sijainti.ToString()
       + ", " + Väri.ToString();
}
```

8. Käännä, pitäisi kääntyä virheettä.

Pixel

1. Lisää `Piirtoavut`-projektiin uusi luokka: `Pixel`.
2. Muuta luokka julkiseksi ja periytymään `Piirtoelementti`-luokasta.

```
public class Pixel : Piirtoelementti {
```

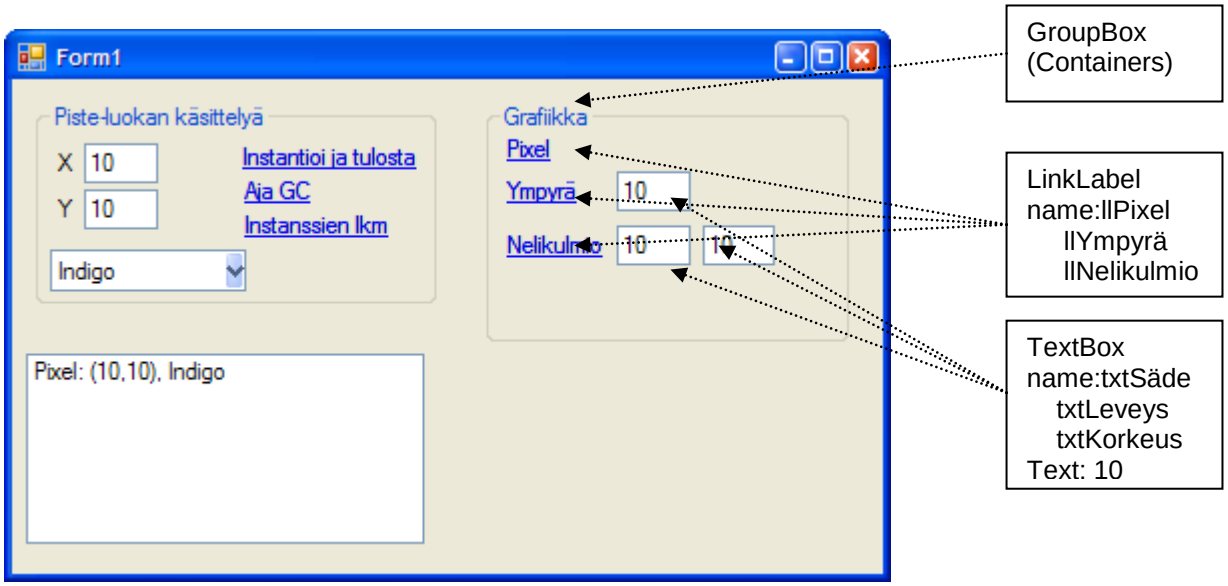
3. Toteuta Pixeliin kaikki kolme konstruktoria, jotka kutsuvat vain vastaavaa kantaluokan konstruktoria.

```
public Pixel() : base() { }

public Pixel(Piste sijainti,
             System.Drawing.KnownColor väri) :
       base(sijainti, väri) { }

public Pixel(Piste sijainti, string väriNimi) :
       base(sijainti, väriNimi) { }
```

4. Käännä.

Testeri

5. Maalaa Testeri-lomakkeelle Piirtoelementtien testauspainonapit seuraavasti (kaikkia ei vielä tässä vaiheessa tarvita). Oheisessa kuvassa kontrollit ja niille tehdyt asetukset:



6. Lisää lomakkeen Load-tapahtumaan `cbVäri`-listan täyttö. Load-tapahtumakäsittelijän saat helpoiten muodostettua tuplaklikkaamalla lomakkeen otsikkopalkkia. Listan täyttökoodi on (ja oletusvärin valinta):

```
cbVäri.DataSource =
System.Enum.GetNames(typeof(KnownColor));
cbVäri.SelectedItem = "Indigo";
```

7. Tee Pixel linkLabelin click-tapahtumaan Pixelin luonti ja tulostus listaan. Koodi on:

```
Piirtoavut.Pixel px;
px = new Piirtoavut.Pixel(
    new Piirtoavut.Piste(int.Parse(txtX.Text),
                         int.Parse(txtY.Text)),
    cbVäri.Text);
lbTulostus.Items.Add(px.ToString());
```

8. Testaa.

Mallivastaus on VSS:n projekti **Olioharjoitus_07**

Harjoitus 8: **Piirtäminen, virtuaalinen metodi**

Tausta

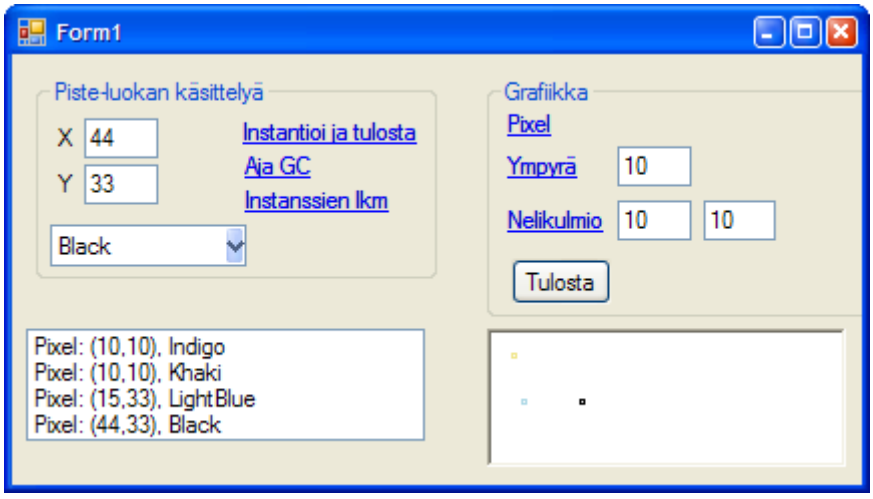
Johdettu luokka voi toteuttaa (tai korvata) kantaluokkaan määritellyt abstact ja virtual -metodit. Metodi määritellään kantaluokkaan (eikä vasta johdettuun luokkaan) jotta

- *voidaan tehdä "pääohjelma" käyttäen kantaluokan rajapintaa (abstact-metodit)*
- *metodille voidaan tehdä oletustoteutus kantaluokkaan (virtual-metodit). Johdetut luokat voivat korvata tämän toiminnallisuuden omalla toteutuksella.*

Tässä harjoituksessa konkretisoidaan tähän mennessä tehtyjä luokkia. Toteutetaan PiirtoElementeille metodi `Piirrä`, jossa objekti piirtää itsensä annettuun grafiikka-alustaan. Vaikka jokainen piirtoelementti toteuttaa sen omalla tavallaan, määritellään metodi kantaluokkaan (`PiirtoElementti`), jotta kuhunkin toteuttavaan luokkaan metodin rajapinta tulee samanlaiseksi. `Piirrä`-metodille ei voida kantaluokkaan tehdä minkäänlaista oletustoteutusta, joten metodi määritellään abstraktiksi.

Tehtävä

Tee `PiirtoElementille` abstrakti metodi `Piirrä(System.Drawing.Graphics g)` ja toteuta metodi `Pixel`-luokassa. Tee Testerin lomakkeelle `PictureBox`, johon voidaan tällä metodilla piirtää `pixel`-objektit. Lisää `Piirrä`-kutsu Pixelin instantiointinappiin.



Toimenpiteet

Piirtoelementti ja Pixel

1. Lisää Piirtoelementti-luokkaan Piirrä-metodin esittely

