Sisällys

Tehtävät:

Kuvat:

Malliohjelmat:

Lukijalle

Tämä moniste on kirjoitettu alun perin "*Graafisen käyttöliittymien ohjelmointi*" -kurssille syksyllä 1996. Monisteen alkuperäinen tarkoitus on olla pikakurssi siirtymiseksi C++:sta *Delphi*-ohjelmointiin.

Delphi on *Borland Internationalin* kehittämä *Visual Basicin* tapainen "visuaalinen" sovelluskehitin. Suurimpana erona *Visual Basiciin* on käytetty ohjelmointikieli: *Borland Object Pascal*. *Borland* nousi aikanaan pinnalle nimenomaan *Turbo Pascal* -kääntäjänsä avulla. *Delphi* on *Turbo Pascalista* edelleen kehitetty tuote. Erityisen hyvä *Delphi* on tietokantoihin liittyvässä ohjelmoinnissa. Myös yleisenä ohjelmankehitysvälineenä se on erinomainen. Suurin puute on välineen toimiminen vain yhdessä laiteympäristössä.

Tätä monistetta ei suinkaan ole yritettykään tehdä täydelliseksi *Delphi*-oppaaksi, vaan ainoastaan lähtökohdaksi. Suomenkielisenä oppaana voin ehdottomasti suositella *Ari Becksin Delphi*-kirjaa.

Monisteen malliohjelmat on saatavissa sähköisesti:

Mikroluokka:	hakemisto:	N:\KURSSIT\WINOHJ\DELPHI
WWW:	URL:	http://www.mit.jyu.fi/vesal/kurssit/winohj/delphi

Edellä mainittuun polkuun lisätään aina monisteessa mainittu polku.

Tässä monisteessa ei tulla puuttumaan olio-ohjelmoinnin saloihin, vaan tämä tietous pitää hakea muista oppaista.

Palokassa 4.8.1996 *Vesa Lappalainen*

Monisteen uuteen painokseen on korjattu joitakin pieniä painovirheitä ja vaihdettu muutama komponenttiesimerkki vähän paremmaksi. Edelleen vikana on mm. *WinAapisen* "alkukantainen" ohjelmointityyli. Ohjelma kannattaisi muuttaa huomattavasti enemmän komponenttipohjaiseksi nykyisestä proseduraalisesta muodosta.

Palokassa 15.9.1997 *Vesa Lappalainen*

Monisteen vuoden 2007 painoksesta on vähennetty listauksia, koska ne löytyvät kätevämmin netistä. Lisäksi muutamia lukuja (mm. komponenttien kirjoittaminen) on uudistettu. Vuosien 1997-2007 välillä monisteen korvasi kirja "***Delphi 4 peruskurssi***". Kirja on kuitenkin loppuunmyyty, joten palataan taas monisteeseen.

Palokassa 17.8.2007 *Vesa Lappalainen*

1. Malliohjelma

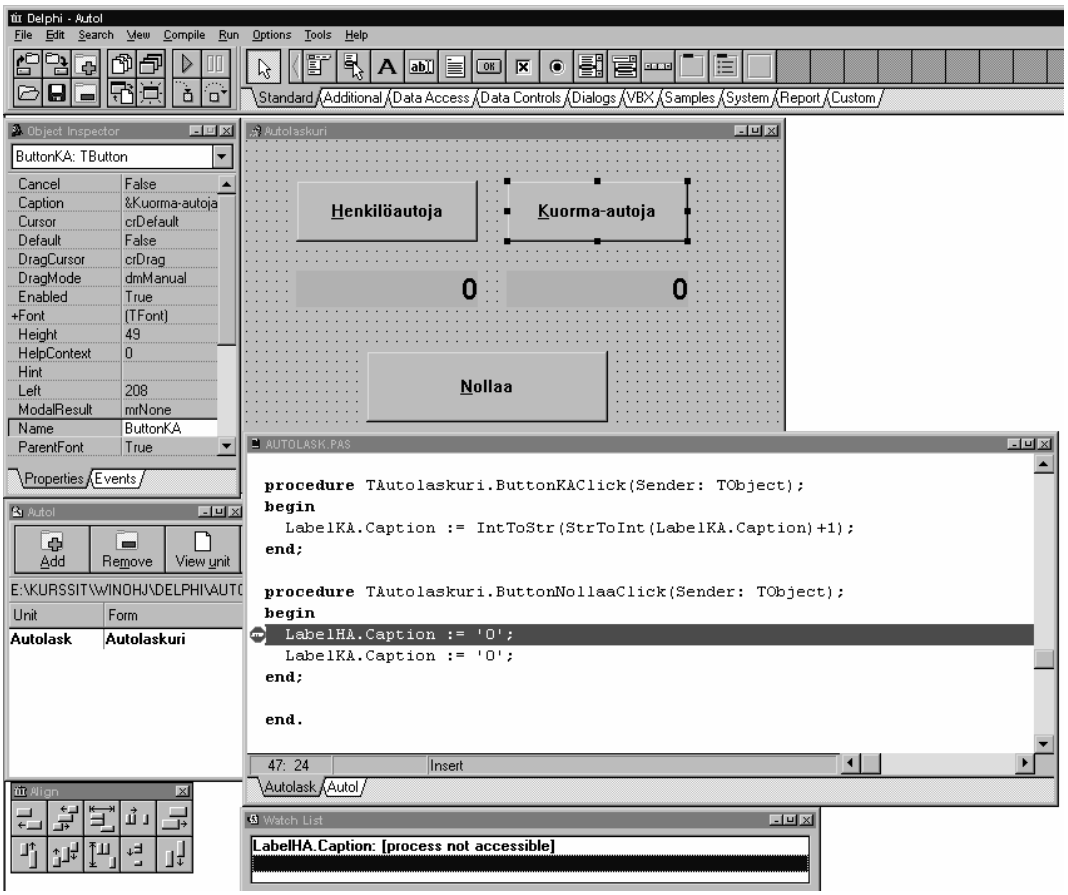
Luvun pääaiheet:

- **yleistä *Delphistä***
- **1. malliohjelma: autolaskuri - ohjelma joka sisältää graafisen käyttöliittymän peruskomponentit**

1.1 Yleistä Delphistä

Delphi on olio-pohjainen visuaalinen sovelluskehitin. Ohjelman kehittäminen on pitkälle erilaisten komponenttien sijoittelua lomakkeille ja sitten komponentteihin liittyvien tapahtumien kirjoittamista.

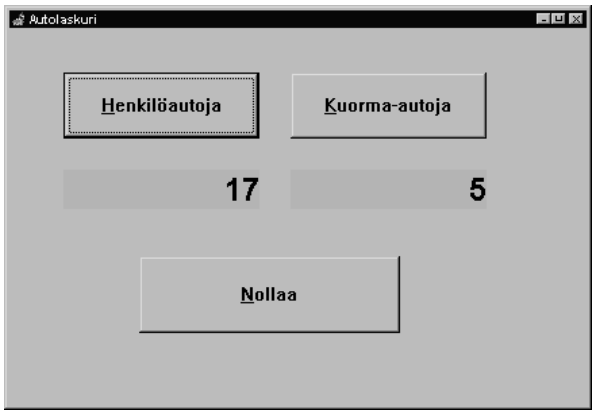
Kukin komponentti, kuten esimerkiksi nappula, menu jne., on itsessään olio. Kun komponentteja laitetaan lomakkeelle saadaan uusi olio. Komponenttioliolla on ominaisuuksia, attribuutteja, kuten esimerkiksi komponentin sijainti lomakkeella (`Left, Top`) komponentin koko (`Width, Height`) jne. Osaa komponenttien ominaisuuksista voidaan muuttaa jo suunnitteluaikana ja osaa vasta ajon aikana.



Kuva 1.1 Delphi, esimerkkiohjelman suunnittelu

1.2 Autolaskuri

Ensimmäisenä malliohjelmana teemme “autolaskurin”, jossa on kaksi nappia, joita painamalla voi lisätä joko henkilöautojen tai kuorma-autojen lukumäärää.



Kuva 1.2 Autolaskuri

1. Käynnistä *Delphi*.
2. Vaihda lomakkeen nimeksi (*Object Inspectorissa* `name`-ominaisuus) Autolaskuri.

1.2.1 Nappuloiden lisääminen

1. Valitse Standard-työkaluvalikosta nappula (`Button`, ): klikkaa kuvaketta kerran
2. Lisää nappula lomakkeeseen: vie hiiri sinne minne haluat nappulan vasemman yläkulman ja paina vasen nappi alas. Rahaa oikea alakulma paikalleen ja päästä hiiren nappi ylös. “Kahvoihin” tarttumalla voit vielä vaihtaa nappulan kokoa.
3. Anna nappulalle nimeksi `ButtonHA`.
4. Vaihda nappulan tekstiksi (`caption`-ominaisuus) &Henkilöautoja. Tässä &-merkki tarkoittaa sitä, että seuraava kirjain tulee alleviivatuksi ja sama toiminto voidaan suorittaa painamalla Alt-H tai joissakin tapauksissa jopa pelkästään H (jollei H muuten voi merkitä lomakkeella mitään).
5. Monista nappula: valitse nappula ja paina Ctrl-Ins. Paina Shift-Ins ja raahaa uusi nappula oikealle kohdalleen.
6. Nimeä uusi nappula `ButtonKA` ja vaihda tekstiksi &Kuorma-autoja.

1.2.2 Laskuriruutujen lisääminen

1. Lisää lomakkeelle “laskuriruutu” nappulan alapuolelle käyttäen vakiotekstiä (`Label`,).
2. Laita tekstin `AutoSize`-ominaisuus epätodeksi: Tuplaklikkaa `True` -sanaa. Näin laskuriruutu saadaan pysymään aina samankokoisena. Joissakin tapauksissa on toisaalta mukavaa että ruutu muuttuu tekstin koon mukaan.
3. Kun olet lisännyt tekstin, saat sen saman levyiseksi kuin nappulankin valitsemalla molemmat aktiiviseksi (shift+klikkaus vasemmalla kumpaankin) ja sitten hiiren oikeasta näppäimestä aukeavasta menusta Size ja ruksi esim. ruutuun Grow to largest leveyden (*Width*) kohdalla.
4. Nimeä teksti `LabelHA` ja tekstiksi (`caption`) 0.
5. Laitetaan teksti ruudun oikeaan reunaan: `Alignment`-ominaisuus: `taRightJustify`
6. Vaihda tekstin väriksi vaikkapa `clAqua`.

7. Vaihda tekstin fontiksi vaikkapa Arial Bold 18: tuplaklikkaa (`Tfont`)-tekstiä.
8. Muuta tekstiruudun korkeus oikeaksi.
9. Monista tekstiruutu ja siirrä uusi ruutu Kuorma-autoja -nappulan alle.
10. Nimeä uusi tekstiruutu: `LabelKA`.

1.2.3 Talletus

1. Lisää vielä nappula, jonka nimeksi annat `ButtonNollaa` ja tekstiksi &Nollaa.
2. Talleta projekti: Alt-F v. Kysymykseen Save Unit As vastataan esim. autolask (tämä tulee sen tiedoston nimeksi, jossa itse ohjelmakoodi on). Save Project As vastataan esim. autol (projekti tallentuu nyt nimellä autol.dpr ja itse käännetystä ohjelmasta tulee autol.exe).

1.2.4 Kääntäminen ja ajaminen

Ohjelma on nyt toimintakuntoinen, mutta se ei vielä tee mitään. Voimme kuitenkin kokeilla miltä valmis ohjelma näyttäisi:

1. Käännä ja aja ohjelma: F9

1.2.5 Ohjelmakoodin lisääminen

Kun olemme laittaneet komponentteja lomakkeelle, *Delphi* on lisännyt koko ajan tiedostoon autolask.pas ohjelmakoodia lomakkeen `Autolaskuri`-olion määrittelevään `TAutolaskuri`-luokkaan (`class`).

Nappuloiden toiminnallisuutta vastaavan koodin lisääminen on ohjelmoijan tehtävä. Onneksi *Delphi* tekee tästäkin suurimman osan:

1. Tuplaklikkaa Henkilöautoja-nappulaa. Nyt aukeaa koodi-ikkuna, jossa on valmiina *Pascal*-kielinen tapahtumankäsittelijän esittely tapahtumalle, joka tulee kun painetaan nappulaa nimeltä `ButtonHA`:

```
procedure TAutolaskuri.ButtonHAClick(Sender: TObject);
begin
_
end;
```

2. Kursori on valmiina paikassa, johon oma koodi kirjoitetaan. Me haluamme että nappulaa painettaessa `LabelHA`:ssa oleva lukema lisääntyy yhdellä. Tämä voitaisiin kirjoittaa: `LabelHA.Caption := LabelHA.Caption + 1;` mutta valitettavasti `LabelHA.Caption` on tekstiä eikä sitä voi numeerisesti lisätä (kuten *Visual Basicin* `Variant`-tyyppiä voi). Siispä kirjoitamme koodin:

```
 LabelHA.Caption := IntToStr(StrToInt(LabelHA.Caption)+1);
```

3. Koodi kannattaa saman tien laittaa leikekirjaan, koska sehän tulee lähes samanlaisena nappulaan `ButtonKA`.
4. Lisää vastaava koodi oikein muutettuna nappulaan `ButtonKA`. Huom! Jos et edellä huomannut laittaa koodia leikekirjaan, löytyy edellinen koodi samasta koodi-ikkunasta hieman ylempää ja voit hakea sen kuin missä tahansa editorissa.
5. Lisää vielä koodi nappulaan `ButtonNollaa`. Nyt koodiksi riittää

```
  LabelHA.Caption := '0';
  LabelKA.Caption := '0';
```

6. Käännä ja aja ohjelma.

1.2.6 Valmis ohjelma

Seuraavana vielä täydelliset listaukset valmiin malliohjelman eri tiedostoista (hakemistossa autol). Itse kirjoitetut tai muutetut osat on varjostettu:

autol.dpr - projetitiedosto

```
program Autol;

uses
  Forms,
  Autolask in 'AUTOLASK.PAS' {Autolaskuri};

{$R *.RES}

begin
  Application.CreateForm(TAutolaskuri, Autolaskuri);
  Application.Run;
end.
```

autolask.pas - autolaskuri-lomakeluokan määrittely ja toteutus

```
unit Autolask;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TAutolaskuri = class(TForm)
    ButtonHA: TButton;
    ButtonKA: TButton;
    LabelHA: TLabel;
    LabelKA: TLabel;
    ButtonNollaa: TButton;
    procedure ButtonHAClick(Sender: TObject);
    procedure ButtonNollaaClick(Sender: TObject);
    procedure ButtonKAClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Autolaskuri: TAutolaskuri;

implementation

{$R *.DFM}

procedure TAutolaskuri.ButtonHAClick(Sender: TObject);
begin
  LabelHA.Caption := IntToStr(StrToInt(LabelHA.Caption)+1);
end;

procedure TAutolaskuri.ButtonKAClick(Sender: TObject);
begin
  LabelKA.Caption := IntToStr(StrToInt(LabelKA.Caption)+1);
end;
```

```
procedure TAutolaskuri.ButtonNollaaClick(Sender: TObject);
begin
  LabelHA.Caption := '0';
  LabelKA.Caption := '0';
end;

end.
```

Seuraavassa lomakkeen listauksessa tummennetut osat ovat niitä, joita on muutettu *Object Inspectorissa*. Luonnollisesti kunkin komponentin paikkaa ja kokoa on muutettu oletuksesta, mutta tämä on tehty siirtämällä komponenttia hiirellä.

autolask.dfm - autolaskuri-lomake, komponenttien ominaisuudet

```
object Autolaskuri: TAutolaskuri
  Left = 190
  Top = 90
  Width = 435
  Height = 300
  Caption = 'Autolaskuri'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 16
  object LabelHA: TLabel
    Left = 40
    Top = 104
    Width = 145
    Height = 29
    Alignment = taRightJustify
    AutoSize = False
    Caption = '0'
    Color = clAqua
    Font.Color = clBlack
    Font.Height = -24
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ParentColor = False
    ParentFont = False
  end
  object LabelKA: TLabel
    Left = 208
    Top = 104
    Width = 145
    Height = 29
    Alignment = taRightJustify
    AutoSize = False
    Caption = '0'
    Color = clAqua
    Font.Color = clBlack
    Font.Height = -24
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ParentColor = False
    ParentFont = False
  end
  object ButtonHA: TButton
    Left = 40
    Top = 32
    Width = 145
    Height = 49
    Caption = '&Henkilöautoja'
    TabOrder = 0
    OnClick = ButtonHAClick
  end
```

```
  object ButtonKA: TButton
    Left = 208
    Top = 32
    Width = 145
    Height = 49
    Caption = '&Kuorma-autoja'
    TabOrder = 1
    OnClick = ButtonKAClick
  end
  object ButtonNollaa: TButton
    Left = 96
    Top = 168
    Width = 193
    Height = 57
    Caption = '&Nollaa'
    TabOrder = 2
    OnClick = ButtonNollaaClick
  end
end
```

Tehtävä 1.1 Polkupyörät

Lisää ohjelmaan myös polkupyörien laskeminen.

1.3 Näkymättömät komponentit

Edellisessä esimerkissä lisättiin vain näkyviä komponentteja. *Delphissä* on myös suuri joukko näkymättömiä komponentteja.

1.3.1 `Timer` - ajastetut tapahtumat

Lisätään vaikkapa aluksi auton kuva, joka ajaa ruudun vasemmasta laidasta oikeaan laitaan.

1. Lisää `Image`-komponentti ruudun vasempaan alalaitaan (löytyy additional sivulta).
2. Valitse `Image`-komponentin `Picture`-ominaisuus ja lataa siihen vaikkapa kuvaksi hauto.bmp. (n:\kurssit\winohj\delphi\autol\hauto.bmp).
3. nyt kannattaa laittaa `AutoSize`-ominaisuus päälle
4. vaihda komponentin nimeksi vaikkapa `ImageHA.`

Jotta auton kuva liikkuisi, pitäisi `ImageHA`-olion paikkaa muuttaa tietyn välein. Tietyn aikavälein tapahtuvia tapahtumia saadaan `Timer`-komponentilta.

1. Lisää lomakkeelle mihin tahansa kohtaan `Timer`-komponentti (löytyy system sivulta). Paikalla ei ole väliä, koska komponentti EI ole näkyvissä ohjelman ajon aikana.
2. vaihda nimeksi vaikkapa `TimerHA`
3. vaihda tapahtumaväliksi esim. 100 (=100 ms).
4. tuplaklikkaa ajastinta ja lisää koodi-ikkunaan koodi:

```
procedure TAutolaskuri.Timer1Timer(Sender: TObject);
begin
  ImageHA.Left := ImageHA.Left + 1;
end;
```

Tehtävä 1.2 Edestakaisin

Muuta ohjelmaa siten, että auton kuva kulkee ruudussa edestakaisin.

1.3.2 Menut

Lisätään ohjelmaan vielä päämenu.