```
public abstract void Piirrä(System.Drawing.Graphics g);
```

2. Kokeile kääntää. Pixel-luokasta tulee nyt käännösvirhe. Miksi?

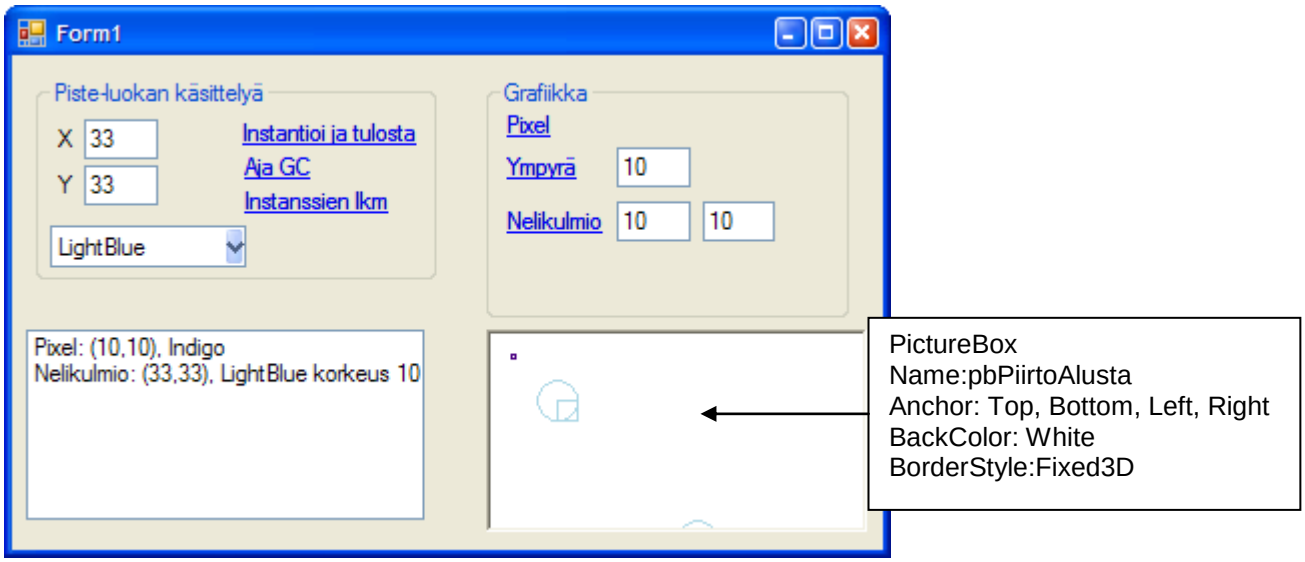
   Vast: _____
   (Pixel-luokka ei ole abstrakti luokka, jonka vuoksi sen tulee toteuttaa kaikki kantaluokkien abstraktit metodit).

3. Toteuta Pixel-luokkaan piirto siten, että valitulla värillä piirretään pieni (koko 2 yksikköä) nelikulmio. Metodin rungon voi generoida helpoiten ”override-wizardilla”. Huomio, että Pen-objekti on disposable-olio, joten sitä on käytettävä using-lohkolla. Tämä ns. resurssien vapauttaminen käsitellään varsinaisesti myöhemmin.
   Huom: tässä koodiesimerkissä on Pixel-luokassa otettu käyttöön `System.Drawing` –nimiavuruus (siis `using System.Drawing;`

```
public override void Piirrä(Graphics g) {
  using (Pen p = new Pen(Color.FromKnownColor(Väri))) {
    g.DrawRectangle(p, Sijainti.X, Sijainti.Y, 2, 2);
  }
}
```

Tester

4. Maalaa Testeriin PictureBox piirtoalustaksi



5. Lisää Pixel-linkin painallukseen myös piirtäminen.

```
px.Piirrä(pbPiirtoalusta.CreateGraphics());
```

6. Testaa. Pixel pitäisi tulla piirtoalustaan pienenä nelikulmiona. Kokeile eri värejä ja sijainteja.

Mallivastaus on VSS:n projekti **Olioharjoitus_08**

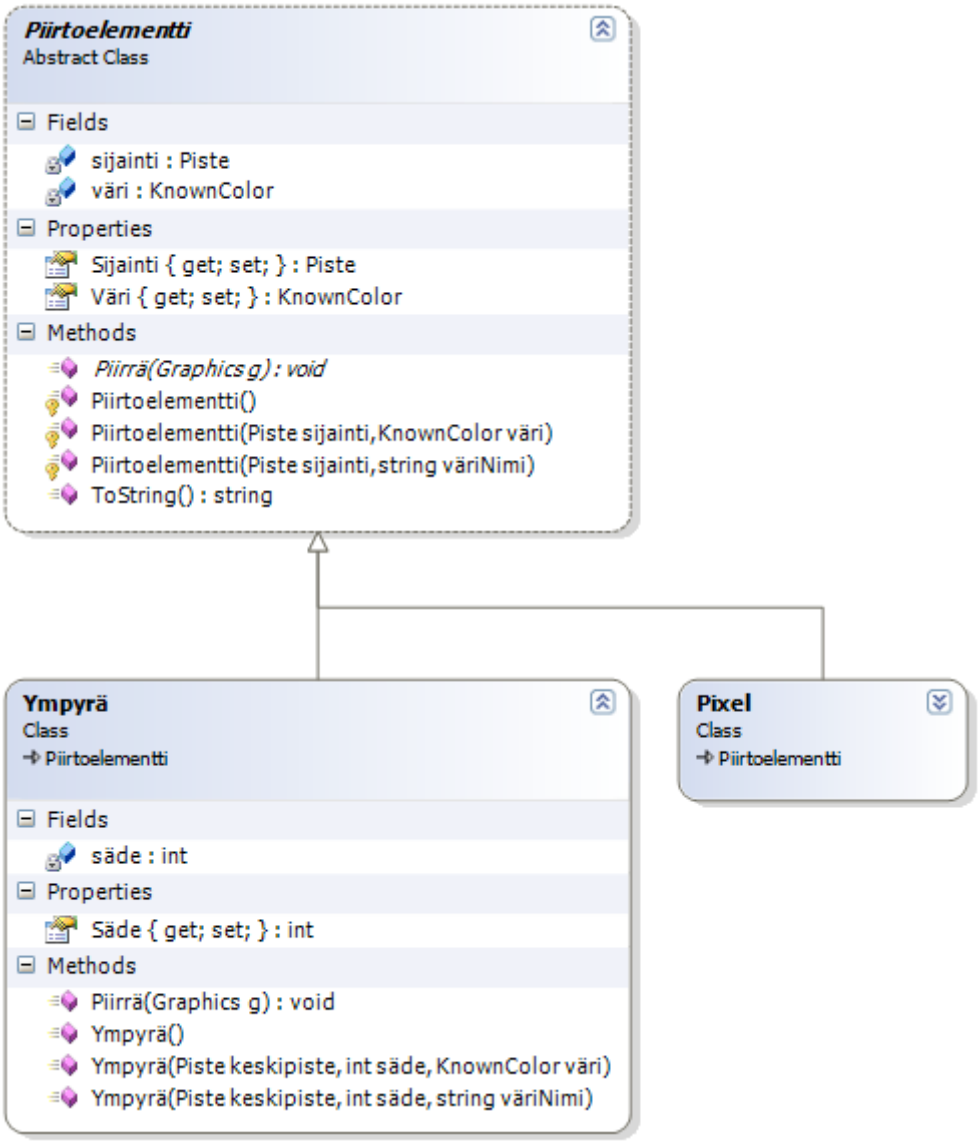
Harjoitus 9: **Periminen, Ympyrä ja Nelikulmio**

Tausta

Tehdään loput piirtoelementti-luokat,eli `Ympyrä` ja `Nelikulmio`. Niissä tulee korvata vain konstruktorit ja `ToString`, sekä toteuttaa `Piirrä`-metodi.

Tehtävä

Toteuta luokka `Ympyrä`, joka periytyy `PiirtoElementistä`. `Sijainti` määrittelee ympyrän keskipisteen. Määrittele luokalle lisäksi ominaisuus `Säde`. Lisää testeriin ympyrän instantiointi ja tulostus.



Toimenpiteet, Piirtoelementti

1. Lisää Piirtoavut-projektiin uusi luokka: `Ympyrä`. Periytä se Piirtoelemantistä ja merkitse julkiseksi luokaksi.

```
public class Ympyrä:Piirtoelementti {
```

2. Lisää luokalle `private int säde` ja julkinen ominaisuus `Säde`.

```
private int säde;

public int Säde {
  get { return säde; }
  set { säde = value; }
}
```

3. Tee Ympyrälle kaikki kolme konstruktoria. Parametrittomassa konstruktorissa tehdään Ympyrä, jonka säde olkoon 10 (oheinen esimerkkikoodi edellyttää `System.Drawing`-nimiavaruuden käyttöönottoa)

```
public Ympyrä() : this(new Piste(), 10, KnownColor.Black)
{ }

public Ympyrä(Piste keskipiste, int säde, KnownColor väri)
      : base(keskipiste, väri) {
  this.Säde = säde;
}

public Ympyrä(Piste keskipiste, int säde, string väriNimi)
     : base(keskipiste, väriNimi) {
 this.Säde = säde;
}
```

4. Toteuta Piirrä-metodi (rungon saat jälleen generoitua ”override-wizardilla”)

```
public override void Piirrä(Graphics g) {
  using (Pen p = new Pen(Color.FromKnownColor(Väri))) {
    g.DrawEllipse(p, Sijainti.X - Säde, Sijainti.Y - Säde,
                  Säde * 2, Säde * 2);
  }
}
```

Toimenpiteet, Tester

5. Toteuta testerin Ympyrä LinkLabelin click. Tapahtumassa instantioidaan Ympyrä ja tulostetaan se sekä `ToString`- että `Piirrä`-metodeilla.

```
Piirtoavut.Ympyrä y = new Piirtoavut.Ympyrä(
     new Piirtoavut.Piste(int.Parse(txtX.Text),
                          int.Parse(txtY.Text)),
     int.Parse(txtSäde.Text), cbVäri.Text);
```

```
lbTulostus.Items.Add(y.ToString());

y.Piirrä(pbPiirtoalusta.CreateGraphics());
```

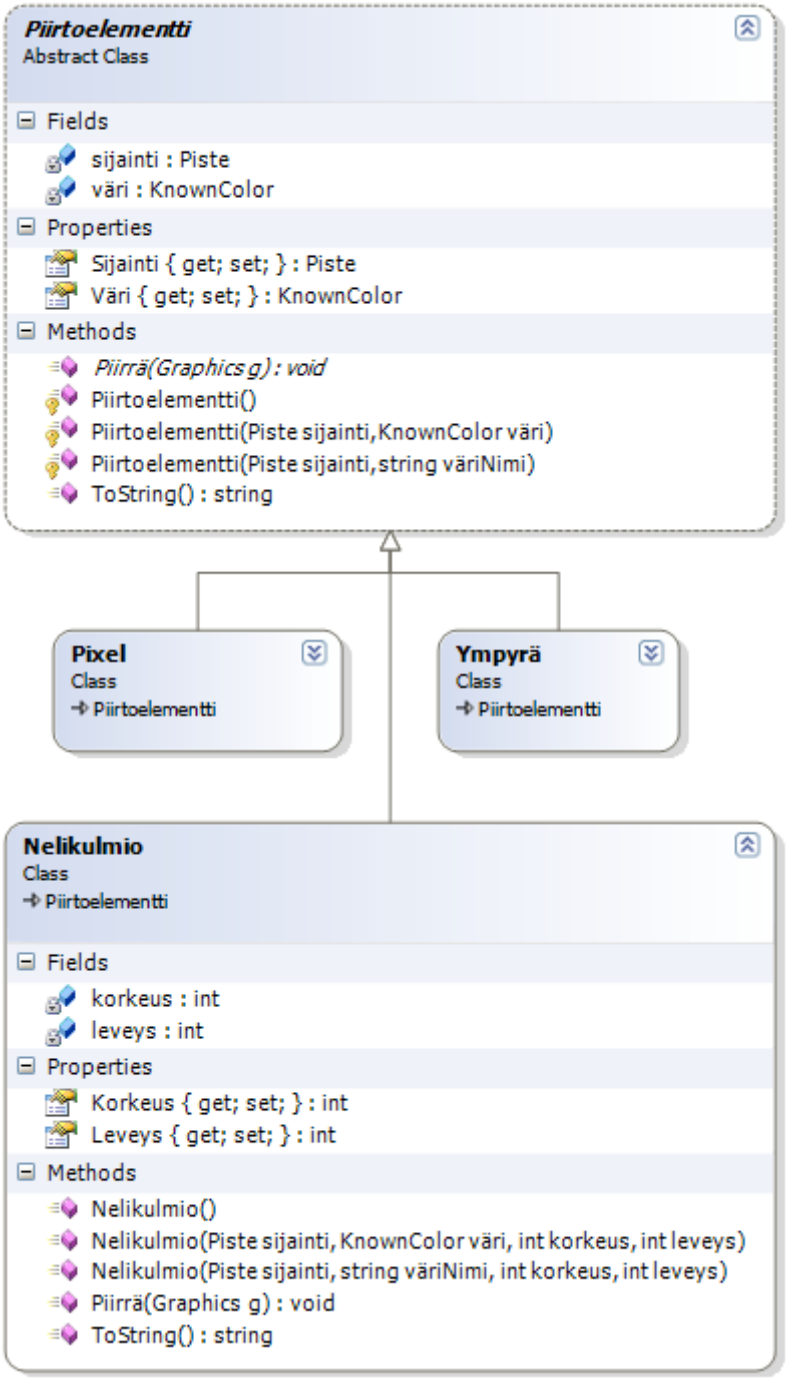
6. Testaa. Totea, että ympyrä tulostuu piirtoalustalle oikein mutta tulostuksessa (`ToString`()) ei ole säteen arvoa.

7. Toteuta ympyrälle oma versio ToString()-metodista jotta myös säde tulostuisi.

```
public override string ToString() {
      return base.ToString() + " säde " + Säde;
}
```

Lisätehtävä

Toteuta myös luokka Nelikulmio siten, että sijainti määrittelee nelikulmion vasemman ylänurkan paikan, ja leveys ja korkeus määrittelevät muut pisteet. Piirtäminen tapahtuu kuten Pixelilläkin (luokkakaavio seuraavalla sivulla).



Mallivastaus on VSS:n projekti **Olioharjoitus_09**

Harjoitus 10: **Kokoelmaluokka**

Tausta

Kokoelmat ovat olio-ohjelmoinnin taulukoita. Generics-kokoelmaluokat ovat tyyppiturvallisia, toisin sanoen kokoelman jäsenten (ja mahdollisen avaimen) tyyppi voidaan määritellä.

System.Collections.ObjectModel-nimiavaruudessa on geneerisiä kantaluokkia omien kokoelmien toteuttamiseen sekä wrappereitä, joilla voidaan esimerkiksi tehdä read-only kokoelmia.

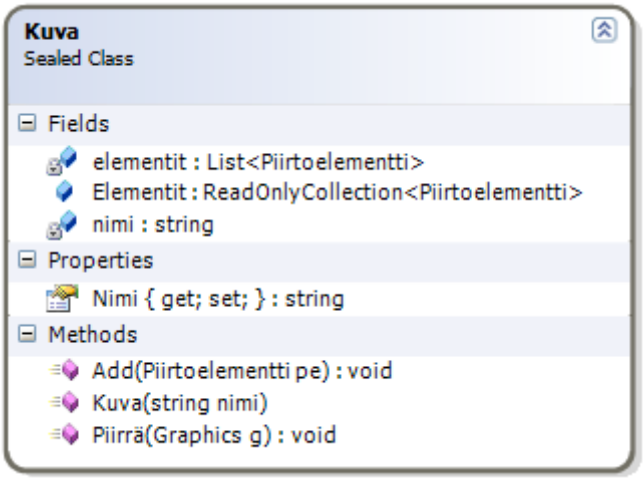
Windows Forms-sovelluksessa sovellus on vastuussa kuvion uudelleen piirrosta. Tämän vuoksi sovelluksen on tiedettävä kaikki piirtoelementtinsä mahdollista uudelleenpiirtoa varten.

Tehtävä

Piirros-sovelluksessa on sellainen ominaisuus, että jos kuvio häipyy (esimerkiksi toinen ikkuna tulee sen päälle tai ikkuna minimoidaan), niin sovellus ei osaa piirtää kuviota uudelleen ruudulle. Tämän vuoksi muutetaan sovellusta siten, että

- Piirtoelementit tallennetaan kokoelmaan. Kokoelma näytetään read-only –tyyppisenä ulospäin (kenttä nimeltä `Elementit`)
- Josta huolehtii uusi luokka nimeltä `Kuva`. Tämä luokka tehdään Piirtoavut-komponenttiin.
- `Kuva`-luokka hoitaa kaikkien piirtoelementtien piirrättämisen `Piirrä`-metodin avulla.
- Kuvalla on myös nimi. Nimi ei saa olla null (mutta tyhjä nimi, siis "", on sallittu). Siis ensi kertaa propertyä todella hyödynnetään näissä harjoituksissa :)

Kuvio-luokkaan tullaan jatkossa lisäämään paljon uusia vastuita, mm. kuvion tallennus ja lukeminen. Kuvio-luokan rakenne tämän harjoituksen jälkeen on seuraava:



Toimenpiteet, Piirtoavut

8. Lisää Piirtoavut projektiin luokka: Kuva. Tee siitä julkinen sealed-luokka.

```
public sealed class Kuva {
```

9. Lisää luokkaan private `elementit`-kokoelmajäsen. Tässä kokoelmassa on kuvan piirtoelementit.

```
private List<Piirtoelementti> elementit =
                new List<Piirtoelementti>();
```

10. Lisää luokkaan julkinen `Elementit` kokoelmajäsen. Tämä jäsen wrappaa private elementit-jäsenen siten, että ulospäin Elementit on vain read-only kokoelma.

```
public System.Collections.ObjectModel.
       ReadOnlyCollection<Piirtoelementti> Elementit;
```

11. Lisää nimi-kenttä ja sille property. Nimi ei saa olla null, joten set-metodissa on mahdollinen `null` arvo on muutettava `string.Empty` –arvoksi.

```
private string nimi;

//kuvalla on oltava nimi, vaikka sitten tyhjä
public string Nimi {
  get { return nimi; }
  set {
    if (value != null)
      nimi = value;
    else
      nimi = string.Empty;  }
}
```

12. Lisää luokkaan konstruktori, jossa kuvalle on annettava nimi.

```
public Kuva(string nimi) {
   Elementit =
      new System.Collections.ObjectModel.
         ReadOnlyCollection<Piirtoelementti>(elementit);
   this.Nimi = nimi;  // käytetään propertyä
}
```

13. Lisää Add-metodi, jossa kuvaan lisätään uusi elementti.

```
public void Add(Piirtoelementti pe){
  elementit.Add(pe);
}
```

14. Lisää luokkaan Piirrä-metodi, jossa piirretään kuvan kaikki elementit.

```
public void Piirrä(System.Drawing.Graphics g) {
  foreach (Piirtoavut.Piirtoelementti pe in Elementit) {
    pe.Piirrä(g);
  }
}
```

Toimenpiteet, Tester

1. Lisää Form1-luokkaan private kenttä kuva (tämän voit lisätä mihin tahansa kohtaa Form1.cs-tiedostoa (toki luokan sisälle)). Yleensä kentät määritellään luokan alussa.