1. Lisää lomakkeelle `MainMenu`-komponentti (standard -sivu). Paikalla ei jälleenkään ole väliä, sillä menu tulee aina lomakkeen yläreunaan.
2. tuplaklikkaa `Menu`-oliota
3. Kirjoita seuraava menusysteemi:

```
&File      &Options     H&elp
E&xit      &Colors      &About
```

4. Laita vielä lisäksi `Exit`-kohtaan pikavalinta Ctrl-X.
5. tuplaklikkaa `Exit`-valintaa ja lisää tapahtumaksi koodi

```
procedure TAutolaskuri.Exit1Click(Sender: TObject);
begin
  Close;
end;
```

Tehtävä 1.3 H&elp

Miksi laitoimme menuun `H&elp` eikä `&Help`? Mikä tässäkin valinnassa on huonoa?

1.3.3 Valmiit lomakkeet

Delphissä on valmiina joukko yleisimpiä dialogeja: tiedoston avaus ja talletus, fonttien valinta, värin valinta, tulostaminen sekä etsintä ja korvaus.

Lisäämme seuraavaksi mahdollisuuden taustavärin vaihtamiseksi.

1. Lisää lomakkeelle väridialogi (dialogs-sivu). Paikalla ei ole väliä.
2. Laita dialogin nimeksi vaikkapa `ColorDialogTausta`.
3. Lisää menun `Colors`-kohdan tapahtumaksi koodi:

```
procedure TAutolaskuri.Colors1Click(Sender: TObject);
begin
  ColorDialogTausta.Color := Autolaskuri.Color;
  if ( not ColorDialogTausta.Execute ) then exit;
  Autolaskuri.Color := ColorDialogTausta.Color;
end;
```

Tehtävä 1.4 Muidenkin komponenttien värin vaihto

Muuta ohjelmaa siten, että voit muuttaa kaikkien muidenkin komponenttien värin (voit käyttää samaa dialogia kaikille komponenteille).

1.3.4 Omat dialogit

Oikeassa ohjelmassa on harvoin vain yksi ikkuna. Lisäämme esimerkin vuoksi vielä ohjelmaamme itse tehdyn `About`-dialogin:

1. Luo uusi lomake (File|New form| Blank form).
2. Vaihda lomakkeen nimeksi `FormAbout` ja otsikoksi Tietoja autolaskurista.
3. Lisää vakioteksti (`Label`) jonka nimeksi vaikkapa `LabelAbout` ja `WordWrap` -ominaisuus todeksi. Tekstiksi laitetaan sitten mikä tahansa ohjelman toimintaa yms. kuvaava teksti.
4. Lisää vielä haluamiasi koristeita, kuten esim. bittikarttoja (vrt. liikkuvan auton lisääminen).
5. Lisää vielä nappula, jonka nimeksi `ButtonOK` ja tekstiksi OK sekä `Default`-ominaisuus todeksi.
6. Lisää OK-nappulan koodiksi:

```
procedure TFormAbout.ButtonOKClick(Sender: TObject);
begin
  Close;
end;
```

Lomake on nyt valmis, mutta siihen ei viitata varsinaisesta lomakkeesta.

1. Talleta `About`-lomakkeen tiedosto nimelle about.pas
2. Lisää varsinaisen ohjelman menunvalintaa `About` seuraava koodi:

```
procedure TAutolaskuri.About1Click(Sender: TObject);
begin
  FormAbout.Show;
end;
```

3. Kokeile ajaa ohjelmaa. Todennäköisesti saat virheilmoituksen:

```
Error 3: Unknown identifier
```

4. Kursori on sanan `FormAbout` alussa. Tämä johtuu siitä, ettei Autolaskuri-lomakkeen toteutuksessa ole kerrottu mitään `About`-lomakkeesta. Korjataan vielä tämä vika.
5. Siirry autolask.pas -tiedostossa aivan alkuun. Sieltä löytyy `uses`-lause. Lisää tämän lauseen loppuun tieto siitä että käytetään myös `About`-lomaketta.

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus, About;
```

6. Kokeile nyt ohjelmaa.

Tehtävä 1.5 Modaalinen dialogi

Muuta rivi `FormAbout.Show;` muotoon `FormAbout.ShowModal`; Mitä eroa on nyt ohjelman toiminnassa?

Tehtävä 1.6 Liikkuva auto myös toisessa dialogissa

Lisää liikkuva auto myös `About`-dialogiin.

1.4 Ohjelmakoodin korjailu

Ohjelmakoodi on aivan tavallista tekstiä ja sitä voidaan muokata kuten millä tahansa tekstieditorilla. Seuraavat seikat on kuitenkin syytä pitää mielessä:

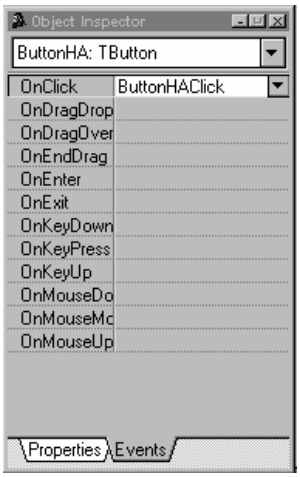
1. Muuta komponenttien nimiä vain *Object Inspectorissa*. Muuten lomake ja ohjelmakoodi eivät pysy synkronissa.
2. Jos haluat hävittää jonkin tapahtuman käsittelijän, poista `begin-end` -parin välissä oleva koodi. *Delphi* hävittää sitten loput seuraavan talletuksen tai käännöksen yhteydessä. Kukin metodi on nimittäin esitelty sekä luokan määrittelyssä, että itse koodissa.

1.5 Muut tapahtumat

1.5.1 Saman tapahtuman käyttö toisessa komponentissa

Olemme voineet kirjoittaa tapahtuman käsittelijän koodin tuplaklikkaamalla komponenttia. Näin voimme kirjoittaa kuitenkin vain komponentin oletustapahtuman käsittelijän. Kullakin komponentilla on lukuisia muitakin tapahtumia. Nämä muut tapahtu-

mat löytyvä *Object Inspectorissa* `Events`-sivulta. Seuraavassa esimerkki `ButtonHa`:n mahdollisista tapahtumista:



Kuva 1.3 Events-sivu

Tapahtuma päästään kirjoittamaan tuplaklikkaamalla tapahtuman nimeä (nimen paikkaa jos nimi on tyhjä). Tapahtuma voidaan laittaa myös samaksi jonkin toisen tapahtuman kanssa jolla on sama parametrilista.

Tehtävä 1.7 Laskenta tapahtumaan myös laskurista

Muuta ohjelmaa (kirjoittamatta lisää koodia) siten, että myös laskurikentän `LabelHA` tai `LabelKA` painaminen lisää vastaavaa laskuria. Laita vielä liikkuvan kuvan painaminen lisäämään henkilöautojen lukumäärää.

1.5.2 Vedä ja pudota (drag and drop, DaD)

"Nykyaikaisessa" suorakäyttöliittymässä olion vetäminen ja pudottaminen on muotia. Lisätään omaan ohjelmaamme vielä ominaisuus, jossa käyttäjä voi "tarttua" henkilöauton kuvaan ja pudottaa sitten sen jomman kumman laskuri-ikkunan päälle. Tällöin vastaava laskuri lisääntyy vaikkapa 10:llä.

1. Muuta henkilöauton kuvan `DragMode` -ominaisuus arvoon `dmAutomatic`.
2. Kokeile ajaa ohjelmaa. Nyt voi tarttua henkilöauton kuvaan, muttei sitä voi vielä pudottaa mihinkään.
3. Lisää `LabelHA`:han tapahtuma (`DragOver`), jolla hyväksytään siihen pudottaminen:

```
procedure TAutolaskuri.LabelHADragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;
end;
```

4. Kokeile ajaa ohjelmaa. Nyt pudottaminen on sallittua `LabelHA`:n päälle, muttei `LabelKA`:n päälle.
5. Laita sama tapahtuma `LabelKA`:n `DragOver` tapahtumaksi (kirjoittamatta koodia).
6. Seuraavan vaiheen helpottamiseksi kirjoitetaan aliohjelma, jonka avulla tekstikenttää voidaan lisätä yhdellä. Samalla vanhat laskut voidaan muuttaa käyttä-

mään tätä aliohjelmaa. Koodi kirjoitetaan vaikkapa ennen kaikkia tapahtuman käsittelijöitä:

```
procedure lisaa(lab:TLabel; n:integer);
begin
  lab.Caption := IntToStr(StrToInt(lab.Caption)+n);
end;

procedure TAutolaskuri.ButtonHAClick(Sender: TObject);
begin
  lisaa(LabelHA,1);
end;
```

7. Lisää käsittelijä `LabelHA`:n pudotuksen vastaanottamiselle (`DragDrop`):

```
procedure TAutolaskuri.LabelHADragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  lisaa(Sender as TLabel,10);
end;
```

8. Laita sama tapahtuma `LabelKA`:n `DragDrop` tapahtumaksi (kirjoittamatta koodia).
9. Jos laskurikentät tulevat vielä hyväksymään muistakin komponenteista tulevia pudotuksia, voidaan em. koodi modifioida:

```
procedure TAutolaskuri.LabelHADragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if ( Source = ImageHA ) then lisaa(Sender as TLabel,10);
end;
```

Tehtävä 1.8 Muitakin lisäysmääriä

Lisää ohjelmaan joukko numeroikkunoita (esim. 1,2,3,4,5), joihin kuhunkin voidaan tarttua ja vetää sitten laskuri-ikkunan päälle. Kun numero pudotetaan, lisääntyy laskuri-ikkunan arvoa vastaavalla numerolla (itse kirjoitettua ohjelmakoodia n. 1 rivi edelliseen lisää).

Tehtävä 1.9 Liikkuvan kuvan siirto toiseen paikkaan

Lisää ohjelmaan ominaisuus: jos tartut henkilöauton kuvaan ja pudotat sen lomakkeen päälle, niin auto siirtyy tähän kohtaan lomaketta (ohjelmakoodia korkeintaan 4 itse kirjoitettua riviä).

Tehtävä 1.10 Fontin ja värin vaihto

Lisää ohjelmaan `PopupMenu` kullekin nappulalle ja laskurille, jotta voit vaihtaa nappuloiden ja laskureiden fonttia ja laskureiden väriä. (Olennaisesti 6 erilaista itse kirjoitettua riviä lisää, käytännössä 9, koska nappuloiden fontti ja laskureiden fontti tarvitsee oman koodinsa).

Tehtävä 1.11 Komponentin paikan vaihtaminen

Lisää ohjelmaan mahdollisuus tarttua komponenttiin (esim. nappulaan) ja siirtää se uuteen paikkaan.

2. Object Pascalin ja C++:n eroja

Luvun pääaiheet:

- ***Delphin Object Pascalin* perusrakenne**
- **parametrin välitys**
- **silmukat**
- **taulukot**
- **`class`**
- **`RTTI` - ajonaikainen tyypin tunnistus**
- **dynaamisesti luotavat kontrollit**
- **säikeet**
- **tämän luvun esimerkit ovat alihakemistossa moniste\esimerki**

2.1 Perusrakenne

Jatkossa käytämme *Delphin Object Pascal* -kielestä pelkästään nimeä *Pascal*. Tästä huolimatta tämän monisteen tekstiä ei tule sotkea *standardi-Pascaliin*, johon *Delphin Object Pascalissa* on lisätty huomattavasti omia lisäpiirteitä, mm:

- merkkijonot
- luokat
- moduulit (`unit`)

Oletamme että lukija tuntee suhteellisen hyvin vähintään C-kielen, mieluummin perusteet *C++*:stakin.

Aloitetaan erojen selvittäminen lyhyellä konsoli -esimerkkiohjelmalla, joka lukee kaksi kokonaislukua ja tulostaa niistä suuremman, lukujen keskiarvon ja luvut suuruusjärjestyksessä. Aluksi sama pääohjelma *C++*:lla ja *Delphi 6.0:lla*.

esim1.cpp - C++ pääohjelma

```
#include <iostream.h>
#include "ali.hpp"

int main(void)
{
  int a,b;
  cout << "Anna kaksi lukua välilyönnillä"
          " erotettuna>";
  cin >> a >> b;
  cout << "Suurempi luvuista on "
       <<  bigger(a,b) << endl;
  cout << "Lukujen keskiarvo on "
       << average(a,b) << endl;
  if ( a > b ) {
    swap(a,b);
    cout << "Luvut järjestyksessä ovat "
         << a << " " << b << endl;
  }
  else
    cout << "Luvut olivat järjestyksessä"
         << endl;
  return 0;
}
```

esim1.dpr - Delphi 6.0 pääohjelma

```
program Esim1;

uses Ali;

var a,b:integer;
begin
  write('Anna kaksi lukua välilyönnillä ',
        'erotettuna>');
  readln(a,b);
  writeln('Suurempi luvuista on ',
          bigger(a,b));
  writeln('Lukujen keskiarvo on ',
          average(a,b):5:2);
  if ( a > b ) then begin
    swap(a,b);
    writeln('Luvut järjestyksessä ovat ',
            a,' ',b);
  end
  else
    writeln('Luvut olivat järjestyksessä');
end.
```

2.1.1 Peruserot

- ei erotella isoja ja pieniä kirjaimia `begin = Begin = BEGIN`
- pääohjelma (moduuli) alkaa aina sanalla **`program`**
- pääohjelma loppuu aina pisteeseen (**`end.`**)
- lohkot suljetaan **`begin-end`** -sanojen väliin
- muuttujat esitellään lohkojen ulkopuolella
- muuttujien esittely alkaa **`var`** -sanalla
- muuttujien esittelyssä tulee ensin muuttujan nimi (nimet) ja sitten muuttujan tyyppi
- merkkijonot suljetaan yksinkertaisiin lainausmerkkeihin
- **`if`** -lauseeseen kuuluu **`then`** -sana
- **`if`** -lauseen ehto ei välttämättä tarvitse kaarisulkuja, paitsi yhdistetyssä ehdossa

```
if ( 0 < a ) and ( a < 10 ) then
```

- **`else`** -sanan edessä EI saa olla puolipistettä
- aliohjelmakirjastot esitellään kääntäjälle **`uses`** -lauseella
- em. käännöksessä etsitään automaattisesti ali.pas -tiedostoa, joka tarvittaessa käännetään ja linkitetään mukaan

Tehtävä 2.12 Sama ohjelma C-kielellä

Kirjoita vastaava ohjelma C-kielellä.

2.1.2 Parametrin välitys

Edellisen esimerkin aliohjelmat on kirjoitettu omaan tiedostoonsa, joka *C++*:ssa pitää muistaa linkittää mukaan. Otsikkotiedostossahan on tiedot vain kääntämistä varten. *Pascalin* `unitissa` ovat sekä otsikkotiedot päämoduulin kääntämiseksi, että varsinainen toteutus. Tarvittaessa voidaan tietysti jakaa käännettyä moduulia (ali.dcu, *Delphi Compiled Unit*).

esim1.hpp - C++ otsikkotiedosto

```
#ifndef ALI_HPP
#define ALI_HPP
double average(int a, int b);
int bigger(int a, int b);
void swap(int &a,int &b);
#endif
```

ali.cpp - C++ aliohjelmat

```
#include "ali.hpp"

double average(int a, int b)
{
  return (a+b)/2.0;
}

int bigger(int a, int b)
{
  if ( a > b ) return a;
  return b;
}
```

ali.pas - Pascal aliohjelmat

```
unit Ali;

interface

  function average(a,b:integer):real;
  function bigger(a,b:integer):integer;
  procedure swap(var a,b:integer);

implementation

function average(a,b:integer):real;
begin
  Result := (a+b)/2.0;
end;

function bigger(a,b:integer):integer;
begin
  Result := b;
  if ( a > b ) then Result := a;
end;

{ Vaihdetaan luvut keskenään }
procedure swap(var a,b:integer);
```

```
// Vaihdetaan luvut keskenään
void swap(int &a,int &b)
{
  int t;
  t = a; a = b; b = t;
}
```

```
var t:integer;
begin
  t := a; a := b; b := t;
end;
end. { Unitin lopetus }
```

- aliohjelmamoduulissa (`unit`) on **`interface`** -osa, jonka alla on lueteltu moduulin ulkopuolelle näkyvät määritykset
- **`implementation`** -osassa on varsinainen toteutus
- moduuli lopetetaan **`end.`** (huom. piste)
- moduulissa voi olla alustusosa **`initialization`** `alustus;` **`end.`**, joka suoritetaan kun moduuli ladataan muistiin
- voi lisäksi olla **`finalization`** -osa, joka suoritetaan kun moduuli poistuu muistista
- aliohjelman otsikkorivi loppuu AINA puolipisteeseen **`;`**
- sijoitusoperaattorina on **`:=`**
- yhtäsuuruusvertailu tehdään operaattorilla **`=`**
- kommenttisulkuina on **`{}`** tai vaihtoehtoisesti **`(* ... *)`**
- rivikommenttina toimii myös **`//`**
- aliohjelmia on kaksi tyyppiä: **`procedure`** ja **`function`**.
- `procedure` vastaa *C:*n `void` -funktioita
- parametrilistan erotin on puolipiste **`;`**
- parametrilistoissa voidaan kirjoittaa useita muuttujia pilkulla erotettuna ennen parametrin tyypin ilmaisemista
- reaalilukutyyppi on **`real`**, voidaan käyttää myös **`double`**.
- funktion paluuarvo sijoitetaan funktion nimeen. Sijoituksia voi olla useita, joista viimeisenä tehty jää voimaan.
- paluuarvolle on myös lokaali muuttuja **`Result`**, jolle voidaan sijoittaa ja jota voidaan käyttää lausekkeen oikeallakin puolella

```
function bigger(a,b:integer):integer;
begin
  Result := b;
  if ( a > Result ) then Result := a;
end;
```

- funktio ja aliohjelma voidaan lopettaa **`exit`**-lauseella, jolloin funktion arvoksi jää viimeisen sijoituksen arvo
- aliohjelmilla on kahta eri parametrityyppiä: *arvoparametrit* (*C:*ssä on vain näitä, *call by value*) ja *muuttujaparametrit* (*call by reference*). Muuttujaparametrit ovat samoja kuin *C*:ssä parametrin välitys osoittimien avulla. *C++*:ssa vastine on referenssi (**`&`**). Muuttujaparametrit ilmaistaan parametrilistassa **`var`** -sanalla.
- parametritonta aliohjelmaa kutsutaan ilman sulkuja, esimerkiksi: `writeln;` Tosin Delphissä saa myös käyttää sulkuja jos sillä haluaa korostaa että kyseessä on aliohjelmakutsu.
- aliohjelmien sisään voidaan kirjoittaa omia "apu"aliohjelmia, jotka voivat käyttää "isäntänsä" muuttujia ilman parametrin välitystä

2.1.3 Silmukat ja taulukot

Seuraavassa esimerkissä on ohjelma, joka ensin tekee viisipaikkaisen kokonaislukutaulukon:

0	1	2	3	4
0	3	6	9	12

Sitten ohjelma laskee, montako taulukon alkiota voidaan ottaa mukaan, ilman että summa ylittää vielä 10. Lopuksi tulostetaan ko. alkiot takaperin:

```
3 lukua mahtuu alle 10 näiden summa on 9
Luvut on: 6 3 0
```

silmu.cpp - esimerkki silmukoista