```
Piirtoavut.Kuva kuva = new Piirtoavut.Kuva(”testi”);
```

2. Muuta Pixelin, Ympyrän ja Nelikulmion luontikohdat (linkLabelien painalluskoodi) siten, että

   tehty piirtoelementti lisätään kuva:an

   Piirtoalusta invalidoidaan. Tämä aiheuttaa Paint-tapahtuman, johon kohta kirjoitetaan koodia

   Olemassa oleva elementin piirto poistetaan

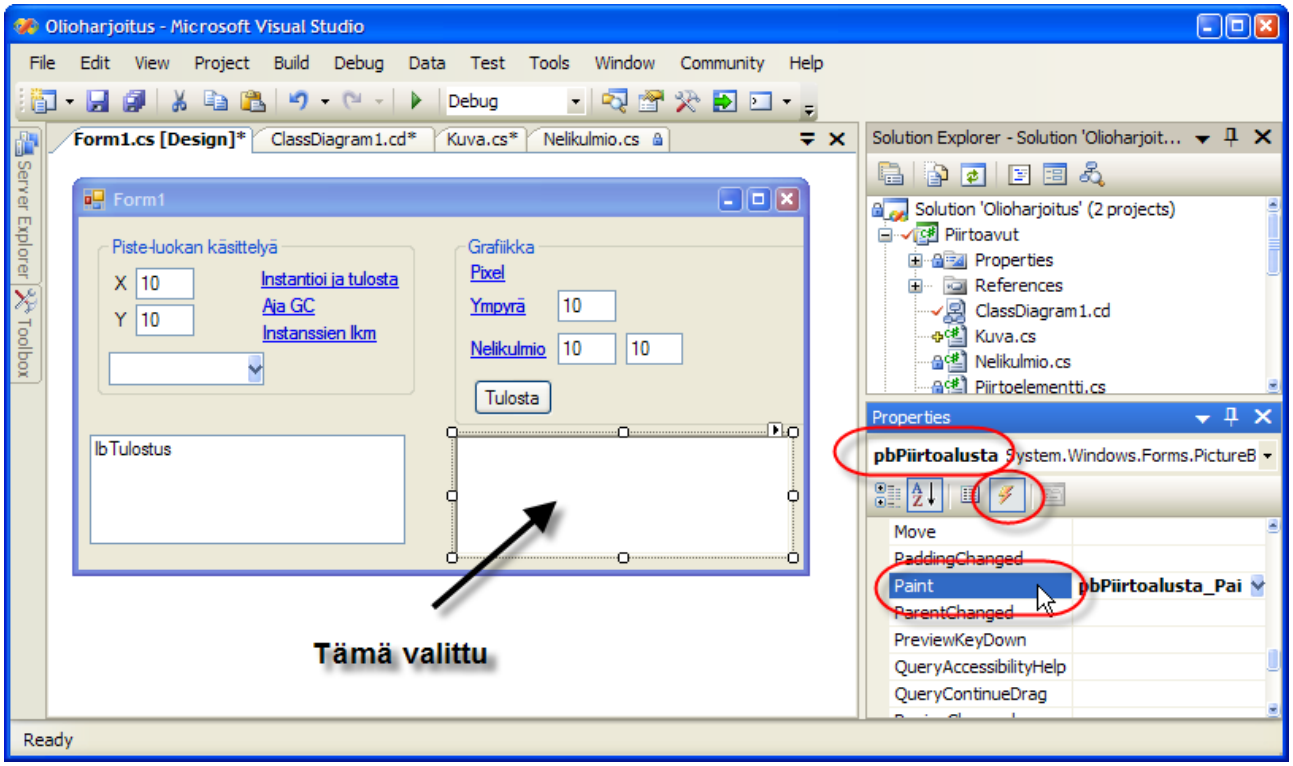
```
//tämä siis kommentoidaan ulos
//px.Piirrä(pbPiirtoalusta.CreateGraphics());

kuva.Add(px);
pbPiirtoalusta.Invalidate();
```

3. Lisää pbPiirtoalusta:n Paint-tapahtumaan kuvan piirto. Paint-tapahtuma ei ole pictureBoxin oletustapahtuma, tämän vuoksi aikaisemmin käytetty tuplaklikki-tapa ei toimikaan. Tapahtumakäsittelijä lisätään seuraavasti:

Form1:n designerissa (siis *Form1.cs [**Design**]* –ikkunassa) valitse pbPiirtoalusta.

Valitse Properties-lomakkeelta salamankuva (Events) 

Valitse Tapahtumalistasta Paint-rivi, jonka päällä tuplaklikkaa. Näin generoidaan tapahtumakäsittelijän tähän tapahtumaan



Tapahtumakäsittelijässä suorita `kuva.Piirrä`, grafiikkaobjekti tulee nyt tapahtuman argumenttina

```
private void pbPiirtoalusta_Paint(object sender,
PaintEventArgs e) {
  kuva.Piirrä(e.Graphics);
}
```

4. Testaa. Sovelluksen pitäisi nyt pystyä tulostamaan kuvio myös silloin, kun sen päältä siirretään ikkuna pois, tai minimize/restore-toimintojen jälkeen.

Lisätehtävä

Lisätehtävänä käytetään Kuva-luokan read-only kenttää Elementit. Lisätään testeriin painonappi ”*Tulosta*”, jolla tulostetaan kuvan kaikki tiedot tulostus-kanvaasille.

5. Lisää lomakkeelle Tulosta-painonappi (nimi: bTulosta)

6. Ja sen painallukseen kuvan tietojen tulostaminen.

```
lbTulostus.Items.Clear();
lbTulostus.Items.Add("Kuva:" + kuva.Nimi);
lbTulostus.Items.Add("piirtoelementtejä:" +
kuva.Elementit.Count + " kappaletta");
foreach(Piirtoavut.Piirtoelementti pe in kuva.Elementit)
  lbTulostus.Items.Add(pe.ToString());
}
```

Mallivastaus on VSS:n projekti **Olioharjoitus_10**

Harjoitus 11: **Rajapinta**

Tausta

Rajapinta on tärkeä olio-ohjelmoinnin abstrahointitapa, jonka merkitys C#:ssa on erityisen suuri koska moniperintää ei ole.

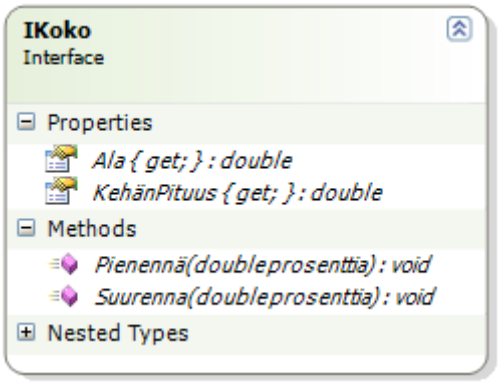
Rajapintaa käytetään erityisesti

- *toteuttamaan tyyppi-yhteneväisyys täysin erillistenkin luokkien välillä*
- *moniperinnän korvikkeena yksiperinnäisissä kielissä*
- *"tyypittämään tyyppejä", merkitsemään luokalle tietty ominaisuus*

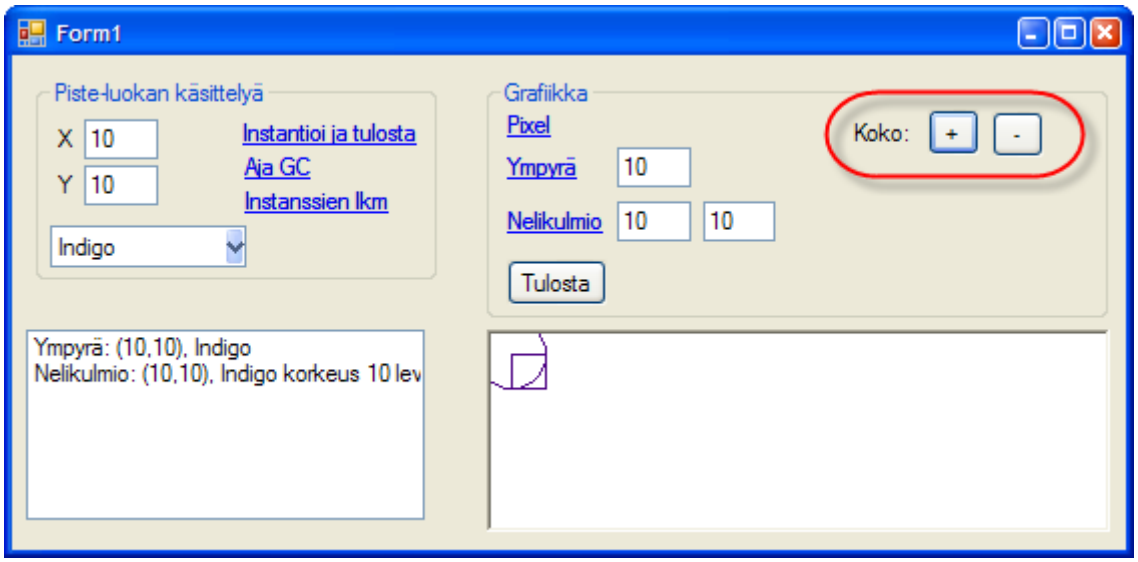
PiirtoAvut–toiminnallisuuteen liittyy mahdollisuus kysyä erityyppisiltä Piirtoelementeiltä niiden pinta-ala ja kehän pituus sekä muuttaa objektin kokoa haluttu määrä. Kaikki piirtoelementit eivät kuitenkaan voi toteuttaa näitä metodeja (kuten `Pixel`). Tämän vuoksi tarvitaan `IKoko`–rajapinta, jonka toteuttaa `Ympyrä` ja `Nelikulmio`-luokat

Tehtävä

Tee `IKoko` rajapinta `Piirtoavut`-projektiin. Rajapinnan kuvaus on alla. Toteuta rajapinta Ympyrä- ja Nelikulmio-luokkiin.



Testeriin lisätään mahdollisuus muuttaa kuvan kokoa.



Oliomaisesti toteutetuissa sovelluksissa tulee usein tilanne, jossa joudutaan tarkoin harkitsemaan, minkä olion vastuulla toiminnot asetetaan. Usein toiminnot ovat ”läpikulku”-metodeja (olio delegoi itseltä pyydetyn toiminnon omistamilleen olioille). Kuvan koon muuttaminen on esimerkki tällaisesta toiminnosta. Toiminto toteutetaan siten, että käyttäjä pyytää käyttöliittymältä kuvan koon muuttamisen. Pyyntö menee lomakkeelta Kuvalle. Kuva delegoi pyynnön edelleen piirtoelementeille, jotka muuttavat oman kokonsa.

Toimenpiteet, IKoko

1. Lisää `Piirtoavut`-projektiin rajapinta `IKoko`

   1.1. hiiren oikeaa Solution Explorerissa Piirtoavut-projektin päällä **| Add New Item… |** valitse Interface, ja anna nimeksi `IKoko`.

   1.2. Muuta rajapinta julkiseksi

   ```
   public interface IKoko {
   ```

2. Koodaa rajapinta. Huomaa, että properteissa on ainoastaan get-osat.

   ```
   double Ala {
     get;
   }
   double KehänPituus {
     get;
   }
   void Pienennä(double prosenttia);
   void Suurenna(double prosenttia);
   ```

Toimenpiteet, Ympyrä

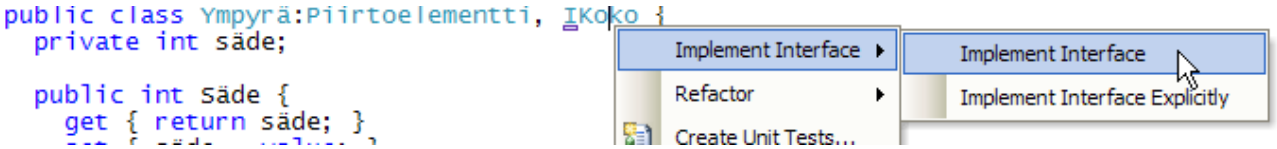
3. Muuta `Ympyrän` määrittelyä siten, että se toteuttaa myös `IKoko` rajapinnna.

```
public class Ympyrä:Piirtoelementti, IKoko {
```

4. Käännä sovellus. Ympyrä-luokasta tulee käännösvirheitä. Miksi?

   _____
   (Vast: luokka ei toteuta `IKoko` –rajapinnan metodeja)

   Toteutetaan rajapinnan metodit III-tavalla. Metodien runko on helpoin tehdä wizardilla. Paina hiiren oikeaa `Ympyrän` määrittelyrivin `IKoko`-sanan päällä **| Implement Interface | Implement Interface**



5. Toteuta rajapinnan metodit. Käytä ”peruskoululaskentaa” tai sitten katso alla olevaa koodiesimerkkiä. Oheisessa koodissa huomioidaan myös pienisäteisten ympyröiden koon muuttaminen.

```
public double Ala {
  get { return Math.PI * (double)Säde * (double)Säde; }
}

public double KehänPituus {
  get { return 2.0 * Math.PI * (double)Säde; }
}

public void Pienennä(double prosenttia) {
  Säde = System.Math.Max(0, (int)((double)Säde *
         (1.0 - prosenttia / 100.0)));
}

public void Suurenna(double prosenttia) {
  Säde += System.Math.Max(1, (int)((double)Säde *
          prosenttia / 100.0));
}
```

Toimenpiteet, Kuva

6. Tee Kuvalle metodi `public void Suurenna(double prosenttia)` ja vastaava `Pienennä` metodi. Metodin toteutuksessa koon muuttaminen delegoidaan piirtoelementeille. Kiinnitä huomio piirtoelementin castaukseen `IKoko`-tyyppiseksi.

```
public void Suurenna(double prosenttia) {
  foreach (Piirtoavut.Piirtoelementti pe in Elementit) {
```

```
    Piirtoavut.IKoko koko = pe as Piirtoavut.IKoko;
    if (koko != null)
      koko.Suurenna(prosenttia);
  }
}

public void Pienennä(double prosenttia) {
  foreach (Piirtoavut.Piirtoelementti pe in Elementit) {
    Piirtoavut.IKoko koko = pe as Piirtoavut.IKoko;
    if (koko != null)
      koko.Pienennä(prosenttia);
  }
}
```

Miksi täytyy olla varautunut siihen, että `koko == null`?

Vast: _____

(kaikki piirtoelementit eivät ole IKoko-tyyppisiä)

Toimenpiteet, Testeri

7. Maalaa testerille koon muuttamisen painonapit, katso tehtävässä olevaa lomakkekuvaa (painonappien nimet `bSuurenna` ja `bPienennä`)
8. Toteuta tapahtumakäsittelijät

```
private void bSuurenna_Click(object sender, EventArgs e) {
  kuva.Suurenna(10.0);
  pbPiirtoalusta.Invalidate();
}

private void bPienennä_Click(object sender, EventArgs e) {
  kuva.Pienennä(10.0);
  pbPiirtoalusta.Invalidate();
}
```

9. Testaa.

Lisätehtävä

10. Toteuta myös Nelikulmiolle `IKoko` rajapinta.

```
public double Ala {
  get { return (double)Korkeus* (double)Leveys; }
}
```

```
public double KehänPituus {
  get { return (2.0 * (double)Korkeus) + (2
*(double)Leveys); }
}

public void Pienennä(double prosenttia) {
  Korkeus = System.Math.Max(0,(int)((double)Korkeus *
            (1.0 - prosenttia / 100.0))) ;
  Leveys = System.Math.Max(0, (int)((double)Leveys *
            (1.0 - prosenttia / 100.0)));
}

public void Suurenna(double prosenttia) {
  Korkeus += System.Math.Max(1,(int)((double)Korkeus *
             prosenttia / 100.0)) ;
  Leveys += System.Math.Max(1,(int)((double)Leveys *
             prosenttia / 100.0));

}
```

11. Lisää Tulosta-painonappiin eri piirtoelementtien koon ja piirin pituuden tulostukset.