```
#include <iostream.h>

const int TKOKO=5;
const int RAJA=10;




void alusta(int luvut[], int n, int kasvu)
/* Alustetaan taulukko sarjalla
   0,kasvu,2*kasvu... */
{
  int i,luku=0;
  for (i=0; i<n; i++) {
    luvut[i] = luku;
    luku += kasvu;
  }
}



int montako_mahtuu(const int luvut[], int n,
                   int raja)
/* Mihin asti lukujen summa ei ylitä rajaa
*/
{
  int i=0,summa=0;
  do {
    summa += luvut[i];
  } while ( summa < raja && ++i < n );
  return i;
}


int summaa(const int luvut[], int n)
{
  int i=0,summa=0;
  while ( i < n ) {
    summa += luvut[i];
    i++;
  }
  return summa;
}



void tulosta(ostream &os,
             const int luvut[],int n)
/* Tulostaa taulukon nurinpäin */
{
  int i;
```

silmu.dpr - esimerkki silmukoista

```
program Silmu;

const TKOKO=5;
const RAJA=10;

{ Seuraavalla korvataan C:n ++i }
function esi_lisaa(var i:integer):integer;
begin inc(i); esi_lisaa := i; end;

procedure alusta(var luvut:array of integer;
                 n,kasvu:integer);
{ Alustetaan taulukko sarjalla
  0,kasvu,2*kasvu...                        }
var i,luku:integer;
begin
  luku := 0;
  for i:=0 to n-1 do begin
    luvut[i] := luku;
    inc(luku,kasvu);
  end;
end;

function montako_mahtuu(const luvut:array of
           integer; n,raja:integer):integer;
{ Mihin asti lukujen summa ei ylitä rajaa }
var i,summa:integer;
begin
  i := 0; summa := 0;
  repeat
    summa := summa + luvut[i];
  until ( summa >= raja ) or
        ( esi_lisaa(i) >= n );
  montako_mahtuu := i;
end;

function summaa(const luvut:array of integer;
                n:integer):integer;
var i,summa:integer;
begin
  i := 0; summa := 0;
  while ( i < n ) do begin
    inc(summa,luvut[i]);
    inc(i);
  end;
  summaa := summa;
end;

procedure tulosta(var f:textfile;
   const luvut:array of integer; n:integer);
{ Tulostaa taulukon nurinpäin }
var i:integer;
begin
```

```
  for (i=n-1; i>=0; i--)
    os << luvut[i] << " ";
  os << endl;
}

int main(void)
{
  int luvut[TKOKO],n;
  alusta(luvut,TKOKO,3);
  n = montako_mahtuu(luvut,TKOKO,RAJA);
  cout << n << " lukua mahtuu alle " << RAJA
       << " näiden summa on "
       << summaa(luvut,n) << endl;
  cout << "Luvut on: ";
  tulosta(cout,luvut,n);
  return 0;
}
```

```
  for i:=n-1 downto 0 do
    write(f,luvut[i],' ');
  writeln(f);
end;

{ Pääohjelma: }
var luvut : array[0..TKOKO-1] of integer;
    n : integer;
begin
  alusta(luvut,TKOKO,3);
  n := montako_mahtuu(luvut,TKOKO,RAJA);
  writeln(n,' lukua mahtuu alle ',RAJA,
          ' näiden summa on ',summaa(luvut,n));
  write('Luvut on: ');
  tulosta(output,luvut,n);
end.
```

Huomioita silmukoiden ja taulukoiden eroista:

- **`while do`** kuten *C*:ssä **`while`**
- **`do-while`**:n tilalle **`repeat until`**, jossa on lopettamisehto, ei jatkamisehto kuten *C:*ssä, eli ehto vastaavan C-ehdon negaatio
- **`for`** -silmukka on huomattavasti rajoittuneempi kuin *C:*ssä. Silmukkalaskurin arvo ei ole välttämättä tunnettu silmukan jälkeen, eikä laskurin arvon muuttaminen kesken silmukan välttämättä katkaise silmukkaa
- **`for-to`** osaa lisätä vain yhdellä
- alaspäin laskemista varten on oma **`for-downto`**
- taulukoiden esittelyssä esitellään `alaraja..yläraja`
- itse asiassa taulukon indeksityppinä voi olla mikä tahansa ordinaalityyppi (numeroituva tyyppi)
- moniulotteinen taulukko on taulukko taulukoista:

```
var T: array[Boolean] of array[1..10] of array[3..6] of Real;
{ on sama kuin: }
var T: array[Boolean,1..10,3..6] of Real;
{ Alkioon viitataan joko }
  r := T[True][3][5];
{tai}
  r := T[True,3,5];
```

- avoimen taulukon `luvut:array of integer` koko saataisiin itse asiassa selville kutsuilla:

```
Low(luvut);     { aina 0 riippumatta varsinaisen taulukon alarajasta  }
High(luvut);    { 4, eli alkuperäisen taulukon koko-1                 }
SizeOf(luvut);  { 10 tai 20 riippuen onko 16 vai 32-bittinen kääntäjä }
```

- esimerkissä kuitenkin käytetään parametrina tuotua lukumäärää; näin voidaan laskea tuloksia myös osataulukoille. Tietysti pitäisi aina testata onko `n <= High(luvut)+1`, esimerkissä on pyritty vain matkimaan mahdollisimman paljon viereistä C-koodia.
- avointa taulukkoa (*open array*) voidaan kutsua myös vakiotaulukolla:

```
tulosta(output,[1,2,4,8,16],5);
```

- jollei olisi tehty apufunktiota `esi_lisaa`, olisi `repeat` -silmukka pitänyt kirjoittaa esimerkiksi:

```
repeat { repeat edelleen väärin, koska jos n=0, viitataan lait.alkioon }
  summa := summa + luvut[i];
  if ( summa >= raja ) then break;
  i := i + 1;
until ( i >= n );
```

- `break` -proseduurissa on sama vika kuin *C*:n `break` -lauseessakin: vain yksi taso voidaan katkaista. Tätä voidaan kiertää joko aliohjelman `exit` -lauseella tai `goto` -lauseella

Tehtävä 2.13 Avoimen taulukon ylärajan tarkistus

Muuta silmu.dpr -ohjelmaa siten, että kussakin aliohjelmassa tarkistetaan, ettei taulukon yläarajaa ylitetä.

Miten em. muutoksen jälkeen kutsu `tulosta(output,[1,2,4,8,16],5);` voitaisiin korvata laskematta itse vakiotaulukon kokoa?

2.1.4 case-lause

- **`case`** -lause poikkeaa hieman *C*-kielen **`switch`** lauseesta, tavallisimmassa tapauksessa se on jopa helpompi käyttää:

caseof.c - esimerkki switch -lauseesta

```
#include <stdio.h>

int main(void)
{
 int tunnit;
 for ( tunnit=1; tunnit<=24; tunnit++ ) {
  printf("%2d: ",tunnit);
  switch ( tunnit ) {
   case  1: case 2: case 3: case 4: case 5:
   case  6: printf("Nukutaan\n");    break;
   case  7: printf("Herätys\n");     break;
   case  8: printf("Töihin\n");      break;
   case  9: case 10: case 11:
   case 13: case 14: case 15:
   case 16: printf("Tehdään töitä\n");
            break;
   case 12:
   case 18: printf("Syödään\n");     break;
   default: printf("Huilaillaan\n"); break;
  }
 }
 return 0;
}
```

caseof.dpr - esimerkki case-of -lauseesta

```
program Caseof;

{ Pääohjelma: }
var tunnit : integer;
begin
  for tunnit:=1 to 24 do begin
    write(tunnit:2,': ');
    case tunnit of
      1..6  : writeln('Nukutaan');
      7     : writeln('Herätys');
      8     : writeln('Töihin');
      9..11,
      13..16: writeln('Tehdään töitä');
      12,18 : writeln('Syödään');
      else    writeln('Huilaillaan');
    end; { case:lle oma end! }
  end;
end.
```

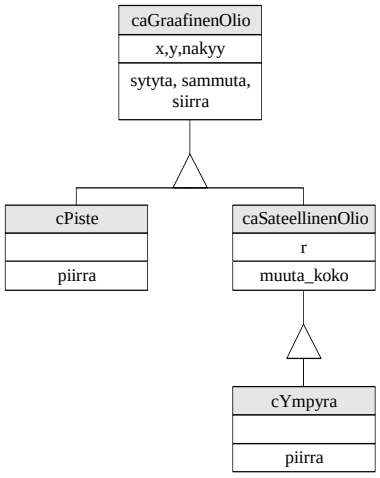
- ei tarvita (eikä ole) **`break`** -lauseita kunkin vaihtoehdon jälkeen (*C*:n ilman `break`iä olevaa tapausta ei voi edes toteuttaa `case`-lauseella)
- samassa vaihtoehdossa voi olla myös väli **`..`** ilmoitettuna
- useita vaihtoehtoja voidaan erottaa pilkulla
- valitsimena voi olla mikä tahansa ordinaalityyppi (numeroituva tyyppi)

Tehtävä 2.14 `break`

Muuta *C*-esimerkkiä casof.c siten, että klo 8:sta tulostetaan sekä *Töihin* että *Tehdään töitä*. Kokeile tehdä vastaava muutos caseof.pas tiedostoon.

2.2 Olio-ominaisuudet

Seuraavassa esimerkissä toteutetaan luokkahierarkia (käytämme unkarilaista nimeämistapaa, missä `c=class` ja `a=abstract`):



Kuva 2.1 Esimerkkiohjelman oliohierarkia

piste.cpp - esimerkki perinnästä

```
#include <stdio.h>

// Esimerkki ehdollisesta kaantamisesta:
#define RTTI
// RTTI = Run Time Type Information,
// toimii esim. BC++ 4.5 alkaen
#ifdef RTTI
#include <typeinfo.h>
#endif

//-----------------------------------------
class caGraafinenOlio {
protected:
  int x,y;
  int nakyy;
  int paikka(int nx,int ny)
    { x = nx; y = ny; return 0;             }
public:
  caGraafinenOlio(int ix=0, int iy=0)
    { paikka(ix,iy); nakyy = 0;             }
  virtual ~caGraafinenOlio() { }
  virtual int piirra() const = 0;
```

piste.dpr - esimerkki perinnästä

```
program piste;

{ Esimerkki ehdollisesta kaantamisesta: }
{$DEFINE RTTI }

uses
  SysUtils;


{-----------------------------------------}
type caGraafinenOlio = class
protected
  x,y:integer;
  nakyy:boolean;
  procedure paikka(nx,ny:integer);
public
  constructor Create;
  constructor Create2(ix,iy:integer);
  destructor Destroy; override;
  procedure piirra; virtual; abstract;
  function nakyvissa:boolean;
  procedure sammuta;
  procedure sytyta;
  procedure siirra(nx,ny:integer);
  procedure tulosta(s:string); virtual;
end;

procedure caGraafinenOlio.paikka(
                                  nx,ny:integer);
begin x := nx; y := ny; end;

constructor caGraafinenOlio.Create2(
                                  ix,iy:integer);
begin
  inherited Create;
  paikka(ix,iy);
  nakyy := False;
end;
```

```
  int nakyvissa() const { return nakyy;     }
  int sammuta();
  int sytyta();
  int siirra(int nx, int ny) {
    if ( !nakyvissa() ) return paikka(nx,ny);
    sammuta(); paikka(nx,ny); return sytyta();
  }




  virtual int tulosta(const char *s="")
   const {
#   ifdef RTTI
    printf("%-10s: ",typeid(*this).name());
#   endif
    printf("%-10s (%02d,%02d)",s,x,y);
    return 0;
  }
}; // caGraafinen olio

int caGraafinenOlio::sammuta()
{
  if ( !nakyvissa() ) return 1;
  printf("Sammutettu: ");
  nakyy = 0;
  return piirra();
}

int caGraafinenOlio::sytyta()
{
  if ( nakyvissa() ) return 1;
  printf("Sytytetty:  ");
  nakyy = 1;
  return piirra();
}

//-----------------------------------------
class caSateellinenOlio :
  public caGraafinenOlio {
protected:
  int r;
  int koko(int nr) { r = nr; return 0;      }
public:
  caSateellinenOlio(int ix=0,int iy=0,
                    int ir=1) :
    caGraafinenOlio(ix,iy), r(ir) {}
```

```
  virtual int tulosta(const char *s="")
```

```
constructor caGraafinenOlio.Create;
begin Create2(0,0); end;

destructor caGraafinenOlio.Destroy;
begin  sammuta;
  inherited Destroy;
end;

function caGraafinenOlio.nakyvissa:boolean;
begin Result := nakyy; end;

procedure caGraafinenOlio.siirra(
                              nx,ny:integer);
begin
  if ( not nakyvissa ) then begin
    paikka(nx,ny); exit;
  end;
  sammuta; paikka(nx,ny); sytyta;
end;

procedure caGraafinenOlio.tulosta(s:string);
begin
  {$IFDEF RTTI}
  write(format('%-10s: ',[ClassName]));
  {$ENDIF}
  write(format('%-10s (%02d,%02d)',[s,x,y]));
end;

procedure caGraafinenOlio.sammuta;
begin
  if ( not nakyvissa ) then exit;
  write('Sammutettu: ');
  nakyy := False;
  piirra;
end;

procedure caGraafinenOlio.sytyta;
begin
  if ( nakyvissa ) then exit;
  write('Sytytetty:  ');
  nakyy := True;
  piirra;
end;

{-----------------------------------------}
type caSateellinenOlio =
  class(caGraafinenOlio)
protected
  r:integer;
  procedure koko(nr:integer);
public
  constructor Create3(ix,iy,ir:integer);
  constructor Create2(ix,iy:integer);
  constructor Create;
  procedure tulosta(s:string); override;
  procedure muuta_koko(nr:integer);
end;

procedure caSateellinenOlio.koko(nr:integer);
begin r := nr; end;

constructor caSateellinenOlio.Create3(
                             ix,iy,ir:integer);
begin
  inherited Create2(ix,iy);
  koko(ir);
end;

constructor caSateellinenOlio.Create2(
                                ix,iy:integer);
begin Create3(ix,iy,1); end;

constructor caSateellinenOlio.Create;
begin inherited Create; koko(1); end;

procedure caSateellinenOlio.tulosta(s:string);
begin
```

```
  const {
    caGraafinenOlio::tulosta(s);
    printf( " r=%d",r);
    return 0;
  }
  int muuta_koko(int nr) {
    if ( !nakyvissa() ) return koko(nr);
    sammuta(); koko(nr); return sytyta();
  }
}; // caSateellinen olio



//-----------------------------------------
class cPiste : public caGraafinenOlio {
public:
  cPiste(int ix=0, int iy=0) :
    caGraafinenOlio(ix,iy) {}
  virtual ~cPiste() { sammuta(); }
  virtual int piirra() const {
   tulosta("Piste"); printf("\n"); return 0;}
};

//-----------------------------------------
class cYmpyra : public caSateellinenOlio {
public:
  cYmpyra(int ix=0, int iy=0, int ir=1) :
    caSateellinenOlio(ix,iy,ir) {}
  virtual ~cYmpyra() { sammuta(); }
  virtual int piirra() const {
   tulosta("Ympyra"); printf("\n"); return 0;}
};


int main(void)
{
  caGraafinenOlio *p;
  caGraafinenOlio *kuvat[10];
  int i;
  cPiste p1,p2(10,20);
  cYmpyra y1(1,1,2);
  p1.sytyta(); p2.sytyta();
  p1.siirra(7,8);
  y1.sytyta(); y1.muuta_koko(5);

  // Esimerkki polymorfismista
  p = new cYmpyra(9,9,9);
  p->sytyta(); p->siirra(8,8);
  // p->muuta_koko(4); ei laillinen
# ifdef RTTI
//  if ( typeid(*p) == typeid(cYmpyra) )
  caSateellinenOlio *ps =
    dynamic_cast<caSateellinenOlio *>(p);
  if ( ps ) ps->muuta_koko(4);
# else
  ((caSateellinenOlio *)p)->muuta_koko(4);
# endif
  delete p;

  // Esimerkki polymorfismista
  kuvat[0] = new cYmpyra(10,10,100);
  kuvat[1] = new cPiste(11,11);
  kuvat[2] = new cYmpyra(12,12,102);
  kuvat[3] = NULL;
  for (i=0; kuvat[i];i++) kuvat[i]->sytyta();
  for (i=0; kuvat[i];i++) delete kuvat[i];

  return 0;
}
```

```
  inherited tulosta(s);
  write(' r=',r);
end;

procedure caSateellinenOlio.muuta_koko(
                                    nr:integer);
begin
  if ( not nakyvissa ) then begin
    koko(nr); exit;
  end;
  sammuta; koko(nr); sytyta;
end;

{-----------------------------------------}
type cPiste = class(caGraafinenOlio)
public
  procedure piirra; override;
end;

procedure cPiste.piirra;
begin tulosta('Piste'); writeln; end;

{-----------------------------------------}
type cYmpyra = class(caSateellinenOlio)
public
  procedure piirra; override;
end;

procedure cYmpyra.piirra;
begin tulosta('Ympyra'); writeln; end;

{-----------------------------------------}
var p     : caGraafinenOlio;
    kuvat : array[0..9] of caGraafinenOlio;
    p1,p2 : cPiste;
    y1    : cYmpyra;
    i     : integer;
begin
  p1 := cPiste.Create;
  p2 := cPiste.Create2(10,20);
  y1 := cYmpyra.Create3(1,1,2);
  p1.sytyta;  p2.sytyta;
  p1.siirra(7,8);
  y1.sytyta; y1.muuta_koko(5);

  (* Esimerkki polymorfismista             *)
  p := cYmpyra.Create3(9,9,9);
  p.sytyta; p.siirra(8,8);
  { p.muuta_koko(4);  ei laillinen }
  {$IFDEF RTTI }
  if ( p is caSateellinenOlio ) then
  {$ENDIF}
  (p as caSateellinenOlio).muuta_koko(4);
  p.Free;

  (* Esimerkki polymorfismista             *)
  kuvat[0] := cYmpyra.Create3(10,10,100);
  kuvat[1] := cPiste.Create2(11,11);
  kuvat[2] := cYmpyra.Create3(12,12,102);
  kuvat[3] := NIL;
  i:=0;
  while ( kuvat[i] <> NIL ) do begin
    kuvat[i].sytyta; inc(i);
   end;
  i:=0;
  while ( kuvat[i] <> NIL ) do begin
    kuvat[i].Free; inc(i);
   end;

  y1.Free;  p2.Free;  p1.Free;
end.
```

Mikäli vakio `RTTI` on määritelty, tulostavat molemmat ohjelmat:

```
Sytytetty:  cPiste    : Piste      ( 0, 0)
Sytytetty:  cPiste    : Piste      (10,20)
Sammutettu: cPiste    : Piste      ( 0, 0)
Sytytetty:  cPiste    : Piste      ( 7, 8)
Sytytetty:  cYmpyra   : Ympyra     ( 1, 1) r=2
Sammutettu: cYmpyra   : Ympyra     ( 1, 1) r=2
Sytytetty:  cYmpyra   : Ympyra     ( 1, 1) r=5
Sytytetty:  cYmpyra   : Ympyra     ( 9, 9) r=9
Sammutettu: cYmpyra   : Ympyra     ( 9, 9) r=9
Sytytetty:  cYmpyra   : Ympyra     ( 8, 8) r=9
Sammutettu: cYmpyra   : Ympyra     ( 8, 8) r=9
Sytytetty:  cYmpyra   : Ympyra     ( 8, 8) r=4
Sammutettu: cYmpyra   : Ympyra     ( 8, 8) r=4
Sytytetty:  cYmpyra   : Ympyra     (10,10) r=100
Sytytetty:  cPiste    : Piste      (11,11)
Sytytetty:  cYmpyra   : Ympyra     (12,12) r=102
Sammutettu: cYmpyra   : Ympyra     (10,10) r=100
Sammutettu: cPiste    : Piste      (11,11)
Sammutettu: cYmpyra   : Ympyra     (12,12) r=102
Sammutettu: cYmpyra   : Ympyra     ( 1, 1) r=5
Sammutettu: cPiste    : Piste      (10,20)
Sammutettu: cPiste    : Piste      ( 7, 8)
```

Jos vakiota `RTTI` ei ole määritelty, on tulostus muuten sama, mutta puuttuu sarake, jossa on `cPiste` ja `cYmpyra`.

*Delphi*ssä on kaksi tapaa tehdä luokkia:

- **`object`**, jolla voidaan tehdä myös staattisia olioita
- **`class`**, jolla voidaan tehdä vain dynaamisia olioita

Tässä monisteessa käytetään pääasiassa vain **`class`**-määreellä esiteltyjä luokkia

- `class`-esitellyt luokat periytyvät aina **`TObject`**-luokasta
- kaikki oliot ovat (`class`-esittelyn jälkeen) dynaamisia (käytännössä osoittimia), eli ne pitää itse luoda ennen käyttöä (=kutsua konstruktoria) ja poistaa käytön jälkeen (kutsua destruktoria)
- **`this`**-osoitinta *Delphi*ssä vastaa **`self`**
- ei *inline*-kirjoitusmahdollisuutta metodeille
- ei funktioiden oletusparametreja
- ei funktioiden lisämäärittelyä (*function overloading*), eli ei voi olla kahta samannimistä funktiota tai kahta samannimistä metodia samassa oliossa. Esimerkissä on tämän takia kirjoitettu useita konstruktoreita: `Create` - 0 parametria, `Create2` - 2 parametria, `Create3` - 3 parametria. Eri olioissa voi tietenkin olla samojakin metodien nimiä (ja usein pitääkin olla).
- ei operaattoreiden lisämäärittelyä (*operator overloading*)
- ei moniperintää (kaikki eivät pidä tätä miinuksena)
- olion sijoittaminen toiseen "olioon" tekee aliasing-ongelman: molemmat "oliot" viittaavat samaan olion esiintymään
- konstruktori periytyy, eli jos isäluokan konstruktori kelpaa sellaisenaan, ei tarvitse kirjoittaa kontruktoria joka pelkästään kutsuu isäluokan kontruktoria
- kannattaa ehkä aina kirjoittaa yksi **`Create`** -niminen parametriton konstruktori, koska tällainen on kantaluokassa ja käyttäjä voi vanhasta tottumuksesta luoda olion esiintymän käyttäen tätä konstruktoria
- myös destruktori näyttäisi periytyvän, eli sitä ei tarvitse kirjoittaa joka luokkaan uudelleen. Destruktorin aikana olio on omaa luokkaansa, eli "luokka ei pienene"

kuten *C++*:ssa. Näin jos destruktorissa kutsutaan jotakin virtuaalista tai välillisesti virtuaalista metodia, käytetään purettavan luokan virtuaalitaulua vaikka kutsu olisikin jonkin kantaluokan destruktorissa

- vaikka destruktoreitakin voi kirjoittaa useita eri nimisiä, kannattaa ehkä kirjoittaa aina vain **`destroy`** -niminen destruktori, koska valmiit oliot tuhotaan **`free`** -metodilla, joka taas kutsuu **`destroy`**:ta (`TObject`-luokassa)
- dynaamisuuden takia joudutaan aina kutsumaan itse eksplisiittisesti olion esiintymän luomiseksi jotakin konstruktioria ja lopuksi olion hävittämiseksi jotakin destruktoria, käytännössä varminta ehkä kutsua *Borlandin* ohjeiden mukaisesti `Free` -metodia.
- *C*:ssä kannattaa joskus tehdä ylimääräisten lauselohkojen välttämiseksi **`int`**-funktioita, vaikkei paluuarvoa tarvitakaan:

```
// Vertaa:
int siirra(int nx, int ny) {
  if ( !nakyvissa() ) return paikka(nx,ny);
  sammuta(); paikka(nx,ny); return sytyta();
}

// Tai:
void siirra(int nx, int ny) {
  if ( !nakyvissa() ) { paikka(nx,ny); return; }
  sammuta(); paikka(nx,ny); sytyta();
}
```

- Pascalissa vastaava ei hyödytä mitään, tosin ei juuri lisää koodiakaan:

```
{ Vertaa: }
function caGraafinenOlio.siirra(nx,ny:integer):integer;
begin
  if ( not nakyvissa ) then begin siirra := paikka(nx,ny); exit;  end;
  sammuta; paikka(nx,ny); siirra := sytyta;
end;

{ Tai: }
procedure caGraafinenOlio.siirra(nx,ny:integer);
begin
  if ( not nakyvissa ) then begin paikka(nx,ny); exit;  end;
  sammuta; paikka(nx,ny); sytyta;
end;
```

- RTTI (=*Run Time Type Information*) on ehkä helpompi käyttää *Delphi*ssä kuin *C++:*ssä

Tehtävä 2.15 C++ ilman inline-funktoita

Kirjoita piste.cpp käyttämättä `inline`-funktioita

Tehtävä 2.16 Neliö ja suorakaide

Lisää kumpaankin esimerkkiin (piste.cpp ja piste.dpr) luokka `cNelio`.
Entä `cSuorakaide`?

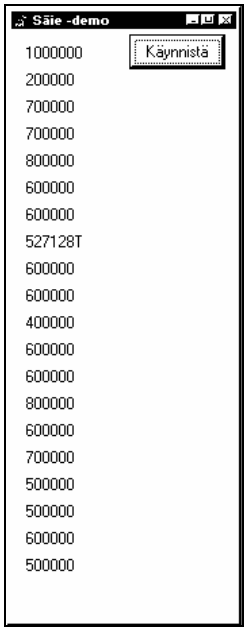
Tehtävä 2.17 Väri ja suunta

Mieti miten luokkahierarkiaa muutetaan, mikäli kuvioista halutaan värillisiä ja eri asennossa olevia.

2.3 Säikeet ja dynaamiset kontrollit

Säikeet (=*threads*) eivät oikeastaan kuulu kieleen, vaan käyttöjärjestelmään. Koska *Delphi*toimii kuitenkin vain *Windowsin* alla, käsitellään tässä säikeet ikään kuin kieleen kuuluvana asiana. Säikeet toimivat *Win95*:sta alkaen. Vertailukohdaksi otetaan *Borland C++ 5.0*, *OWL 5.0* ja *AppExpert*illä osittain tehty sovellus.

Seuraava esimerkki luo dialogi-ikkunassa (lomakkeessa, *form*) valmiiksi olevan Käynnistä-näppäimen lisäksi joukon laskureita (*Delphi*ssä tekstikenttiä ja *C++*:ssa `Tbutton-`nappuloita). Kun Käynnistä-nappia painetaan, käynnistetään kutakin laskurikenttää varten oma prosessi (säie), joka pyörittää kentässä lukuja 0:sta ylöspäin. Kenttää painamalla voidaan ko. säie "tappaa". Kummastakin ohjelmasta on jätetty listaamatta sekä pääohjelma että resurssitiedosto.



saie\saiedemo.cpp - esimerkki säikeistä

```
#include <owl/button.h>
#include <classlib/thread.h>

#include "saieapp.rh" // Def of all resources




const int SAIKEITA = 20;    // Säikeiden lkm
const int PAIVITYS = 100000;// Minkä väl.päiv.
const int KIERROKSIA = 1000000;



class cLaskuri : public TThread
// Laskurisäie perit.yleisestä säikeestä
// ja siihen lisätään omat erikoispiirt.
{
  TControl *Text;  //Mihin teksti-ikkunaan
  int n;           //Sis. laskurin arvo
  int raja;        //Mihin asti lasketaan
protected:
  void count();    //Yhden laskuaskeleen suor
  //Perit.luokan Run korvataan omalla
  int Run();
public:
  //Rakentaja, joka alustaa mm. sis. muut.
  cLaskuri(TControl *oLabel,int r) :
    Text(oLabel), raja(r), n(0) {}
  ~cLaskuri() { ;}
};

//{{TDialog = TFormSaieDemo}}
class TFormSaieDemo : public TDialog {
// Lomake,jossa laskureita pyöritetään
    TControl *Labels[SAIKEITA];
    cLaskuri *saikeet[SAIKEITA];
  public:
    TFormSaieDemo(TWindow* parent,
                TResId resId = IDD_SAIEDEMO,
```

saie\saiedemo.pas - esimerkki säikeistä

```
unit saiedemo;

interface

uses
  Windows, Messages, SysUtils, Classes,
  Graphics, Controls, Forms, Dialogs,
  StdCtrls;


const SAIKEITA = 20;    // Säikeiden lkm
      PAIVITYS = 100000;// Minkä välein päiv.
      KIERROKSIA = 1000000;

type
  //----------------------------------------
  cLaskuri = class(TThread)
  // Laskurisäie perit.yleisestä säikeestä
  // ja siihen lisätään omat erikoispiirt.
  private
    Text : TLabel;   //Mihin teksti-ikkunaan
    n:integer;       //Sis. laskurin arvo
    raja:integer;    //Mihin asti lasketaan
  protected
    procedure count; //Yhden laskuaskeleen su
    //Perit.luokan Execute korvataan omalla
    procedure Execute; override;
  public
    //Rakentaja, joka alustaa mm. sis. muut.
    constructor Create(oLabel:TLabel;
                       r:integer);
  end;


  //----------------------------------------
  TFormSaieDemo = class(TForm)
  // Lomake,jossa laskureita pyöritetään
    ButtonKaynnista: TButton;
    procedure ButtonKaynnistaClick(
      Sender: TObject);
    procedure ThreadDone(Sender:TObject);
    procedure FormCreate(Sender: TObject);
```

```
              TModule* module = 0);
    virtual ~TFormSaieDemo();

//{{TFormSaieDemoVIRTUAL_BEGIN}}
  public:
    virtual bool Create();
    virtual TResult EvCommand(uint id,
            THandle hWndCtl, uint notifyCode);
    virtual bool CanClose();
//{{TFormSaieDemoVIRTUAL_END}}

//{{TFormSaieDemoRSP_TBL_BEGIN}}
  protected:
    void BNKaynnistaClicked();
//{{TFormSaieDemoRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(TFormSaieDemo);
};    //{{TFormSaieDemo}}
```

saie\saiedemo.cpp - esimerkki säikeistä

```
#include <owl/pch.h>

#include "saiedemo.h"

#define ALKU_ID 3000

//----------------------------------------
// cLaskuri ==============================
//----------------------------------------

//----------------------------------------
void cLaskuri::count()
{
  n++;
  if ( n % PAIVITYS == 0 ) {
    char s[30];
    wsprintf(s,"%d",n);
    Text->SetWindowText(s);
  }
}

//----------------------------------------
int cLaskuri::Run()
// Tämä suorittaa varsinaisen säikeen
// toimenpiteet.  Kun tämä metodi loppuu,
// pysähtyy säie.
{
  const char *mes="";
  while ( n < raja ) {
    if ( ShouldTerminate() ) {
      mes = "T";  break;
    }
    count();
  }
  char s[30];
  wsprintf(s,"%d%s",n,mes);
  Text->SetWindowText(s);
  return 0;
}

//----------------------------------------
// TFormSaieDemo ============================
//----------------------------------------

//----------------------------------------
DEFINE_RESPONSE_TABLE1(TFormSaieDemo,TDialog)
//{{TFormSaieDemoRSP_TBL_BEGIN}}
  EV_BN_CLICKED(IDKAYNNISTA,
                BNKaynnistaClicked),
//{{TFormSaieDemoRSP_TBL_END}}
END_RESPONSE_TABLE;

//{{TFormSaieDemo Implementation}}

//----------------------------------------
TFormSaieDemo::TFormSaieDemo(TWindow* parent,
```

```
    procedure LabelsClick(Sender: TObject);
  private
    { Private declarations }
    Labels: Array [0..SAIKEITA-1] of TLabel;
    saikeet:Array[0..SAIKEITA-1] of cLaskuri;
 public
    { Public declarations }
  end;
```

```
//----------------------------------------
// Globaalit muuttujat
//----------------------------------------
var
  FormSaieDemo: TFormSaieDemo;

implementation

//----------------------------------------
// cLaskuri ==============================
//----------------------------------------

//----------------------------------------
procedure cLaskuri.count;
begin
  inc(n);
  if ( n mod PAIVITYS = 0 ) then
    Text.Caption := IntToStr(n);
end;
```

```
//----------------------------------------
procedure cLaskuri.Execute;
// Tämä suorittaa varsinaisen säikeen
// toimenpiteet.  Kun tämä metodi loppuu,
// pysähtyy säie.  Säie häviää autom.
// jos FreeOnTerminate := True
var mes:String;
begin
  mes := '';
  while ( n < raja ) do begin
    if terminated then begin
      mes := 'T'; Break;
    end;
    count;
  end;
  Text.Caption := IntToStr(n)+mes;
end;

//----------------------------------------
constructor cLaskuri.Create(oLabel:TLabel;
                            r:integer);
begin
  inherited Create(False);
  Text := oLabel;
  raja := r;
  n := 0;
  FreeOnTerminate := True;
end;

{$R *.DFM}

//----------------------------------------
// TformSaieDemo ============================
//----------------------------------------
```

```
 TResId resId, TModule* module)
  :  Tdialog(parent, resId, module)
{
}


//----------------------------------------
TFormSaieDemo::~TFormSaieDemo()
{
  Destroy();
}

//----------------------------------------
static bool Clear(cLaskuri * &saie)
// Tämä funktio tarkistaa onko säie jo
// siivottu, eli se on tuhottu.
// Mikäli säie ei ole tuhottu, mutta se
// on valmis, tuhotaan säie.
{
  if ( saie == NULL ) return true;
  if ( saie->GetStatus() !=
         TThread::Finished ) return false;
  delete saie;
  saie = NULL;
  return true;
}

//----------------------------------------
void TFormSaieDemo::BNKaynnistaClicked()
{
  for (int i=0; i<SAIKEITA; i++) {
    if ( !Clear(saikeet[i]) ) continue;
    saikeet[i] = new cLaskuri(Labels[i],
                              KIERROKSIA);
    saikeet[i]->Start();
  }
  saikeet[0]->SetPriority(
    THREAD_PRIORITY_ABOVE_NORMAL);
}




//----------------------------------------
TResult TFormSaieDemo::EvCommand(uint id,
           THandle hWndCtl, uint notifyCode)
// Jos laskuria klik., "tapetaan vast.säie"
{
  TResult result=TDialog::EvCommand(id,
                        hWndCtl,notifyCode);
  int i = id-ALKU_ID;
  if ( 0 <= i && i < SAIKEITA )
    if ( !Clear(saikeet[i]) )
      saikeet[i]->Terminate();
  return result;
}

//----------------------------------------
bool TFormSaieDemo::Create()
// Luodaan laskurit alekkain näytölle
{
  bool result;

  int y = 10, dy = 20;
  for (int i=0; i<SAIKEITA; i++) {
    saikeet[i] = NULL;
    Labels[i] = new Tbutton(this,ALKU_ID+i,
                        "Terve",10,y,70,dy);
    y += dy;
  }

  result = Tdialog::Create();

  TRect rc = this->GetWindowRect();
  rc.top = 0; rc.bottom = rc.top + y + 50;
  this->MoveWindow(rc,TRUE);
```

```
//----------------------------------------
procedure TFormSaieDemo.ThreadDone(
                                Sender:TObject);
// Tämä tapahtuma, kun jokin säie valmistuu.
var i:integer;
begin
  for i:=0 to SAIKEITA-1 do // Etsitään säie
    if ( Sender = saikeet[i] ) then
      saikeet[i] := NIL;
end;




//----------------------------------------
procedure TFormSaieDemo.ButtonKaynnistaClick(
                               Sender: TObject);
var i:integer;
begin
  // Luodaan uudet säikeet niiden tilalle
  // joita ei ole
  for i:=0 to SAIKEITA-1 do
    if (  saikeet[i] = NIL ) then begin
      saikeet[i] :=
       cLaskuri.Create(Labels[i],KIERROKSIA);
      saikeet[i].OnTerminate := ThreadDone;
    end;
  saikeet[0].Priority  := tpHigher;
end;

//----------------------------------------
procedure TFormSaieDemo.LabelsClick(
                                Sender:TObject);
// Jos laskuria klik., "tapetaan vast.säie"
var i:integer;
begin
  i := (Sender as  TLabel).Tag;
  if ( saikeet[i] <> NIL ) then
    saikeet[i].Terminate;
end;



//----------------------------------------
procedure TFormSaieDemo.FormCreate(
                               Sender: TObject);
// Luodaan laskurit alekkain näytölle
var i,y,dy:integer;
begin
  y := 10; dy := 20;
  for i:=0 to SAIKEITA-1 do begin
    saikeet[i] := NIL;
    Labels[i] := TLabel.Create(Self);
    Labels[i].AutoSize := False;
    Labels[i].top := y;
    Labels[i].left := 10;
    Labels[i].Width := 70;
    Labels[i].Caption := 'Terve';
    Labels[i].Parent := Self;
    Labels[i].Tag := i;
    Labels[i].OnClick := LabelsClick;
    y := y + dy;
  end;
  Top := 0; Height := y + 50;
```

```
  return result;
}

bool TFormSaieDemo::CanClose()
{
  bool result = TDialog::CanClose();

  for (int i=0; i<SAIKEITA; i++) {
    if ( Clear(saikeet[i]) ) continue;
    result = false;
    saikeet[i]->Terminate();
  }
  return result;
}
```

```
end;

end.
```

- *Delphi*ssä säikeen loppumisesta saadaan selvä viesti. Tämä helpottaa pysähtyneiden säikeiden poistamista tai muuta valmistuneen prosessin jatkokäsittelyä
- *Delphi*ssä ohjelman voi kokeilujen perusteella turvallisesti sammuttaa vaikka säikeiden suoritus olisikin kesken. *C++*:ssa täytyy sammuttaminen estää, mikäli säikeitä on käynnissä ja suorittaa sammutus loppuun vasta kun säikeet ovat loppuneet
- *Delphi*ssä on helpompi laittaa säie käyntiin jo konstruktorissa, koska tätä explisiittisesti kuitenkin kutsutaan: *C++:*ssa voi olla myös staattinen olio ja tällöin säie lähtisi käyntiin jo staattisen olion esittelyn yhteydessä
- *Delphi*ssä dynaamisten kontrollien tekeminen on jossain määrin helpompaa, kun viestinkäsittelijä voidaan sijoittaa kontrolliin jo luontivaiheessa
- vaikka dynaamiset kontrollit onkin itse luotu, häviävät ne automaattisesti lomakkeen poistuessa, koska kontrollit ovat lomakkeen lapsia
- luotaessa kontrolleja dynaamisesti, kannattaa kontrollin `Tag`-ominaisuuteen laittaa jokin kontrollia hyvin kuvaava arvo. Edellisessä esimerkissä tämä on ollut kontrollin järjestysnumero
- *Delphi*ssä on huomattavasti helpompi muuttaa kontrollin tiettyä ominaisuutta. Toisaalta jos muutetaan kerralla esim. paikka ja kokoa, on *C++:*n suorakaiteen välittäminen järkevämpää, koska jokaisesta yksittäisestä sijoituksesta tulee periaatteessa muutostarve näytöllä. Tosin *Delphi*ssä voidaan käyttää `SetBounds` -metodia.
- *Delphi*n merkkijonoja on huomattavasti helpompi käyttää kuin *C*:n perusmerkkijonoja, joiden päälle *Windows* ja myös *OWL*-luokkakirjasto on suurelti rakennettu

Tehtävä 2.18 TLabel => TButton

Muuta saiedemo.pas-esimerkissä laskurikentät nappuloiksi.

2.3.1 Delphi ei ole ”thread safe”

Ainakaan *Delphi 3.0* ei ole vielä ”*thread safe*”. Tällä tarkoitetaan mm. Sitä, että vain ohjelman pääsäie saa käyttää tiettyjä VCL-kirjaston kutsuja. Itse asiassa edellisen esimerkkiohjelman ei tämän mukaan kuuluisi toimia lainkaan!

Jokaisen säikeen tulisi synkronoitua pääsäikeeseen jos säikeen tarvitsee mm. päivittää näyttöä. Näin säikeestä tulee hetkellisesti osa pääsäiettä, jolla on VCL:än käyttöoikeus. Edellisessä esimerkissä tämä tehtäisiin esimerkiksi siten, että

```
Text.Caption := IntToStr(n)+mes;
```

siirrettäisiin esimerkiksi omaksi metodikseen nimeltä `UpdateDisplay` ja metodia kutsuttaisiin:

```
Synchronize(UpdateDisplay);
```

Synkronointikutsuun kelpaa vain parametriton säikeen metodi. Mahdollinen parametrin välitys (edellä `mes`) pitää valitettavasti hoitaa olion attribuuttien avulla.

Esimerkkiohjelma tosin toimii synkronoituna huomattavasti huonommin, mutta esimerkiksi `Canvas`ta käsittelevät säikeet on pakko synkronoida.

Tehtävä 2.19 `Synchronize`

Muuta saiedemo.pas-esimerkissä näytön päivitys synkronoiduksi. Kokeile muuttaa säikeiden prioriteettia.

Tehtävä 2.20 Erillinen näytön päivitys

Toinen tapa hoitaa näytön päivitys ”siististi” olisi esimeriksi seuraava: Lisää `cLaskuri`-luokkaan:

attribuutti:	`DispText`	- näytössä oleva teksti
metodi:	`IsUpdated`	- palauttaa tiedon onko laskuri päivitetty
metodi:	`UpdateDisplay`	- päivittää näytön

Pääohjelmassa voi sitten esimerkiksi ”kellokeskeytysten” avulla kysellä mitkä laskurit tarvitsevat päivitystä ja sitten päivittää ne. Tai `UpdateDisplay` voi suoraan olla sellainen, ettei se turhaan päivitä näyttöä. Tällä menetelmällä säikeet saavat rauhassa keskittyä laskemiseen ja pääohjelma hoitelee näyttöä. Vaikka metodit ovatkin säikeen sisällä, ne suoritetaan pääohjelman säikeessä, mikäli pääohjelma niitä kutsuu.

3. Tietokantojen käyttö

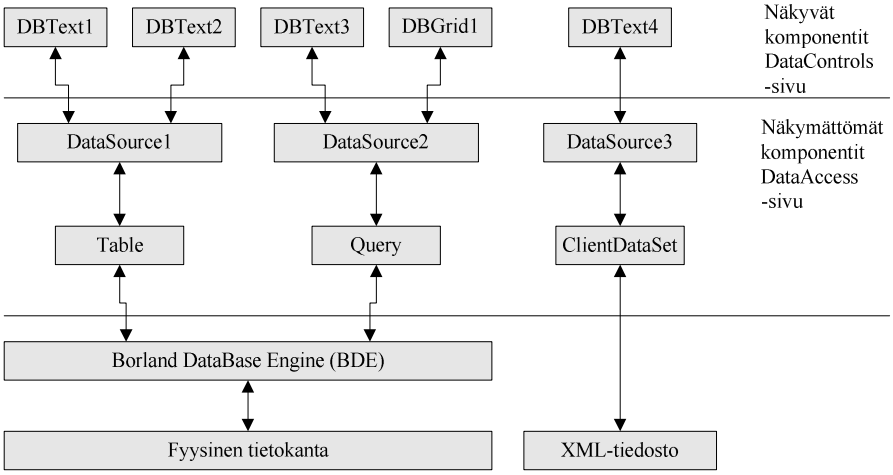
Luvun pääaiheet:

- **tietokantataulujen käyttö Delphissä**
- **paneelit auttamassa komponenttien sijoittelua**
- **SQL-kyselyt**
- **dynaamisesti luotavat tietueen kentät**
- **luvun esimerkit hakemistossa puh**

Eräs *Delphi*n vahvimmista puolista on mahdollisuus vähällä ohjelmoimisella käsitellä valmiita, jopa muussa koneessa olevia tietokantoja.

3.1 Tietokantakomponentit

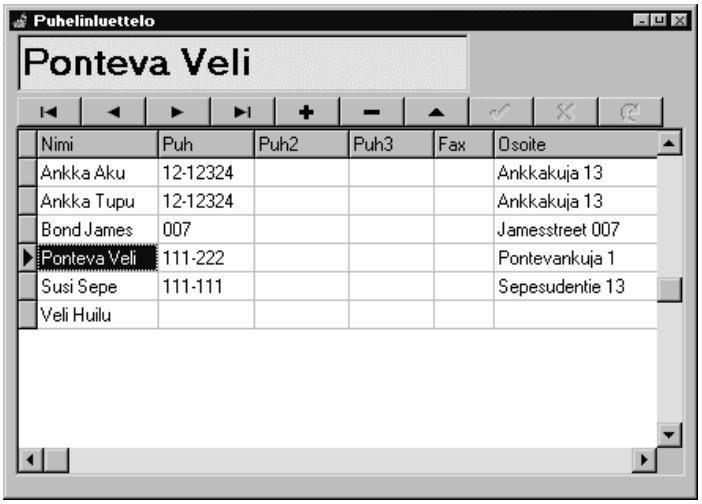
Seuraavassa kuvassa on esitelty *Delphi*in tietokantakomponenttien toimintaperiaate



Kuva 3.1 Tietokantakomponenttien toimintaperiaate

3.2 Yhden taulun esimerkki

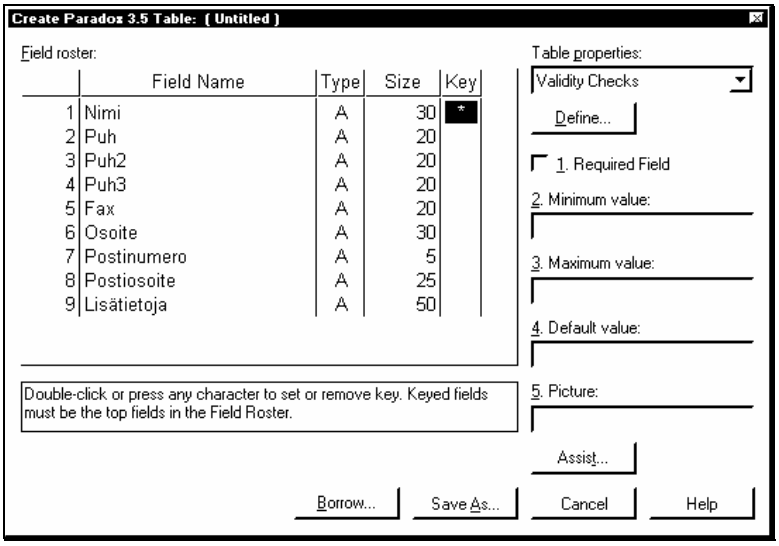
Aloitamme aivan yksinkertaisella puhelinluettelo-esimerkillä. Seuraava ohjelma syntyy kirjoittamatta riviäkään ohjelmakoodia:



Kuva 3.2 "Valmis" puhelinluettelo ilman koodaamista

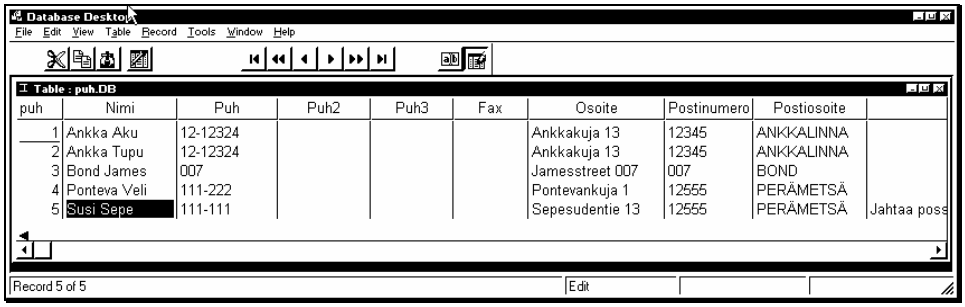
3.2.1 Tietokantataulun luominen

1. Käynnistä aluksi *Delphi*n mukana tullut *Database Desktop* (DBD).
2. Tarkista että työhakemisto on oikea - vaihda tarvittaessa
3. Luo uusi taulu. Jos mitään ihmeominaisuuksia ei tulla tarvitsemaan, kannattaa ehkä taulun tyypiksi valita *Paradox 3.5*. Määrittele esim. seuraavat kentät:



Kuva 3.3 Tietokantataulun kenttien määrittely

4. Talleta taulu vaikkapa nimelle puh
5. Avaa edellä luotu taulu puh ja täytä aluksi muutama henkilö:
6. Sulje *DBD*.



Kuva 3.4 Valmis tietokantataulu

3.2.2 Delphi-sovellus, joka käyttää valmista taulua

Nyt kun meillä on valmis tietokantataulu, voimme tehdä *Delphi*-sovelluksen, jolla taulua käsitellään. Tämä voitaisiin tehdä vielä helpommin käyttämällä valmista tietokanta-experttiä, mutta asian ymmärtämiseksi teemme vähän enemmän käsityötä:

1. Luo uusi tyhjä *Delphi*-sovellus
2. Muuta lomakkeen nimi ja otsikko sopivasti
3. Aluksi tarvitaan yhteys itse tietokantaan. Tämä voidaan tehdä joko taulukkokomponentilla tai *SQL*-komponentilla `Query` (*Structured Query Language*), jotka löytyvät Data Access -sivulta. Valitsemme yleiskäyttöisyyden vuoksi *SQL*-komponentin. Sijoita komponentti mihin tahansa lomakkeella (näkymätön komponentti).
4. Laita ensin nimi ja SQL-ominaisuus ja sen jälkeen muut ominaisuudet:

```
Name = QueryPuh
SQL.Strings = select * from puh
Active = True
RequestLive = True
```

5. Yhteys tauluun on valmis. Seuraavaksi tarvitaan komponentti, joka käsittelee yhtä tietuetta (=taulun yksi rivi). Lisää *DataSource*-komponentti johonkin ja muuta ominaisuudet:

```
Name = DataSourcePuh
DataSet = QueryPuh
```

6. Lopuksi tarvitaan vielä jokin komponentti, joka näyttää tietueen käyttäjälle. Näiksi voidaan `Data Controls -sivulta` valita joko yksittäisiä kenttiä tai jopa koko taulun näyttävä komponentti `DBGrid`. Tämä on näkyvä komponentti, joten sijoita se siten kuin haluat taulun näkyvän näytöllä. Muuta ominaisuudet:

```
name = DBGridPuh
DataSource = DataSourcePuh
```

7. Nyt voit jopa kokeilla ohjelman toimintaa!
8. Lisätään vielä komponentti, jolla on helppo liikkua taulukossa edestakaisin: `DBNavigator:`

```
name = DBNavigatorPuh
DataSource = DataSourcePuh
```

9. Lopuksi komponentti, joka näyttää isolla kohdalla olevan henkilön nimen: `DBEdit`

```
name = DBEditNimi
Color = clYellow
DataSource = DataSourcePuh
DataField = 'Nimi'
Font.Height = 18
Font.Name = MS Sans Serif
Font.Style = [fsBold]
```

Tehtävä 3.21 Database form

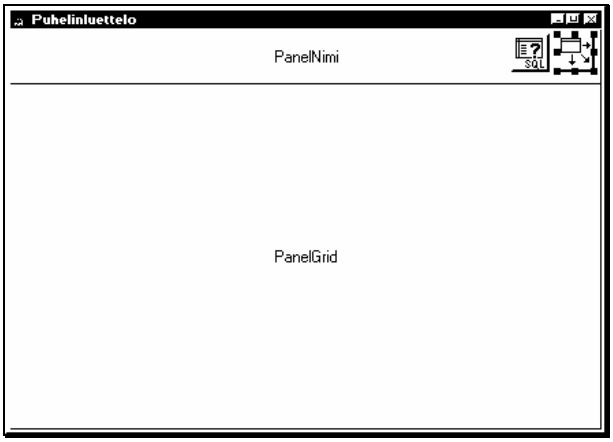
Luo edellisen esimerkin sovellus seuraavasti: File/New/Forms/Database Form (Delphi 2.0) tai Options/Gallery/Database form (Delphi 1.0).
Kokeile vielä saman "expertin" avulla lisätä lomake, jossa onkin "lomakemuotoinen näkyvä" tietueeseen.

3.3 Paneelit

Edellisessä ohjelmassa on se huono puoli, että mikäli ikkunan kokoa suurennetaan, ei taulukosta näy yhtään enempää. Useilla komponenteilla on kuitenkin `Align`-ominaisuus, jolla voidaan säätää miten komponentti muuttaa kokoaan sen isä-ikkunan muuttaessa kokoaan. Eli lisäämällä sopiva määrä "ylimääräisiä" isä-ikkunoita, saadaan sovelluksen komponentit toimimaan halutulla tavalla. Valitettavasti *Delphi*ssä ei voi suoraan siirtää komponenttia toisen ikkunan omistukseen (?), mutta tämä voidaan onneksi kiertää leikekirjan kautta.

3.3.1 Ikkunan jakaminen kahteen osaan

Tavoitteena on nyt aluksi jakaa sovellusikkuna kahteen loogiseen osaan, joista ylemmässä (`PanelNimi`), aina samankorkuisena pysyvässä, on henkilön nimi ja alemassa (`PanelGrid`) ikkunan koon mukaan muuttuva tietokantataulu:



Kuva 3.5 Paneelit

Ominaisuudet paneeleille laitetaan seuraavasti

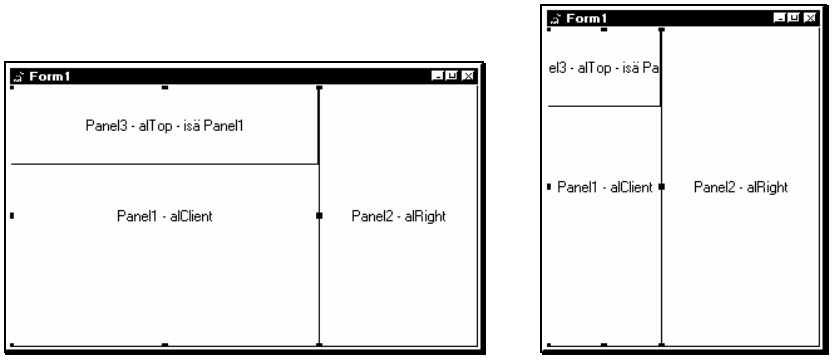
```
name = PanelNimi
Height = 40
Align = alTop
Caption = ''

name = PanelGrid
Align = alClient
Caption = ''
```

Tässä `Align`-ominaisuus määrää miten komponentti - tässä tapauksessa paneeli - muuttaa kokoaan.

1. `alTop` tarkoittaa, että paneeli menee aina ylimpään mahdolliseen paikkaan korkeutensa säilyttäen, leveys muuttuu, täyttäen kaiken tilan (mikä isä-ikkunan sisällä on käytettävissä). Mikäli kaksi `alTop` paneelia on päällekkäin, menee ylempi isä-ikkunansa yläreunaan ja alempi alkaa ylemmän alareunasta. Tällöin kumpikin säilyttää korkeutensa.
2. `alClient` tarkoittaa, että komponentti täyttää kaiken isä-ikkunasta jäljelle jäävän tilan.
3. `alBottom` on `alTop`in vastakohta.
4. `alLeft` ja `alRight` säilyttävät komponentin leveyden, mutta täyttävät korkeussuunnassa kaiken mahdollisen tilan.

3.3.2 Ikkunan jakaminen kolmeen osaan

Mikäli esim. `alTop` ja `alRight` kohtaavat, voittaa `alTop` kiistelyn tilan leveydestä. Tämän takia usein joutuu laittamaan ylimääräisiä paneeleja jakamaan tila ensin pystysuorasti ja sitten näiden sisään paneeleja jakamaan vaakasuorasti. Seuraavassa esimerkissä ikkuna on ensin jaettu pystysuorasti kahtia `Panel1` ja `Panel2`. `Panel2` säilyttää aina leveytensä (`alRight`). Sitten `Panel1` on laitettu täyttämään koko jäljelle jäänyt tila (`alClient`) ja kun `Panel1` on aktiivinen, niin sen sisälle on lisätty `Panel3`, joka aina säilyttää korkeutensa (`alTop`). Isyys määrätään siis sillä, mikä komponentti on aktiivinen kun uusi komponentti lisätään. "Isänä" voi toimi mm. `Form`, `GroupBox`, `RadioGroup`, `TabbedNotebook` ja `Panel`.



Kuva 3.6 Ikkunan jako kolmeen osaan

Tehtävä 3.22 Ikkunan jakaminen 9 osaan

Kokeile mitkä paneelit pitää sijoittaa, jotta saat seuraan jaon ikkunoilla: A, B, C ja D säilyttävä aina sekä korkeutensa ja leveytensä ja E muuttaa sekä korkeutta että leveyttä ikkunan koon muuttuessa:

A		B
	E	
C		D

3.3.3 Paneelien näkyminen

Paneelien väliset rajat voidaan tietysti tarpeen mukaan tehdä näkyviksi tai näkymättömiksi. Kokeile!

3.3.4 Suhteellisen koon säilyttäminen

Mikäli ikkuna halutaan jakaa esim. 3 yhtäsuureen osaan korkeussuunnassa, joudutaan itse ohjelmoimaan miten paneelien koot muuttuvat. Tämä voidaan kirjoittaa esim. `FormResize`-tapahtumaan:

```
{ Panel1.Align = alTop      }
{ Panel2.Align = alTop      }
{ Panel3.Align = alClient   }
procedure TFormPanelDemo.FormResize(Sender: TObject);
var korkeus:integer;
begin
  korkeus := ClientHeight div 3;
  Panel1.Height := korkeus;
  Panel2.Height := korkeus;
end;
```

Tehtävä 3.23 Ikkunan jakaminen 9 osaan ¼ suhteessa

Muuta edellistä yhdeksään osaan jaettua ohjelmaa siten, että ikkunoiden A, B, C ja D leveydet ja korkeudet ovat aina ¼ koko ikkunan leveydestä ja korkeudesta

3.3.5 Paneelien lisääminen jälkikäteen

Olisi tietysti mainiota jos paneelit osattaisiin sijoittaa etukäteen. Usein paneelien tarpeen kuitenkin huomaa vasta jälkeenpäin ja paneeleja pitää laittaa olemassa olevien komponenttien "alle". Näinhän meidän puhelinluettelo-esimerkissäkin:

Lisätään ensin `PanelNimi`

1. Leikkaa leikekirjaan (Cut) `DBEditNimi`
2. Lisää `PanelNimi` ja laita `Align = alTop`
3. Varmista että `PanelNimi` on valittuna.
4. Liimaa (Paste) `DBEditNimi` paneeliin ja siirrä siten oikealle kohdalleen

Seuraavaksi lisätään `PanelGrid`. Tähän tulee kaksi komponenttia:

1. Valitse ensin vaikkapa `DBNavigatorPuh` ja laita Shift pohjaan ja valitse `DBGridPuh`. Varmista että molemmat ja ainoastaan nämä on valittuna (valinta näkyy harmaana).
2. Leikkaa molemmat komponentit leikekirjaan (`Cut).`
3. Klikkaa lomaketta tyhjältä kohdalta, jottei seuraava paneeli mene vahingossa `PanelNimi`-paneelin sisälle
4. Lisää `PanelGrid` ja laita `Align = alClient`
5. Varmista että `PanelNimi` on valittuna.
6. Liimaa (Paste) leikekirjan sisältö paneeliin ja siirrä siten komponentit oikealle kohdalleen.
7. Laita `DBNavigatorPuh`iin `Align = alTop`
8. Laita `DBGridPuh`iin `Align = alClient`
9. Kokeile muuttaa ikkunan kokoa ja tarkista että komponenttien koot muuttuvat halutulla tavalla

3.3.6 Muiden komponenttien koon automaattinen koon muutos

Kaikkien komponenttien `Align`-ominaisuutta ei voi muuttaa suunnitteluaikana (ei ole tietoa miksi ei voi!). Tarpeen vaatiessa `Align`-ominaisuus voidaan kuitenkin muuttaa ajon aikana sijoituksella sopivassa tapahtumassa, esimerkiksi lomakkeen luonnissa:

```
ButtonNollaa.Align := alBottom;
```

Tehtävä 3.24 Nappulat nurkissa

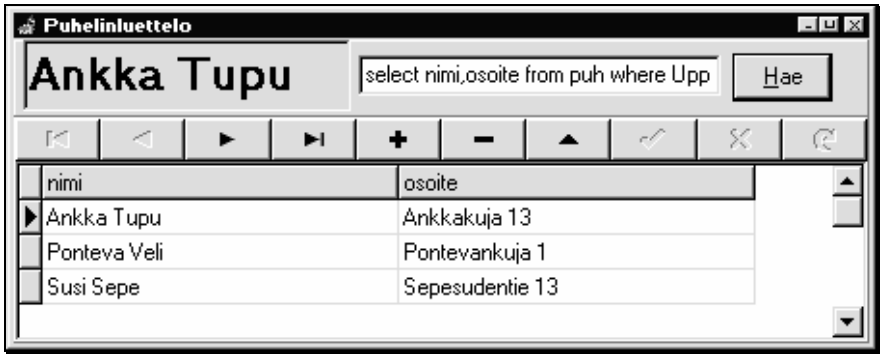
Muuta 9-osaan ¼ -suhteessa jaettua ohjelmaa siten, että kussakin nurkassa on nappula, joka on koko nurkkapaneelin kokoinen.

3.4 SQL-kyselyt

`SQL`-komponenttiin laitettiin hakuehto

```
select * from puh
```

Tämä tarkoittaa, että valitaan kaikki kentät (`*`) taulusta `puh`. Muutetaan ohjelmaa siten, että hakuehto voidaan kirjoittaa itse.



Kuva 3.7 SQL-hakuehto

1. Lisää komponentit `Edit` ja `Button` paneeliin `PanelNimi:`

```
TEdit:
  name = EditHakuehto
  Text = 'select * from puh'
TButton:
  name = ButtonHae
  Caption = '&Hae'
  Default = True
```

2. Laita `Hae`-nappulan tapahtumaksi:

```
procedure TFormPuh.ButtonHaeClick(Sender: TObject);
begin
  QueryPuh.Close;
  QueryPuh.SQL.Clear;
  QueryPuh.SQL.Add(EditHakuehto.Text);
  QueryPuh.Open;
end;
```

3. Kokeile ajaa ohjelmaa esim. SQL-hakuehdoilla:

```
1: select nimi from puh
2: select nimi,postinumero from puh
3: select * from puh where nimi > "Bond"
4: select * from puh where nimi like "%p%"
5: select * from puh where Upper(nimi) like "%P%"
6: select nimi,osoite from puh order by osoite
7: select osoite from puh
```

Huomattakoon ettei järjestetyn haun (`order by`) tulosta voida muokata! *Delphi 1.0*:ssa rajoitukset ovat vielä voimakkaammat, esim. `like`-hakuehdolla haettuakaan taulua ei voi editoida.

Tehtävä 3.25 Haku nimen alkuosan perusteella

Hakuehdolla

```
select * from puh where upper(substring(nimi from 7 for 2)) ="AK"
```

löydettäisiin mm. `Ankka Aku`. Lisää ohjelmaan `Edit`-ikkuna, johon käyttäjä voi kirjoittaa nimeä alusta päin. Kun käyttäjä on kirjoittanut `a`, haetaan kaikki `a`:lla alkavat nimet, kun käyttäjä painaa vielä lisäksi `n`, haetaan kaikki `an` -alkavat nimet jne. Haku tapahtuu heti kun kirjainta on painettu. Onko tehtävän alussa annettu vinkki sopiva tähän tarkoitukseen, vai olisiko joku vielä yksinkertaisempi hakuehto käytettävissä. Jos haun tulos halutaan korjailtavaksi, tulee tämä "ilmaiseksi" *Delphi2.0*:ssa, mutta *1.0*:ssa joutuu koko haun tekemään toisella tavalla.
(idean alku *Delphi 1.0*:aa varten : `"A"<=nimi and nimi <"B"`)

3.5 Tietueen ominaisuuksien selvittäminen

Hakuehto "`select osoite from puh`" päättyy virheeseen. Miksi? Koska `DBEditNimi`-komponentti tarvitsee `nimi`-kenttää ja ao. haun tuloksena nimikenttää ei saada. Voisimme muuttaa ohjelmaa siten, että `DBEditNimi`-komponentissa olisikin aina tietueen 1. kentän tiedot:

1. Lisää `Hae`-nappulan tapahtuman alkuun:

```
DBEditNimi.DataField := '';
```

2. ja tapahtuman loppuun:

```
DBEditNimi.DataField := QueryPuh.Fields[0].FieldName;
```

3. Kokeile nyt esim. hakuehtoa:

```
select puh,osoite from puh
```

Voisimme vielä muuttaa ohjelmaa siten, että "isossa" kenttäikkunassa näkyy aina aktiivisen kentän arvo:

1. Lisää käsittelijä taulukon sarakkeen vaihtumiselle

```
procedure TFormPuh.DBGridPuhColEnter(Sender: TObject);
begin
  DBEditNimi.DataField := DBGridPuh.SelectedField.FieldName;
end;
```

3.6 Dynaamisesti luotavat data-kontrollit

DataForm-Expertillä syntyi mukavasti lomake, jossa on kaikki tietueen kentät näkyvissä. Huono puoli tässä on se, että mikäli tehdään hakuja, joissa rajoitetaan kenttien lukumäärää, ei kaikkia tarvittavia lomakkeeseen olekaan saatavilla.

Muutetaan vielä ohjelmamme sellaiseksi, että siinä on kaksi sivua:

1. sivu, jolla näkyy taulu taulumuotoisena
2. sivu, jolla aktiivinen tietue näkyy lomakemuotoisena, lomake generoidaan haun perusteella dynaamisesti

3.6.1 TabbedNotebook

Lisätään aluksi `DBGridPuh`:in tilalle "lärpsykkäkirja":

1. Aktivoi `DBGridPuh` ja leikkaa se leikekirjaan
2. Aktivoi `PanelGrid` ja laita sille `TabbedNotebook`-komponentti:

```
name  = TabbedNotebookPuh
Align = alClient
Pages = '&Taulu','&Lomake'   (tuplaklikataan Pages ominaisuutta)
```

3. Valitse hiiren oikealla näppäimellä `TabbedNotebookPuh` ja valitse sivu `Taulu`.
4. Liimaa leikekirjassa oleva `TabbedNotebookPuh` tälle sivulle
5. Valitse sivu `Lomake` ja lisää tälle sivulla `ScrollBox`:

```
name = ScrollBoxLomake
Align = alClient
```

6. Kokeile ajaa ohjelmaa.

3.6.2 PageControl

Delphi 6.0:ssa on `TabbedNotebook`kia parempi komponentti: `PageControl`.

Kokeile edellinen esimerkki `PageControl`in avulla.

3.6.3 Apuluokka `cKentat`

Kirjoitetaan ensin apuluokka `cKentat`, joka hoitelee kontrollien lisäämisen:

dynkent.pas - luokka dynaamisten kenttien tekemiseen

```
unit dynkent;
{
  Purpose: Dynaamisesti lisätä taulun kenttiä paneelin sisälle
  Author:  Vesa Lappalainen
  Date:    21.08.1996
  Usage:   Alustus:
             olion varaus: kentat.cKentat
             kentat := cKentat.Create(MyPanel);
           Täyttö:
             kentat.LisaaKaikki(MyDataSource);
           Poisto:
             kentat.Free
}

interface

uses
  DB, Classes, StdCtrls,  DBCtrls, Controls;

const MAX_KENTTIA       = 20;
      C_KENTAN_KORKEUS  = 0;
      C_NIMEN_LEVEYS    = 60;
      C_EDIT_LEVEYS     = 200;
```

```
type
  {-------------------------------------------------------------------------}
  cKentta = object { Pari: otsikko - tietueen kenttä }
    nimi : TLabel;
    edit : TDBEdit;
    index : integer;
  end;

  {-------------------------------------------------------------------------}
  cKentat = class { Luokka, joka tallettaa monta tieueen kentää esim. Paneliin }
  private
    kentat : array [0..MAX_KENTTIA] of cKentta; { Taulukko kentistä     }
    panel : TWinControl;                 { Panelin, johon kentät luod. }
    n : integer;                         { Kenttien lukumäärä          }
    kentan_korkeus,
    nimen_leveys,
    edit_leveys:integer;
    procedure Paikka(var kentta:cKentta); { Vaihtaa kentän paikanja koon}
    procedure MuutaKoko;                 { Vaihtaa kaikk kenttien koot }
  public
    constructor Create(p:TWinControl);
    destructor Destroy; override;
    procedure Siivoa;                    { Poistaa kaikki kentät       }
    procedure Lisaa(s:string;data:TDataSource); { Lisää yhden kentän    }
    procedure LisaaKaikki(data:TDataSource);    { Lisää kaikki kentät   }
    procedure SetEditLeveys(l:integer);
    procedure SetNimenLeveys(l:integer);
    procedure SetKentanKorkeus(h:integer);
  end;


implementation

{-------------------------------------------------------------------------}
constructor cKentat.Create(p:TWinControl);
begin
  inherited Create;
  n := 0;
  kentan_korkeus := C_KENTAN_KORKEUS;
  nimen_leveys   := C_NIMEN_LEVEYS;
  edit_leveys    := C_EDIT_LEVEYS;
  panel          := p;
end;

{-------------------------------------------------------------------------}
procedure cKentat.Siivoa;
var i:integer;
begin
  for i:=0 to n-1 do begin
    kentat[i].nimi.Free; kentat[i].nimi := NIL;
    kentat[i].edit.Free; kentat[i].edit := NIL;
  end;
  n := 0;
end;

{-------------------------------------------------------------------------}
destructor cKentat.Destroy;
begin
  Siivoa;
  inherited Destroy;
end;
```

```
{-------------------------------------------------------------------------}
procedure cKentat.Paikka(var kentta:cKentta);
var korkeus:integer;
begin
  korkeus := kentan_korkeus;
  if ( korkeus <= 0 ) then korkeus := kentta.Edit.Height+1;
  with ( kentta.nimi ) do begin
    Left := 10;
    Width := nimen_leveys;
    Top := 10 + kentta.index*korkeus;
   end;
  with ( kentta.edit ) do begin
    Left := 10 + kentta.nimi.Left + nimen_leveys;
    Width := edit_leveys;
    Top := kentta.nimi.Top;
   end;
end;

{-------------------------------------------------------------------------}
procedure cKentat.MuutaKoko;
var i:integer;
begin
  for i:=0 to n-1 do
    Paikka(kentat[i]);
end;


{-------------------------------------------------------------------------}
procedure cKentat.Lisaa(s:string;data:TDataSource);
begin
  if ( n >= MAX_KENTTIA ) then exit;
  kentat[n].nimi := TLabel.Create(panel);
  with ( kentat[n].nimi ) do begin
    Caption := s;
    Parent := panel;
   end;
  kentat[n].edit := TDBEdit.Create(panel);
  with ( kentat[n].edit ) do begin
    DataSource := data;
    DataField := s;
    Parent := panel;
   end;
  kentat[n].index := n;
  Paikka(kentat[n]);
  inc(n);
end;

{-------------------------------------------------------------------------}
procedure cKentat.LisaaKaikki(data:TDataSource);
var i:integer;
begin
  Siivoa;
  for i:=0 to data.DataSet.FieldCount-1 do
    Lisaa(data.DataSet.Fields[i].FieldName,data);
end;

{-------------------------------------------------------------------------}
procedure cKentat.SetEditLeveys(l:integer);
begin
  edit_leveys := l;
  MuutaKoko;
end;

{-------------------------------------------------------------------------}
procedure cKentat.SetNimenLeveys(l:integer);
begin
  nimen_leveys := l;
  MuutaKoko;
end;
```

```
{-------------------------------------------------------------------------}
procedure cKentat.SetKentanKorkeus(h:integer);
begin
  kentan_korkeus := h;
  MuutaKoko;
end;


end.
```

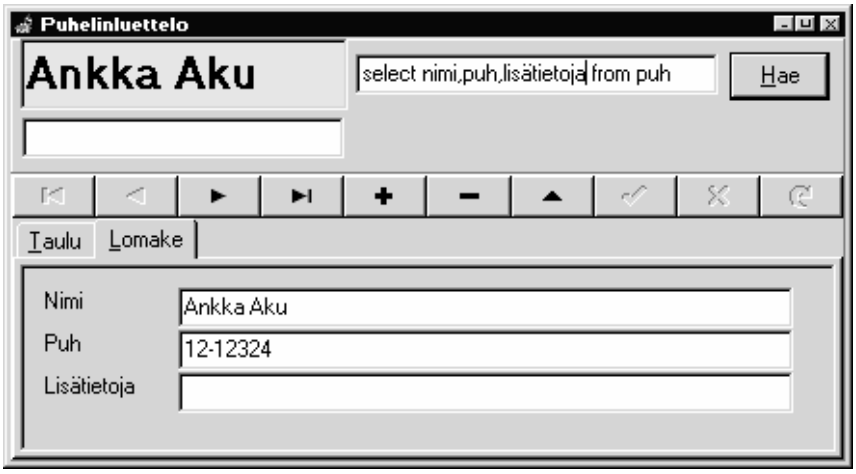
3.6.4 Data-kontrollien lisääminen dynaamisesti

Seuraavaksi lisätään luokan käyttö omaan lomakkeeseemme, harmaalla on esitetty koodiin edellisen kerran jälkeen tulleet muutokset:

puh.pas - muutokset dynaamisten kenttien käyttämiseksi

```
unit puh;
interface
uses ...
   dynkent;

type
  TFormPuh = class(TForm)
    ...
    ScrollBoxLomake: TScrollBox;
    procedure ButtonHaeClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ScrollBoxLomakeResize(Sender: TObject);
  private
    { Private declarations }
    kentat : cKentat;
    procedure TeeHaku(s:string);
  public
    { Public declarations }
  end;

var
  FormPuh: TFormPuh;

implementation


{$R *.DFM}
procedure TFormPuh.TeeHaku(s:string);
begin
  kentat.Siivoa;
  DBEditNimi.DataField := '';
  QueryPuh.Close;
  QueryPuh.SQL.Clear;
  QueryPuh.SQL.Add(s);
  QueryPuh.Open;
  DBEditNimi.DataField := QueryPuh.Fields[0].FieldName;
  kentat.LisaaKaikki(DataSourcePuh);
end;

procedure TFormPuh.ButtonHaeClick(Sender: TObject);
begin
  TeeHaku(EditHakuehto.Text);
end;

procedure TFormPuh.FormCreate(Sender: TObject);
begin
  kentat := cKentat.Create(ScrollBoxLomake);
  TeeHaku(EditHakuehto.Text);
end;
```

```
procedure TFormPuh.FormDestroy(Sender: TObject);
begin
  kentat.free;
end;

procedure TFormPuh.ScrollBoxLomakeResize(Sender: TObject);
begin
  kentat.SetEditLeveys(ScrollBoxLomake.ClientWidth-100);
end;

end.
```



Kuva 3.8 Dynaamisesti luodut DBEdit-kentät

3.7 Useiden tietokantataulujen yhdistäminen

Oikeassa ohjelmassa on lähes aina kyseessä relaatiotietokannat ja näin ollen useammasta taulusta saatavan tiedon yhdistäminen. Meidänkin esimerkissämme esimerkiksi `postiosoite` on itseään toistavaa tietoa ja kannattaisi ehkä käytännössä tehdä oma taulu, jossa olisi pelkkiä `postinumero`-`postiosoite` -pareja. Varsinaisessa päätaulussa ei sitten olisi `postiosoite`tta lainkaan, vaan postiosoite haettaisiin `postinumero`n avulla (relaatio) postiosoite-taulusta.

Tehtävä 3.26 Relaation hallitseminen *Delphi*llä

Kirjoita *DataFormExpert*in avulla ohjelma, joka hakee `postiosoitteen postinumero`n avulla `postisosoite`-taulusta.

Tehtävä 3.27 Monta puhelinnumeroa

Oikeastaan ei ole järkevää tehdä taulua, jossa on kentät `puh1`, `puh2` jne.. Voihan jollakin ihmisellä olla jopa 10 numeroa, mistä häntä pitää tavoitella. Suunnittele relaatiotietokanta, jossa jokaisella ihmisellä on mahdollisuus 0-n puhelinnumeroon. Pohdi ratkaisun tilankäyttöä normaalitilanteessa verrattuna meidän alkuperäiseen ratkaisuun. Kokeile toteuttaa taulut *DBD*:llä ja niitä käyttävä ohjelma *Delphi*llä.

3.8 Raportointi

Raportointi hoidetaan erillisillä komponenteilla, jotka ovat suuresti vaihdelleet eri Delphi-versioiden välillä.

Tehtävä 3.28 Raportit

Kokeile raportin tekemistä käyttämälläsi Delphin versiolla.

3.9 XML-tietokannat

Jos sovelluksen tietokanta on pieni ja on todennäköistä ettei siihen liity monia asiakkaita yhtä aikaa, kannattaa tietokanta tehdä ehkä XML-tiedostoksi. Etuna on silloin, että sovelluksen levittämisessä ei tarvitse jakaa raskasta BDE tms. tietokantaa lainkaan. Midas.dll on ainoa lisätiedosto sovelluksen ja XML-tiedoston lisäksi, joka pitää toimittaa käyttäjälle.

Alla on esimerkki puhelinluettelosta tehtynä `TClientDataSet`-komponenttia käyttäen.

PuhluForm.pas - tietokanta XML-pohjaiseksi

```
unit PuhluForm;
{
  Esimerkki siitä, kuinka käytetään tekstitiedostoa
  ClientDataSet-komponentin avulla.
  1) Lisää tarvittavat komponentit, erityisesti ClientDataSet
     (ks. alla)
  2) Liitä nyt tai myöhemmin DataSource ClientDataSet-komponenttiin
     ja näkyvät komponentit DataSourceen.
  3) Jos sinulla on valmis XML tms- tiedosto, niin laita se
     ClientDataSet-komponentin (jatkossa cds) FileNameen.
     Jos olet luomassa uutta, niin kirjoita FileNameen
     vaikka puh.xml (mielellään ei absoluuttista polkua).
  4) Lisää cds:n FieldDefs-kohdasta haluamasi kentät.
     Laita kenttien nimeksi selväkielisiä nimiä.  Jopa
     skandit kelpaavat.
  5) Kun olet luonut kaikki kentät, paina cds:ää oikealla napilla
     ja valitse Create DataSet.
  6) Laita ohjelma käyntiin ja lisäile tietueita.
  7) Voit halutessasi pistää cds:n Active := false ja jopa
     poistaa nyt kenttien määritelmät jolloin dfm-tiedosto
     pienenee.  Tällöin laita FormCreateen
       ClientDataSetPuh.Active := true;
  8) Toimita ohjelman mukana puh.xml-tiedosto käyttäjille.