```
//looppiin lisätään
Piirtoavut.IKoko koko = pe as Piirtoavut.IKoko;
if (koko != null) {
  lbTulostus.Items.Add("   ala=" + koko.Ala);
  lbTulostus.Items.Add("   piiri=" + koko.KehänPituus);
}
}
```

Mallivastaus on VSS:n projekti **Olioharjoitus_11**

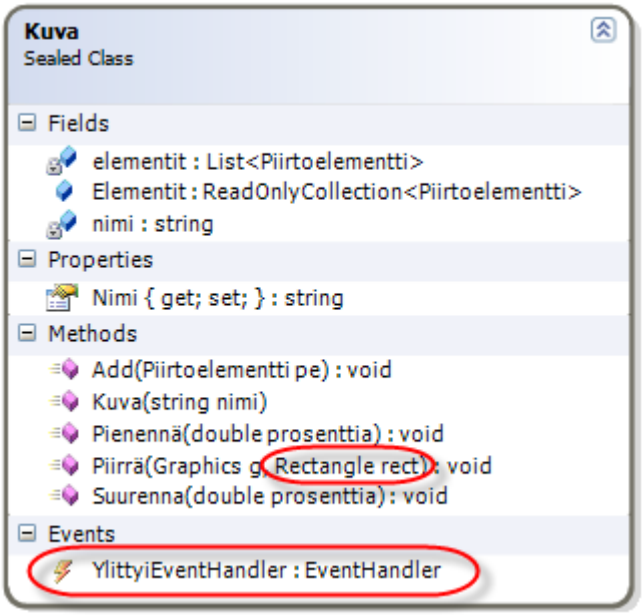
Harjoitus 12: **Delegaatti**

Tausta

Delegaatti-tyyppiä käytetään asynkroniseen metodikutsuun, callback-pointterina sekä tapahtumakäsittelyyn. Tässä harjoituksessa sitä käytetään jälkimmäiseen.

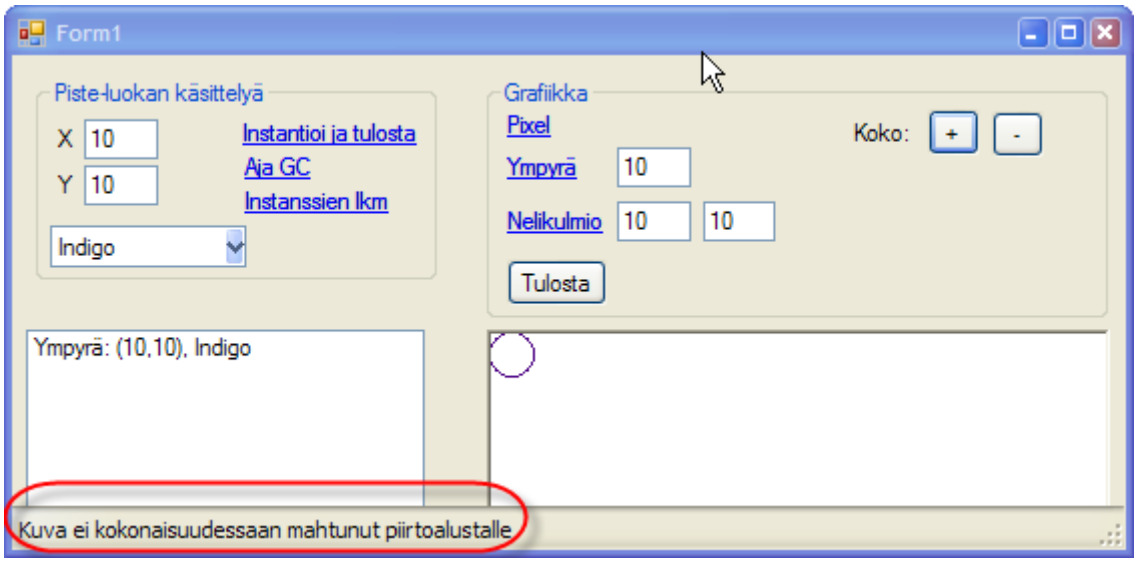
Tehtävä

Tee Kuva-tyypille tapahtuma, jossa se kertoo jos se ei mahdu piirtokanvaasille. Piirrä-metodiin pitää uutena parametrina tuoda piirtokanvaasin koko. Kuvan uusi rakenne on:



Piirtoelementille lisätään metodi `public virtual bool Mahtuuko(System.Drawing.RectangleF rect)`, jolla voidaan kysyä, mahtuuko pe annettuun nelikulmioon.

Lisää testeriin kuvan ylitys-tapahtumien kuuntelu. Jos kuva ei mahdu kanvaasille, siitä kerrotaan status-rivillä.



Toimenpiteet, Piirtoelementit

1. Lisää Piirtoelementtiin `public virtual bool Mahtuuko` metodi. Testaamisen yksinkertaistamiseksi metodista ei tehdä abstraktia, vaan tehdään oletustoteutus, joka palauttaa `true`.

```
public virtual bool Mahtuuko(System.Drawing.RectangleF
rect) {
  return true;
}
```

2. Tehdään ympyrään todellinen toteutus. Jälleen tarvitaan ”peruskoulumatikkaa”

```
public override bool Mahtuuko(RectangleF rect) {
  return rect.Contains(new RectangleF(Sijainti.X - Säde,
                Sijainti.Y - Säde, 2 * Säde, 2 * Säde));
}
```

Toimenpiteet, Kuva

3. Lisätään `Kuva`-luokkaan `YlittyiEventHandler` kenttä. Tällä tapahtuma tullaan nostamaan piirron yhteydessä, mikäli havaitaan, että jokin piirtoelementeistä ei mahdukaan kanvaasille.

```
public event EventHandler YlittyiEventHandler;
```

4. Korjaa Piirrä-metodia

4.1. lisää parametri `System.Drawing.Rectangle rect`, jolla kutsuja ilmoittaa kanvaasin koon (Graphics-parametrin rect sisältää tiedon vain siitä, kuinka suuri alue piirrossa todella päivitetään. Nämä alueet voivat olla eri kokoisia).

4.2. Nosta `Ylittyi..` tapahtuma, jos piirto menee alueen ulkopuolelle.

```
public void Piirrä(System.Drawing.Graphics g,
System.Drawing.Rectangle rect) {

  bool ylittyi = false;
  foreach (Piirtoavut.Piirtoelementti pe in Elementit) {
    pe.Piirrä(g);
    if (!pe.Mahtuuko(rect))
      ylittyi = true;
  }

  if (ylittyi && YlittyiEventHandler != null)
    YlittyiEventHandler(this, EventArgs.Empty);
}
```

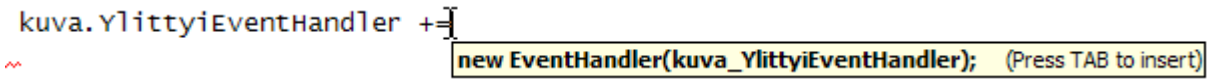
Toimenpiteet, Tester

1. Maalaa lomakkeelle `StatuStrip` (Menus&Toolbars välilehti) ja lisää kontrolliin label (StatusStripin vasemmassa reunassa olevalla painikkeella).

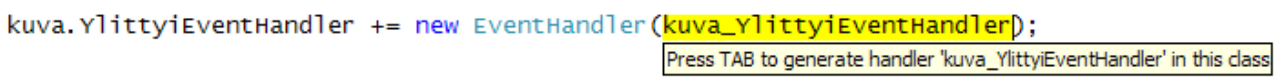
2. Lomakkeen `Load`-tapahtumassa kiinnitä Ylittyi-tapahtumakäsittelijä (wizardi generoi myös tapahtumakäsittelijämetodin). Tämä tehdään seuraavasti:

   2.1. Etsi Load-tapahtumakäsittelijä (siinähän asetetaan vbVäri-kontrolli)

   2.2. Kirjoita (siis valitse intellisensestä) `kuva.YlittyiEvenHandler +=` ja esiin tulevan wizardin ohjeen mukaisesti paina TAB (jolloin täydentyy instantiointi-lause)



   2.3. ja paina uudelleen Tab, jolloin generoidaan tapahtumakäsittelymetodi.



3. Koodaa käsittelijään status-kentän päivitys