  Vesa Lappalainen 16.9.2006
}

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, DBCtrls, Grids, DBGrids, ExtCtrls, DB,
  DBClient;

type
  TFormPuhlu = class(TForm)
    PanelOtsikko: TPanel;
    DBGridPuhlu: TDBGrid;
    DBNavigatorPuhlu: TDBNavigator;
    DBTextNimi: TDBText;
    DataSourcePuh: TDataSource;
    ClientDataSetPuh: TClientDataSet;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FormPuhlu: TFormPuhlu;

implementation

{$R *.dfm}

procedure TFormPuhlu.FormCreate(Sender: TObject);
begin
  ClientDataSetPuh.Active := true;
end;
```

```
end.
```

Esimerkkitiedosto:

```
<?xml version="1.0" standalone="yes"?>
<DATAPACKET Version="2.0">
<METADATA>
<FIELDS>
  <FIELD attrname="Nimi" fieldtype="string" WIDTH="30"/>
  <FIELD attrname="Puh" fieldtype="string" WIDTH="20"/>
  <FIELD attrname="Puh2" fieldtype="string" WIDTH="20"/>
  <FIELD attrname="Puh3" fieldtype="string" WIDTH="20"/>
  <FIELD attrname="Fax" fieldtype="string" WIDTH="20"/>
  <FIELD attrname="Osoite" fieldtype="string" WIDTH="30"/>
  <FIELD attrname="Postinumero" fieldtype="string" WIDTH="5"/>
  <FIELD attrname="Postiosoite" fieldtype="string" WIDTH="25"/>
  <FIELD attrname="Lisatietoja" fieldtype="string" WIDTH="50"/>
</FIELDS>
<PARAMS LCID="1053"/>
</METADATA>
<ROWDATA>
  <ROW Nimi="Ankka Aku" Puh="12-12324" Osoite="Ankkakuja 13"
     Postinumero="12345" Postiosoite="ANKKALINNA"/>
  <ROW Nimi="Ankka Tupu" Puh="12-12324" Osoite="Ankkakuja 13"
     Postinumero="12345" Postiosoite="ANKKALINNA"/>
  <ROW Nimi="Bond James" Puh="007" Osoite="Jamesstreet 007"
     Postinumero="007" Postiosoite="BOND"/>
  <ROW Nimi="Ponteva Veli" Puh="111-222" Osoite="Pontevankuja 1"
     Postinumero="12555" Postiosoite="PERÄMETSÄ"/>
  <ROW Nimi="Susi Sepe" Puh="111-111" Osoite="Sepesudentie 13"
     Postinumero="12555" Postiosoite="PERÄMETSÄ"
     Lisatietoja="Jahtaa possuja"/>
  <ROW Nimi="Veli Huilu"/>
  <ROW Nimi="Ääliö Älä lyö" Lisatietoja="ööliä läikkyy"/>
</ROWDATA>
</DATAPACKET>
```

Vikana siis tässä ratkaisussa on se, että jos samaa ”tietokantaa” haluaa päivittää useampi ohjelma yhtä aikaa, niin se ei kunnolla onnistu. Mutta jos kyseessä on todellakin yhden käyttäjän yksi ohjelma, niin tämä on hyvä ja kevyt ratkaisu, joka oikein tehtynä on helppo myöhemmin tarvittaessa muuttaa käyttämään oikeita tietokantoja.

Toki yhdenkin ohjelman sisällä voi tulla tarvetta esittää samasta datasta useita eri näkymiä. Tällöin menetellään niin, että jokaista taulua kohti luodaan yksi ”master” `TClientDataSet`-komponentti ja sitten jokaista erilaista näkymää kohti luodaan oma `TClientDataSet`-komponentti, joka kloonataan käyttämään pääkomponenttiaan.

```
...
DSKokoPuhlu        : TClientDataSet;
DSPostinmeroNakyma : TClientDataSet;
...

DSPostinumeroNakyma.CloneCursor(DSKokoPuhlu);
DSPostinumeroNakyma.Filter := ...

/// editoinnin jälkeen
DSKokoPuhlu.MergeChangeLog();
DSKokoPuhlu.SaveToFile();
...
```

Tällöin osanäkymä aina tiedottaa muutoksista pääkomponentilleen ja pääkomponentti jokaisella mahdolliselle muulle osanäkymälle. Tiedoston käsittely pitää muistaa hoitaa pääkomponentin kautta.

Mikäli käytössä on oikea tietokantapalvelin, niin silloin `TClienDataSet`-komponentti liitetään käyttämään tietokantapalvelinta pitemmän ketjun kautta:

```
  object SQLConnectionPuh: TSQLConnection
/// Tästä yhteys oikeaan tietokantapalvelimeen
...
  object SQLDataSetPuh: TSQLDataSet
    SQLConnection = SQLConnectionPuh
    CommandText = 'select * from PUH'
...
  object DataSetProviderPuh: TDataSetProvider
    DataSet = SQLDataSetPuh
...
  object ClientDataSetPuh: TClientDataSet
    ProviderName = 'DataSetProviderPuh'
```

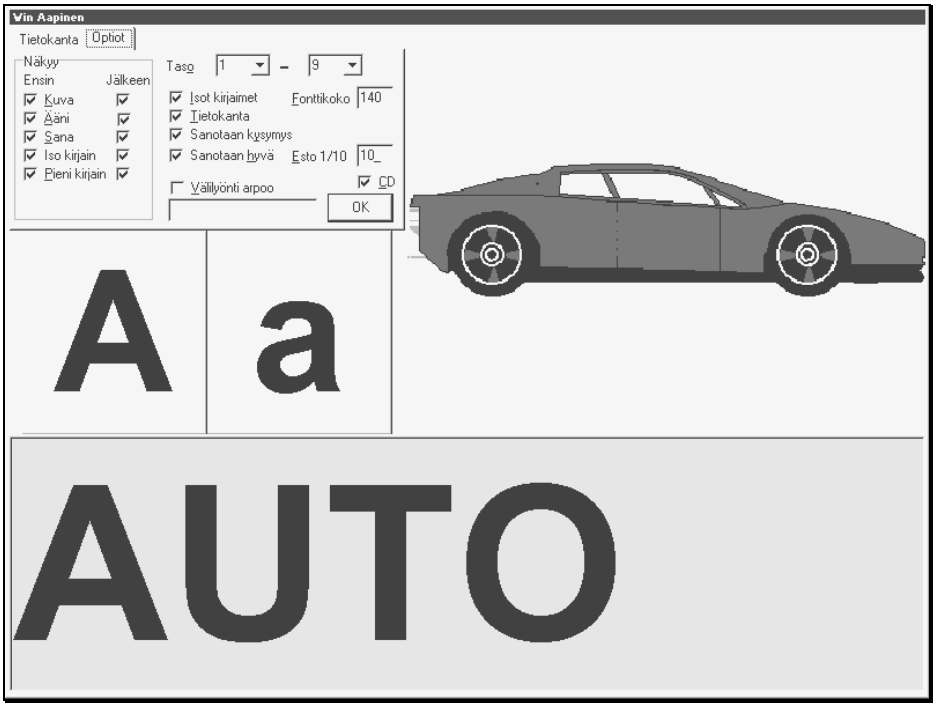
4. Multimediaa Delphillä

Luvun pääaiheet:

- **valmiiden .wav tiedostojen soittaminen**
- **kuvan näyttäminen, .bmb, .wmf**
- **beta-version koodit waapi/beta**
- **valmiin version koodit waapi-hakemistossa**

Tämän monisteen viimeisenä malliohjelmana toteutamme 1½-4 vuotiaille sopivan kirjainten opetteluohjelman: *WinAapinen*.

4.1 Lähtökohta



Kuva 4.1 WinAapinen 1.0

Ensimmäinen tavoite on tehdä ohjelma, joka painettaessa esimerkiksi kirjainta A, sanoo ääneen AUTO, näyttää ruudulla isoilla kirjoitetun tekstin AUTO, sekä lisäksi vielä auton kuvan. Ohjelman pitää kestää "jumiintumatta" 1½-vuotiaan käyttöä!

Toteutus on helpointa aloittaa siitä, mistä saataisiin puhuttavat äänet. Jos olisi käytössä puhesyntetisaattoriohjelma, voitaisiin äänet ottaa sieltä. Käytännössä on ehkä kuitenkin helpointa äänittää äänet itse. Usein äänikortin mukana tulee ainakin jokin ohjelma, jolla voidaan äänittää .wav-tiedostoja. Äänitetään aluksi muutama ääni jotta pääsemme alkuun ohjelman teossa. Esimerkiksi auto.wav, aiti.wav jne.

Kuvat järjestetään vastaavasti tiedostoihin auto.bmp, aiti.bmp jne. Myös .wmf (=*Windows MetaFile*) tiedostojen näyttäminen tulee onnistumaan.

4.1.1 Tietokanta

Miten sitten yhdistämme kirjaimet vastaaviin kuviin ja ääniin. Yksi mahdollisuus olisi tekstitiedosto, jossa olisi rivi:

```
A |auto.wav |auto.bmp  |auto
M |mummo.wav|          |mummo
O |ovi.wav  |ovi.bmp   |ovi
Ä |aiti.wav |          |äiti
```

Toisaalta saamme etsimisen "ilmaiseksi" jos annamme tietokantahaun tehdä sen. Luomme siis *DBD*:llä tietokantataulun waapi.db:

```
     Field Name              Type    Size     Key
-----------------------------------------------------
     Kirjain                 A         3       *
     Ääni                    A        20
     Kuva                    A        20
     Sana                    A        20
```

Taulu voidaan aluksi täyttää vaikkapa em. tietueilla.

4.2 Äänen soittaminen

4.2.1 Lyhyt esimerkki

Ennen kuin yhdistämme tietokantataulun ohjelmaamme, teemme pienen kokeen:

1. Avaa uusi projekti
2. Laita lomakkeelle:

```
Button:
  Name = ButtonSoita
  Caption = '&Soita'

MediaPlayer:  (system-sivu)
  Name = MediaPlayerWav
  AutoOpen = True
  FileName = auto.wav
```

3. Kirjoita Soita-nappulan tapahtumaksi:

```
procedure TForm1.ButtonSoitaClick(Sender: TObject);
begin
  MediaPlayerWav.Play;
end;
```

4. Kokeile ohjelman toimintaa.

Siinä kaikki! Periaatteessa ainoa mitä meidän siis tarvitsee tehdä, on sijoittaa `MediaPlayer` -komponentin `FileName` ominaisuuteen sen tiedoston nimi, jonka haluamme soittaa. Samalla komponentilla voimme soittaa myös .mid (midi-musiikki), .avi (audiovideo) yms. tieostoja. Kokeile!

4.2.2 Tietokantataulun lisääminen

Yhdistetäänpä tietokantataulu edelliseen esimerkkiin.

1. Ota nappula pois.
2. Nimeä lomake `Aapinen` ja otsikoksi `WinAapinen`
3. Talleta nimelle: unit: aapinen.pas ja projekti: waapi.dpr
4. Lisää seuraavat komponentit:

```
Query:
  Name = QueryAapinen
  SQL.Strings = 'select * from waapi'
  Active = True

DataSource:
  Name = DataSourceAapinen
  DataSet = QueryAapinen

DBEdit:
  Name = DBEditSana
  Width = ... iso ...
  CharCase = ecUpperCase
  Color = clYellow
  DataField = Sana
  DataSource = DataSourceAapinen
  Font.Height = ... iso ...
  Font.Name = 'Arial'
  Font.Style = fsBold
```

4.2.3 Painetun kirjaimen tunnistaminen

Mistä tahansa lomakkeella painetusta kirjaimesta saadaan viesti `OnKeyPress`. Kuitenkin *Windows* saattaa käsitellä suuren osan näppäinten painalluksista. Siksi pitääkin ensin muuttaa lomakkeen `Aapinen` ominaisuus `KeyPreview = True`. Näin kaikki painallukset tulevat ENSIN omalle ohjelmallemme ja vasta sitten joku muu saa käsitellä niitä.

4.2.4 Taulusta haetun äänen soittaminen

1. Kirjoitetaan lomakkeelle tapahtuman käsittelijä:

```
procedure TFormAapinen.FormKeyPress(Sender: TObject; var Key: Char);
var nimi,c:string; lkm:integer;
begin

  c := AnsiUpperCase(Key);

  QueryAapinen.Close;
  QueryAapinen.SQL.Clear;
  QueryAapinen.SQL.Add('select * from WAapi where kirjain = "' + c +'"');
  QueryAapinen.Open;
  lkm := QueryAapinen.RecordCount;
  if ( lkm = 0 ) then exit;

  nimi := QueryAapinen.Fields[1].AsString;
  if ( nimi = '' ) then exit;

  MediaPlayerWav.Filename := nimi;
  try
    MediaPlayerWav.Open;
    MediaPlayerWav.Play;
  except
    on EMCIDeviceError do ;
  end;

end;
```

2. Kokeile ohjelmaa

4.2.5 Tietueen kentän hakeminen nimen perusteella

Edellinen toteutus olisi arka sille, jos ääni-kenttä olisikin jokin muu kuin tietueen toinen kenttä (kenttien indeksit 0,1,2,3). Voitaisiin käyttää myös lausetta:

```
nimi := QueryAapinen.FieldByName('Ääni').AsString;
```

4.2.6 Erillisen kenttä-komponentin käyttö

Toisaalta jo ohjelman suunnitteluvaiheessa voitaisiin luoda oma kenttä-komponentti:

1. Tuplaklikkaa `QueryAapinen` komponenttia
2. Klikkaa aukeavaa dialogia hiiren oikealla näppäimellä
3. Valitse Add Fields
4. Valitse kaikki kentät aktiiviseksi.
5. Katso Object Inspectorista: olet saanut 4 uuttaa komponenttia:

```
QueryAapinenKirjain
QueryAapinenKuva
QueryAapinenni       (Ääni muuttui ni)
QueryAaapinenSana
```

6. Muuta `QueryAapinenni` nimeksi `QueryAapinenAani`.
7. Nyt äänitiedoston nimi saataisiin kutsulla:

```
nimi := QueryAapinenAani.AsString;
```

4.2.7 Suora hyppy taulun oikealle riville

Olisi vielä eräs mahdollisuus oikean äänen löytämiseksi: Ei suoriteta tietokantahakua uudelleen, vaan siirrytään aina kaikki tietueet sisältävässä taulussa oikealle riville:

1. Jos `QueryAapinen`-komponentin tilalla käytettäisiinkin `TableAapinen`-komponenttia, niin voitaisiin hakuehdon muodostamisen sijasta vain siirtyä:

```
TableAapinen.FindKey([c]);
```

Valitettavasti tämä ei toimi suoraan `Query`-komponentille. Toimintoa voidaan kuitenkin jäljitellä `Query`n `Filter`-ominaisuuden avulla.

Tehtävä 4.29 `Filter`

Toteuta äänen soittaminen siirtymällä taulukon oikealle riville.

4.3 Kuvan esittäminen

Ohjelmasta puuttuu vielä kuvan piirto:

1. Lisää komponentti:

```
Image   (  Additional-sivu )
  Name = ImageBmp
  koko = ...  mahdollisiman iso ...
```

2. Lisää painetun näppäimen käsittelykoodin loppuun:

```
nimi := QueryAapinenKuva.AsString;
if ( nimi = '' ) then exit;
ImageBmp.Picture.LoadFromFile(nimi);
```

3. Aja ohjelma :-)

Ohjelmassa on vielä se vika, että kuva ei aina välttämättä mahdu sille varattuun tilaan. Toisaalta pienet kuvat ovat todella pieniä. Asiaa voitaisiin auttaa laittamalla päälle ominaisuus `ImageBmp.Stretch`. Kokeile! Nyt kuitenkin kuvat ovat vääränmuotoisia. Ongelma voidaan korjata muuttamalla `ImageBmp`-komponentin kokoa samassa

suhteessa, mikä on kuvan alkuperäinen suhde. Nämä laskut on tehty ohjelman versiossa 1.0, joka löytyy monisteen lopussa olevasta liitteestä.

4.4 WinAapinen 1.0

Ohjelman "valmis" versio on hieman monimutkaisempi:

1. kutakin kirjainta kohti voi olla useita sanoja, painettua kirjainta vastaava sana arvotaan näistä
2. kirjaimilla tasonumerot, arvonta ottaa mukaan vain valitun tasonumerovälin täyttävät sanat
3. myös kuvaikkunassa voidaan soittaa .avi-elokuvia => tyypin mukaan pitää valita joko `MediaPlayer` tai `Image`
4. mahdollisuus "pelata" myös käänteisesti, eli ohjelma arpoo välilyöntiä painamalla sanan ja kysyy esimerkiksi: "Mistä tulee mummo". Tämän jälkeen pitää painaa m ennen kuin voi jatkaa.
5. voidaan valita mitä tulee näyttöön kun painetaan kirjainta ja mitä tulee näyttöön kun painetaan välilyöntiä
6. asetusten tallettaminen .ini-tiedostoon.
7. mahdollisuus soittaa CD-levyjä taustamusiikkina
8. fonttikoon vaihtaminen
9. systeemimenu jätetty pois
10. esto, jonka aikana ei voi painaa uutta näppäintä, mutta jonka jälkeen näppäimen painaminen katkaisee meneillään olevan soiton
11. erillinen editointi-ohjelma *WMuokkaa* tietokannan ylläpitoon

5. Omien komponenttien tekeminen

Luvun pääaiheet:

- **yksinkertainen mallikomponentti `Tlaskuri`**
- **muutama muu esimerkkikomponentti: `TEditPanel`, `TColorChange`**
- **koostamalla käytettävä luokka: `TIniSave`**
- **koodit comps/nimi-hakemistossa**

Lopuksi otetaan vielä pikainen katsaus omien komponenttien tekemiseen.

5.1 Miksi omia komponentteja

Visual Basicin vahvin puoli on erittäin helppo valmiiden komponenttien käyttäminen. Tämä ominaisuus on myös *Delphi*ssä. *Visual Basicissa* omien komponenttien tekeminen on todella työlästä (ennen .Net) ja vaatii *Windowsin* perus-API -ohjelmointia.

*Delphi*ssä omien komponenttien tekeminen on "lähes" samanlaista kuin *Object Pascalin* luokkien tekeminen ja voidaan siis tehdä ja testata *Delphi*llä.

Hyvin tehtyjen ja testattujen komponenttien avulla ohjelmointi on helppoa ja koodin kirjoittamisen tarve vähenee radikaalisti. Oikeastaan aina kun tekee uuden luokan, kannattaa miettiä olisiko siitä uudelleen käytettäväksi komponentiksi.

5.2 Yksinkertainen `TLaskuri`

Seuraavassa tehdään ensin yksinkertaistettu autolaskuriin kelpaavasta komponentista.

```
Kirjoittamisvinkki Delphi 4.0:
    1) Lisää ensin rivi:  property Count : integer;
    2) Paina Shift+Ctrl+C  (Complete Class at Cursor)
    3) Täydennä syntynyt SetCount -metodi.
    4) Kirjoita rivi: constructor Create(...
    5) Paina Shift+Ctrl+C
    6) Täydennä Create-metodi
    7) Täydennä puuttuvat virtual -määritykset
```

```
unit laskuri;

interface

uses SysUtils, Classes, Controls, Graphics,  StdCtrls;

type
  TLaskuri = class(TLabel)
  private
    FCount : integer;{ Ominaisuuksien attribuutit yleensä F-alkuisiksi}
  protected
    procedure SetCount(const Value: integer);  virtual;
  public
    constructor Create(AOwner:TComponent); override;
    function Inc(i:integer) : integer;     virtual;
  published  { Published declarations } { Yleensä vain ominaisuudet.    }
{ Ominaisuuksiin ilmoitetaan miten niitä luetaan ja miten asetetetaan   }
{ mahdollinen oletusarvo. Oletusarvo on ainoastaan ohje siitä, ettei    }
{ arvoa talleteta resurssitiedostoon, jos arvo on sama kuin oletus.     }
{ Luku/kirjoitusohje voi olla saman tyyppisen attribuutin nimi tai      }
{ aliohjelma asettamiseen ja funktio lukemiseen                         }
    property Count:integer read FCount write SetCount default 0;
end;
```

```
procedure Register;

implementation


constructor TLaskuri.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);     { Muista tämä!!! Muuten käy huonosti!!!}
  Count := 0;
end;


procedure TLaskuri.SetCount(const Value: integer);
begin
  FCount := Value;
  inherited Caption := IntToStr(Count)+' '; { Välilyönti lop. on par.näk}
end;


function TLaskuri.Inc(i:integer) : integer;
begin
  Count := Count + i;
  Result := Count;
end;


procedure Register;
{ Komponentit pitää rekisteröidä komponenttisivulle }
begin
  RegisterComponents('Samples', [TLaskuri]);
end;

end.
```

5.3 Väärinkäytön poistaminen:

Komponentissamme on vielä pieni vika: voidaan tehdä esimerkiksi sijoitus:

```
  lask.Caption := 10;
```

ja kun tämän jälkeen lisätään laskurin arvoa, ei seuraava arvo olekaan 11, vaan yksi suurempi kuin `FCount`-attribuuttiin on jäänyt edelliseltä kerralta.

Miten vika voidaan poistaa? Suositeltavin vaihtoehto olisikin periä luokka `TCustomLabel`. Ero `TLabel`-luokkaan on siinä, että `TCustomLabel`issa on kaikki samat ominaisuudet kuin `TLabel`issa, mutta `protected`-osassa, eli niitä ei voi ulkopuolinen käyttää. Tällöin em. sijoitus ei olisi mahdollista. Kuitenkin moni muukin `TLabel`in kiva ominaisuus jäisi suojatuksi. Ominaisuuksia voidaan siirtää kyllä julkaistulle puolelle kirjoittamalla:

```
published  // ks mallia TLabel-komponentin lähdekoodista StdCtrl.pas
    property Align;
    property Alignment;
    property Anchors;
    property AutoSize;
    property BiDiMode;
//    property Caption;  // Tätä ei julkaista
    property Color;
. . .
```

Yleensä kaikista komponenteista on ensin `TCustom`… -versio joka on tarkoitettu perittäväksi.

Toinen mahdollisuus on tehdä uusi `Caption`-ominaisuus, joka korvaa `TLabelin Caption`-ominaisuuden:

Lisää

```
published
. . .
  property Caption:string read GetCaption write SetCaption stored false;
```

ja tee vastaavat metodit (käytä luokkatäydennintä):

```
function TLaskuri.GetCaption: string;
begin
  Result := Trim(inherited Caption); { Varo rekursiota!!! }
end;

procedure TLaskuri.SetCaption(const Value: string);
begin
  Count := StrToIntDef(Trim(Value),0);
end;
```

Huomaa `inherited` avainsanan käyttö edeltäjäluokan ominaisuuden käyttämiseksi.

5.4 Oletusarvojen muuttaminen

Jos haluamme laskurimme tulevan valmiiksi isommalla fontilla oikeaan reunaan, voimme tehdä vielä pieniä parannuksia:

Lisää valmiiden ominaisuuksien uudet oletusarvot:

```
 published
. . .
    property Alignment        default taRightJustify;
    property AutoSize         default False;
    property Color            default clAqua;
    property ParentColor      default false;
```

Tämä on vasta tieto siitä, että jos ominaisuuden arvo sattuu olemaan sama kuin oletus, niin arvoa ei tarvitse tilan säästämisen vuoksi tallettaa lomakkeelle. Varsinainen arvon asetus pitää tehdä itse:

```
constructor TLaskuri.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);      { Muista tämä!!! Muuten käy huonosti!!!}
  Count := 0;                    { Defaulteissa luvatut arvot:          }
  Alignment := taRightJustify;
  Color := clAqua;
  ParentColor := False;
  Font.Height := -32;
  Font.Style := [fsBold];
  AutoSize := False;
end;
```

5.5 Komponentin ikonin piirtäminen

Jotta komponentti olisi tunnistettavissa työkalupalkissa, sille kannattaa piirtää ikoni:

1. Avaa Tools/Image editor
2. Tee uusi .dcr-tiedosto (*Dynamic Component Resource*)
3. Valitse `Contents` oikealla näppäimellä ja tee uusi 24x24 pxl-kokoinen *bitmap*.
4. Anna kuvalle nimeksi TLASKURI (kirjoita nimi isoilla!)
5. piirrä ikoni (tuplaklikkaa kuvan nimeä)

6. talleta tiedosto nimelle laskuri.dcr (olettaen että komponentti on nimellä laskuri.pas)

5.6 Komponentin lisääminen komponenttikirjastoon

Komponentti voidaan lisätä kirjastoon seuraavasti (Delphi 3.0-):

1. Kopioi laskuri.dcr samaan paikkaan kuin laskuri.pas (tänne on tehty komponenttisivulle tuleva bittikartta TLASKURI, 24x24, Huom! nimi isolla)
2. Valitse Component/Install Component
3. a) Jos haluamasi kirjasto on jo olemassa, valitse "Into Existing package" ja sitten Yes
4. b) Jos kirjastoa ei ole, valitse "Into New Package", kirjoita kirjaston nimi (esim. `lask`) ja kuvaus ja OK
5. Jos kirjastoa (*Package*) pitää korjailla, valitse esim. Component/Install Packages ja valitse kirjasto ja Edit. Korjausten jälkeen paina Compile ja tarvittaessa Install

Tehtävä 5.30 Uusi uljas autolaskuri

Tee uusi `Autolaskuri`-ohjelma ja käytä siinä `Laskuri`-komponentteja.

Tehtävä 5.31 `TLisaaButton`

Tee uusi komponentti `TLisaaButton`, jolla on
ominaisuutena

`Laskuri` : `TLaskuri` - laskuri, jonka arvoa lisätään aina, kun nappia painetaan

`Lisays` : `Integer` - paljollako laskuria lisätään

Tehtävä 5.32 Autolaskuri vähällä koodaamisella

Tee autolaskuri em. komponenteilla niin, että tarvitsee kirjoittaa vain yksi rivi koodia: `Nollaa`-nappulalle

Tehtävä 5.33 `TNollaa`

Tee vielä komponentti `TNollaa`, joka nollaa kaikki samalla lomakkeella olevat laskurit (ks. `ComponentCount` ja `Components`).

Tehtävä 5.34 Autolaskuri koodaamatta

Tee autolaskuri kirjoittamatta riviäkään koodia. Jos lisätään uusi laskuri-lisää -pari, niin tarvitaanko vieläkään koodia?

Tehtävä 5.35 Uusi `TNollaa`

*Modifioi `TNollaa` -komponenttia siten, että sille voidaan antaa ominaisuutena kaikki komponentit, jotka halutaan nollata.

Tehtävä 5.36 Laskuri panelista

Peri `TLaskuri` panelista ja aseta oletuksena sopivat reunukset päälle.

Tehtävä 5.37 Laskuri `TCustom`???-komponentista

Peri `TLaskuri TCustomLabelista` tai `TCustomPanelista` ja pidä huoli ettei `Caption`-ominaisuutta voi käyttää.

Tehtävä 5.38 `LiikkuvaAuto`-komponentti

Tee uusi komponentti `LiikkuvaAuto`, jolla on oma nopeus ja joka tutkii itse, milloin tulee seinä vastaan.

6. Tapahtumapohjainen ohjelmointi

Edellisessä luvussa saimme tehtyä ensimmäisen oman komponentin. Komponentin teko ei ollut kovin vaikeaa. Mutta komponentteja tehdessä pitää ehkä sittenkin huomioida vähän enemmänkin. Itse asiassa HYVÄN komponentin tekeminen on jo taidetta sinänsä. Pitäisi pystyä ennakoimaan mitä mahdolliset komponentin käyttäjät haluavat komponentiltamme. Tai pitäisi ainakin jättää mahdollisuus periä komponenttiamme siten, että halutut muutokset on mahdollista toteuttaa itse. Esimerkki hyvästä komponenttihierarkiasta on *Delphin VCL*-kirjasto. Sen lähdekoodeihin kannattaa tutustua.

Tehdäänpä autolaskuriin pieni lisäys: `Summa`-laskuri, jossa on kaikkien muiden laskureiden summa. Tietysti ensin lisäämme `Autolaskuri`-lomakkeelle:

```
LaskuriSumma : TLaskuri
```

seuraavaksi voi tulla mieleen tehdä nappuloihin painamisiin muutoksia:

```
procedure TFormAutolaskuri.ButtonHAClick(Sender: TObject);
begin
  LaskuriHA.Inc(1);
  LaskuriSumma.Count := LaskuriHA.Count + LaskuriKA.Count;
end;
```

Tämä tietysti toimii, mutta sama koodimuutos pitää tehdä kuorma-auton nappulaan ja myös `Nollaa`-nappulaan. Ensimmäisessä autolaskurissamme lisäsimme lopulta myös *Drag and Drop* -ominaisuuksia. Näihinkin pitäisi tehdä em. koodimuutos. Asiaa voidaan tietysti kiertää tekemällä lisäämistä ja muuttamista varten sopivia aliohjelmia (tai metodeja), jotka pitävät huolta myös summalaskurista. Mutta ennenpitkää koodiin tulee kuitenkin jokin muutos, jossa `LaskuriSumma` unohtuu päivittää.

Mikä on parempi tapa hoitaa ongelma? Jos laskurit itse kertoisivat omasta muutoksestaan, voitaisiin summalaskuria päivittää aina kun jokin laskuri muuttuu. Siispä toimeen:

Lisätään `TLaskuri`-komponentin koodiin tapahtuma:

```
TLaskuri = class(TLabel)
  private
. . .
    FOnChange: TNotifyEvent;
. . .
  published
. . .
    property OnChange : TNotifyEvent read FOnChange write FOnChange;
end;
```

Tämä tapahtuma tulee komponentin `Events`-välilehdelle, koska ominaisuuden tyyppinä on metodi. Tämä lisäys ei vielä yksin riitä. Meidän täytyy myös pitää huoli siitä, että tapahtuman käsittelijää kutsutaan siellä missä on tarkoituskin. Jos `TLaskuri` komponentti on tehty hyvin, ei laskurin arvoa muuteta muualla kuin yhdessä paikassa. Onneksi meidän kohdallamme näin on:

```
procedure TLaskuri.SetCount(const Value: integer);
begin
  FCount := Value;
  inherited Caption := IntToStr(Count)+' '; { Välilyönti lop. on par.näk}
  if ( Assigned(OnChange) ) then OnChange(self);
end;
```

Eli jos joku on halunnut tapahtuman käsiteltäväksi (`Assigned(OnChange)`), kutsutaan sitä metodia, johon `OnChange` -metodiosoitin osoittaa (`OnChange(self)`).

Itse autolaskuri toimii vielä kuten ennenkin, mutta jos klikkaamme (muista kääntää komponentti uudelleen) `LaskuriHA`:n `Events`-välilehdellä `OnChange` -tapahtumaa ja kirjoitamme koodin:

```
procedure TFormAutolaskuri.LaskuriHAChange(Sender: TObject);
begin
  LaskuriSumma.Count := LaskuriHA.Count + LaskuriKA.Count;
end;
```

niin johan rupesi laskemaan. Sijoitetaan sama tapahtuma vielä `LaskuriKA`:n `OnChange` -tapahtumaksi.

Saavutettu hyöty on nyt siinä, että muutettiinpa laskureita millä tavalla tahansa, niin muutos tulee aina käsiteltyä. Pienenä miinuksena on se, että jos lisätään esim. `LaskuriPP`, niin tällöin summan laskukaavaa pitää muuttaa.

Voisimme tehdä myös erilaisen tapahtuman. Tapahtuman, jossa kerrotaan itse laskurin lisäksi muutoksen suuruus. Tällaista tapahtumaosoitinta ei ole tietenkään valmiina, mutta voimme sellaisen tehdä itsekin:

```
type
  TLaskuri = class;

  TLaskuriEvent = procedure (sender:TLaskuri; diff:integer) of object;

  TLaskuri = class(TLabel)
  private
    FCount : integer;
    FOnChange: TLaskuriEvent;
. . .
  published
. . .
    property OnChange : TLaskuriEvent read FOnChange write FOnChange;
  end;
```

Ja tietysti muutos itse laskurin asettamiseen:

```
procedure TLaskuri.SetCount(const Value: integer);
var diff : integer;
begin
  diff := Value - Count;
  FCount := Value;
  inherited Caption := IntToStr(Count)+' '; { Välilyönti lop. on par.näk}
  if ( Assigned(OnChange) ) then OnChange(self,diff);
end;
```

Nyt autolaskurissamme riittäisi koodi:

```
procedure TFormAutolaskuri.LaskuriHAChange(sender: TLaskuri;
                                           diff: Integer);
begin
  LaskuriSumma.Inc(diff);
end;
```

ja kun kaikki summaan halukkaat laskurit ilmoittavat saman tapahtuman, niin summalaskuri pysyy ajantasalla vaikka lisäisimme kuinka monta laskuria.

Tehtävä 6.39 Nollautuuko

Nollautuuko summalaskuri kuin painetaan `Nollaa`-nappulaa? Miksi?

Tehtävä 6.40 Yläraja

Lisää laskuriin yläraja-ominaisuus ja tapahtuma kun yläraja ylitetään.

Tehtävä 6.41 `OnBeforeChange`

Lisää laskuriin `OnBeforeChange` -tapahtuma, joka suoritetaan, ennenkuin arvoa muutetaan. Parametrina tapahtumalle on mm. `var Accept:boolean`, johon tapahtumankäsittelijän tulee sijoittaa arvo, sen mukaan sallitaanko muutos vai ei.

6.1 `TLaskuri`

```
unit laskuri;

interface

uses SysUtils, Classes, Controls, Graphics,  StdCtrls;

type
  TLaskuri = class;

  TLaskuriEvent = procedure (sender:TLaskuri; diff:integer) of object;

  TLaskuri = class(TLabel)
  private
    FCount : integer;
    FOnChange: TLaskuriEvent;
    function GetCaption: string;
    procedure SetCaption(const Value: string);
  protected
    procedure SetCount(const Value: integer);  virtual;
  public
    constructor Create(AOwner:TComponent); override;
    function Inc(i:integer) : integer;     virtual;
  published  { Published declarations } { Yleensä vain ominaisuudet.   }
    property Count:integer read FCount write SetCount default 0;
    property Caption:string read GetCaption write SetCaption
                                                    stored false;
    property OnChange : TLaskuriEvent read FOnChange write FOnChange;
    { Seuraavat ominaisuudet isältä, ainoastaan oletusarvot erilaiset   }
    property Alignment        default taRightJustify;
    property AutoSize         default False;
    property Color            default clAqua;
    property ParentColor      default false;
  end;

procedure Register;

implementation
```

```
constructor TLaskuri.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);     { Muista tämä!!! Muuten käy huonosti!!!}
  Count := 0;                   { Defaulteissa luvatut arvot:          }
  Alignment := taRightJustify;
  Color := clAqua;
  ParentColor := False;
  Font.Height := -32;
  Font.Style := [fsBold];
  AutoSize := False;
end;


procedure TLaskuri.SetCount(const Value: integer);
var diff : integer;
begin
  diff := Value - Count;
  FCount := Value;
  inherited Caption := IntToStr(Count)+' '; { Välilyönti lop. on par.näk}
  if ( Assigned(OnChange) ) then OnChange(self,diff);
end;


function TLaskuri.Inc(i:integer) : integer;
begin
  Count := Count + i;
  Result := Count;
end;


procedure Register;
{ Komponentit pitää rekisteröidä komponenttisivulle }
begin
  RegisterComponents('Samples', [TLaskuri]);
end;

function TLaskuri.GetCaption: string;
begin
  Result := Trim(inherited Caption); { Varo rekursiota!!! }
end;

procedure TLaskuri.SetCaption(const Value: string);
begin
  Count := StrToIntDef(Trim(Value),0);
end;


end.
```

Kirjallisuutta

Becks, Ari: *Delphi,* Sovelluksentekijän opas, *Suomen ATK-kustannus OY, 1996*

Borland: Delphi manuals - *Borland International Inc, 1996*

Borland: Component Writer's Guide - *Borland International Inc, 1996*

Cantù, Marco *:* Mastering Delphi 6, *SYBEX, 2001*

Järvinen Jani, Piispa Juha*:* Delphi, Sovellusten Opas - *Docendo 2000*

Hakemisto