```
toolStripStatusLabel1.Text=
     "Kuva ei kokonaisuudessaan mahtunut piirtoalustalle";
```

4. Joudut korjaamaan Kuva.Piirrä-metodin kutsua johon nyt tarvitaan kaksi parametriä. Kontrollin koon saat selville ClientRectangle-ominaisuudella.

5. Testaa. Imoituksen pitäisi jo tulla näkyviin, mutta status-kenttää ei siivota.

6. Lisätään viimeistelyt.

    6.1. Paint-tapahtumassa tyhjätään statusStrip. Huom: tee tämä tapahtumakäsittelyssä ENSIN, siis ennen piirtoa!

    ```
    toolStripStatusLabel1.Text = string.Empty;
    ```

    6.2. Generoi pbPiirtoalustalle Resize –tapahtumakäsittelijä (designerissa, muista ”salama”). Tapahtumassa piirretytään kuva uudelleen, jotta tiedetään, mahtuuko se tähän uuteen alustakokoon.

    ```
    pbPiirtoalusta.Invalidate();
    ```

7. Testaa. Pitäisi toimia.

Lisätehtävä

8. Toteuta Mahtuuko myös nelikulmiolle ja Pixelille

    Mallivastaus on VSS:n projekti **Olioharjoitus_12**

Harjoitus 13: **Lisätehtävä: Operaattorien kuormitus ja yksikkötestaus**

Tausta

C#:lla on mahdollista kuormittaa myös operaattoreita. Tämä on tyypillistä matemaattistyyppisille luokille, kuten tämän esimerkin Piste-luokalle.

Operaattorit toteutetaan aina luokan staattisina metodeina.

Visual Studio sisältää yksikkötestausympäristön. Yksikkötestit on myös mahdollista generoida VS:lla.

Tehtävä

Tee Piste-luokkaan + ja – operaattoreiden toteutus. Tee operaatioiden testausta varten Tester-projekti.

Lisää käyttöliittymään kuvan siirto-toiminta, jossa tarvitaan näitä operaatioita. Kuvan elementtien siirrosta vastaa Kuva-tyyppi.

Toimenpiteet, Piste

1. Tee operaattorimetodit

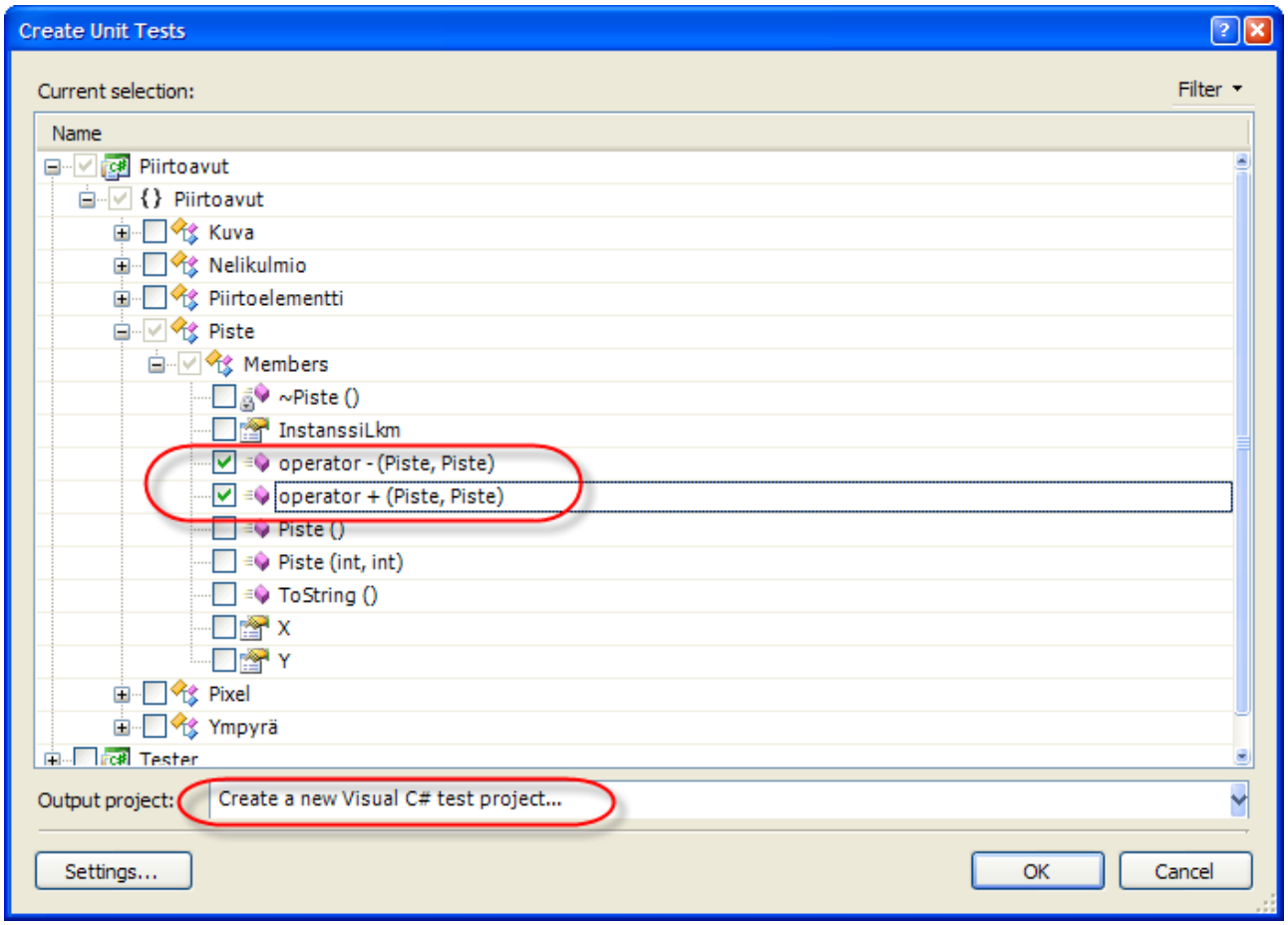
```
public static Piste operator +(Piste p1, Piste p2) {
  return new Piste(p1.X + p2.X, p1.Y + p2.Y);
}
public static Piste operator -(Piste p1, Piste p2) {
  return new Piste(p1.X - p2.X, p1.Y - p2.Y);
}
```

Toimenpiteet, Yksikkötestaus

2. Generoi yksikkötestausprojekti, jossa testataan nämä uudet operaattorit.

   2.1. Paina hiiren oikeaa Piste-luokan koodissa **| Create Tests…**

   2.2. Valitse molemmat operaattori-metodit ja Ouput-projektiksi uusi C# testiprojekti



3. Toteuta SubtractionTest. Ohessa esimerkkikoodi

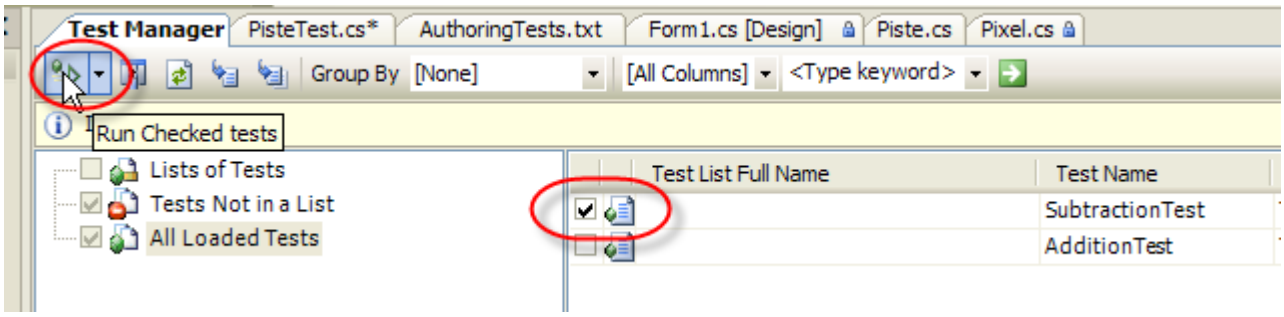
```
[TestMethod()]
public void SubtractionTest() {
  Piste p1 = new Piste(5,7);
  Piste p2 = new Piste(3,4);

  Piste expected = new Piste(2,3);
  Piste actual;

  actual = p1 - p2;
  Assert.AreEqual(expected, actual,
"Piirtoavut.Piste.operator - did not return the expected
value.");
}
```

4. Aja testi

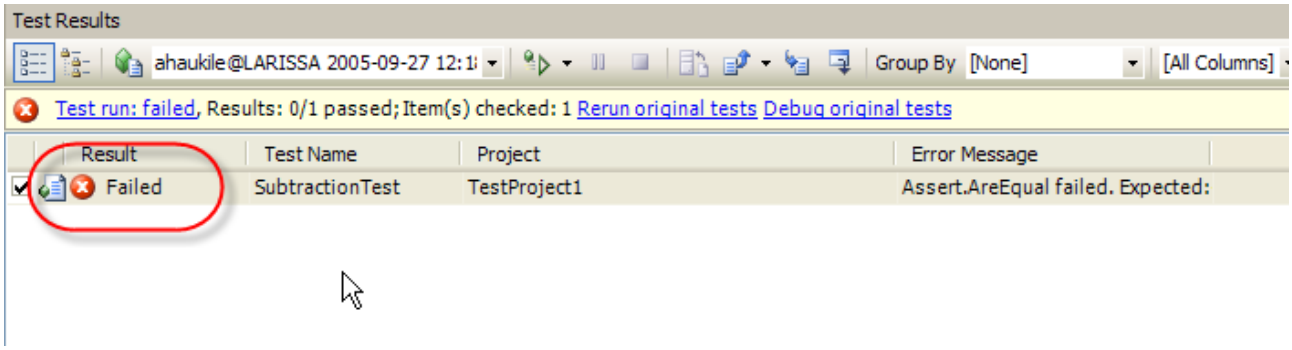
    4.1. Ota esiin Test Manager-ikkuna (**Test | Windows | Test Manager**)

    4.2. Ruksaa SubstractionTest

    4.3. Aja testi



5. Totea, että testi epäonnistui. Tuplaklikkaamalla testiriviä näet testin lisätiedot. Nyt virhe johtui testin koodista, testissä tutkitaan Piste-olioiden samanarvoisuutta (`AreEqual`), ja se taasen käyttää Equals() –metodia. Equals-metodi on kuormitettu object-luokassa, ja se taasen palauttaa `ReferenceEquals()` –arvon. Yhtäkaikki, Piste-tyyppisille utility- luokille on syytä toteuttaa myös == -operaattori.



6. Toteuta Piste-luokalle `==` operaattori (joka edellyttää `!=` operaattorin toteutusta), sekä kuormita `Equals()` –metodi (joka taasen edellyttää `GetHashCode()`:n toteuttamista)

```
public override bool Equals(object obj) {
  if (obj == null) return false;
  if (! (obj is Piste))
    return false;

  Piste p2 = (Piste)obj;
  return (this.X == p2.X && this.Y == p2.Y);
}


public static bool operator ==(Piste p1, Piste p2) {
 if (p1 == null)
   return false;
  return (p1.Equals(p2));
}

public static bool operator !=(Piste p1, Piste p2) {
  return  !(p1 == p2);
}
```

```
public override int GetHashCode() {
  return X ^ Y;
}
```

7. Aja testi uudelleen. Nyt pitäisi onnistua.

8. Toteuta AdditionTest ja aja myös sen testi.

```
[TestMethod()]
public void AdditionTest() {
  Piste p1 = new Piste(5, 7);
  Piste p2 = new Piste(3, 4);

  Piste expected = new Piste(8, 11);
  Piste actual;

  actual = p1 + p2;

  Assert.AreEqual(expected, actual,
"Piirtoavut.Piste.operator + did not return the expected
value.");
}
```

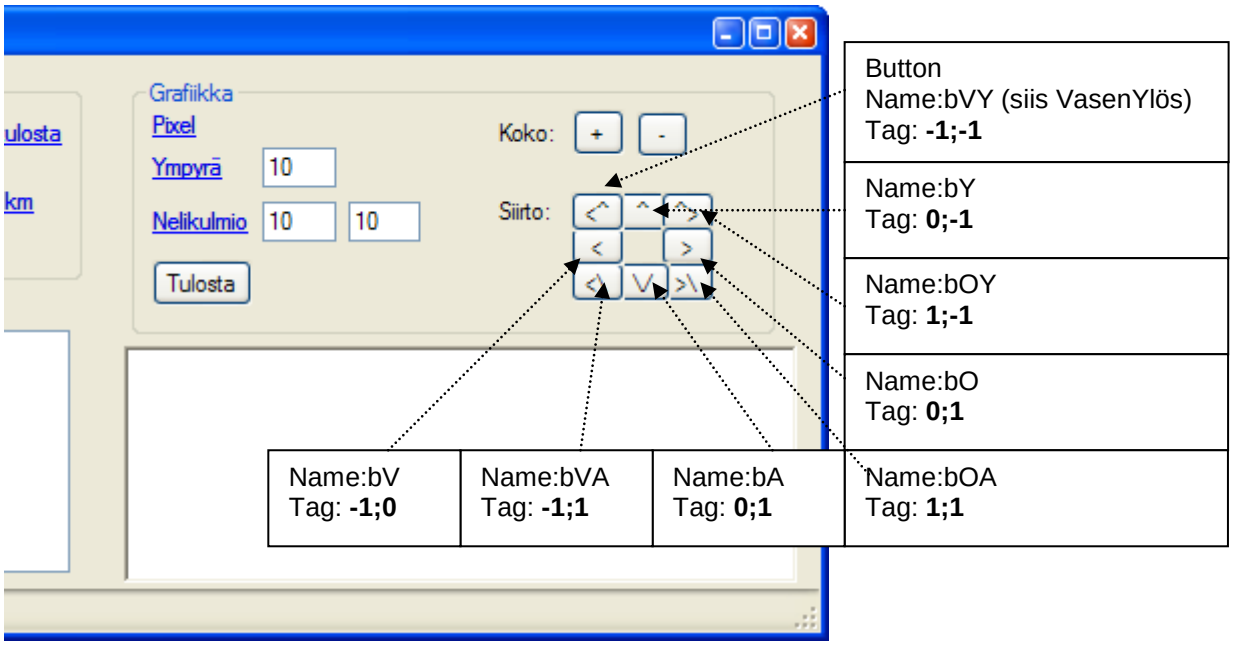
Toimenpiteet, Kuva

9. Tee Kuva-luokkaan metodi Siirrä, jonka avulla koko kuvaa voidaan siirtää.

```
public void Siirrä(Piste siirtopiste) {
  foreach (Piirtoelementti pe in Elementit)
    pe.Sijainti += siirtopiste;
}
```

Toimenpiteet, Testeri

1. Maalaa testerin lomakkeelle kuvan siirtonapit. Kunkin napin Tag- propertyyn laitetaan x;y –koordinaatit, paljonko kuvaa halutaan siirrettävän. Huomaa, että Windows-kanvaasin y-akseli kulkee alaspäin!

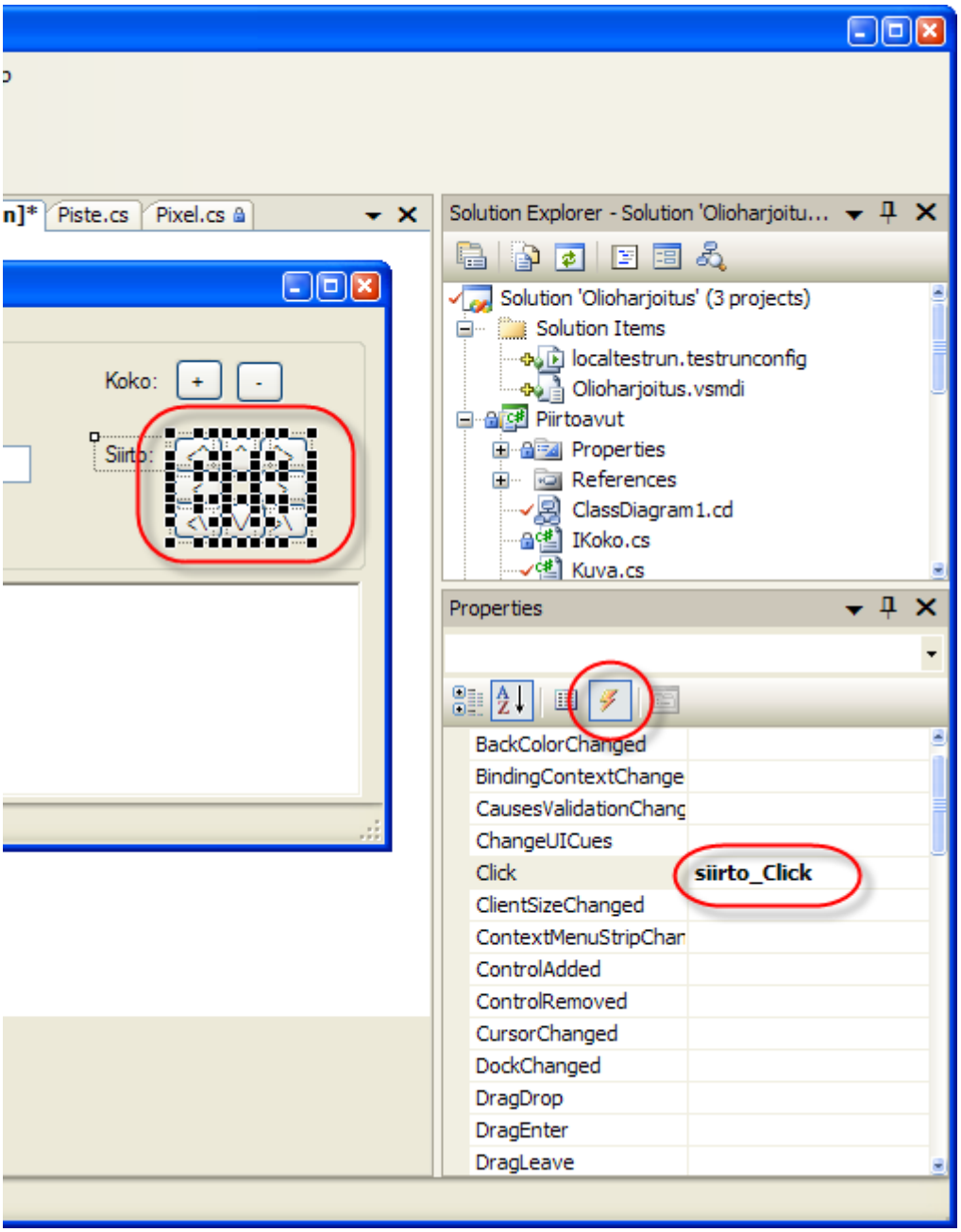


2. Valitse kaikki 8 painonappia, ja liitä niihin kaikkiin sama tapahtumakäsittelijä nimeltään `siirto_click`.

    2.1. Valitse kaikki siirto-painonapit lassoamalla

    2.2. Valitse Properties-lomakkeelta Events (=salamankuva)

    2.3. Valitse Click-tapahtumarivi. Kirjoita siihen metodinimeksi `siirto_click`. Tämä generoi tapahtumakäsittelijän ja kytkee kaikkien valittujen kontrollien click-tapahtuman tähän käsittelijään.



3. Toteuta siirto-koodi.

    3.1. painetun kontrollin Tag:issa on puolipisteellä eroteltuna siirtopisteen koordinaatit. Muodosta tästä tiedosta siirtopiste

    3.2. siirrä kuvaa em. pisteen verran

3.3. piirrä kuva uudelleen.

```
Button btn = sender as Button;
string[] osat = btn.Tag.ToString().Split(';');
if (osat.Length != 2) {
  MessageBox.Show("Siirto-painonapin tagissa ei ole
oikein");
  return;
}
Piirtoavut.Piste siirto = new
Piirtoavut.Piste(int.Parse(osat[0]), int.Parse(osat[1]));

kuva.Siirrä(siirto);
pbPiirtoalusta.Invalidate();
```

4. Testaa. Kuvan pitäisi siirtyä haluttuun suuntaan.

Mallivastaus on VSS:n projekti **Olioharjoitus_13**