LAZARUS

Írta: Kálcza Tamás

Nagy Balázs

Tartalomjegyzék

ELŐSZÓ A JEGYZETHEZ **5**

A FEJLESZTŐKÖRNYEZET BEMUTATÁSA **6**

Konvertálók és egyéb függvények, eljárások 6
A fájlkezelés változásai 7
A komponensekről általánosan 8
A Lazarus kezelőfelülete 9
Ha túl nagy az exe 11

AZ OBJEKTUM ORIENTÁLT PROGRAMOZÁS LÉNYEGE **12**

Objektum vagy osztály 12
Ábrázolás 13
Objektum születése 13
Objektumok megsemmisítése 13
Tagfüggvények 14
Betokozás 14
Öröklődés 15
Többalakúság 16
Zárszó 17

A LÁTHATÓ KOMPONENSEK TULAJDONSÁGAI (PROPERTIES) **18**

Align (egyenesbe hozás) 18
Anchors (horgonyok) 18
AutoSize (automatikus méretezés) 18
Caption (felirat) és Text (szöveg) 18
Color (szín) 19
Constraints (korlátok) 19
Cursor (egérmutató) 19
Enabled (engedélyezettség) 19
Font (betűkészlet) 19
Height (magasság) és Width (szélesség) 19
Top és Left (bal felső pont-koordináták) 20
Hint (útmutatás) és ShowHint (útmutató engedélyezése) 20
Name (komponensnév) 20
Parent-tulajdonságok (szülőtulajdonságok engedélyezése) 20
TabOrder (Tab-sorrendbeli hely) 20
TabStop (Tab-kirekesztés) 20
Tag (cédula) 20
Visible (láthatóság) 20

AZ LEGGYAKORIBB ESEMÉNYEK (EVENTS) **21**

OnClick (kattintáskor) 21
OnEnter (kiválasztáskor) és OnExit (elváltáskor) 21
OnKeyPress (billentyű lenyomásakor) 22
OnMouseDown (egérgomb lenyomásakor) 22
OnMouseUp (egérgomb felengedésekor) 22
OnMouseMove (egér mozgatásakor) 22
OnCreate (létrehozáskor) 23
OnChange (változáskor) 23

A FORMOK **24**

A formok – Tulajdonságok (Properties) 25
A formok – Használatuk 26

A STANDARD-PALETTA (ALAPKOMPONENSEK) **30**

A főmenü (TMainMenu) – Bevezető 30
A főmenü – Tulajdonságok 31

A főmenü – Elemeinek tulajdonságai 31
A felugró menü (TPopupMenu) 33
A gomb (TButton) – Bevezető 34
A gomb – Tulajdonságok 34
A címke (TLabel) – Bevezető 35
A címke – Tulajdonságok 35
Az egyszerű szövegmező (TEdit) – Bevezető 36
Az egyszerű szövegmező – Tulajdonságok 36
A jegyzettömb (TMemo) – Bevezető 36
A jegyzettömb – Tulajdonságok 36
A jelölőnégyzet (TCheckBox) – Beveztő 37
A jelölőnégyzet – Tulajdonságok 37
A választógomb (TRadioButton) 38
A listadoboz (TListBox) – Bevezető 38
A listadoboz – Tulajdonságok 38
A kombinált lista (TComboBox) – Bevezető 39
A kombinált lista – Tulajdonságok 39
A csoportmező (TGroupBox) – Bevezető 40
A csoportmező – Tulajdonságok 40
A választógomb csoportmező (TRadioGroup) – Bevezető 40
A választógomb csoportmező – Tulajdonságok 41
A jelölőnégyzet csoportmező (TCheckGroup) 41
Az eseménylista (TActionList) – Bevezető 41
Az eseménylista – Tulajdonságok 42

AZ ADDITIONAL-PALETTA (KIEGÉSZÍTŐK) 43

A képgomb (TBitBtn) – Bevezető 43
A képgomb – Tulajdonságok 43
A gyorsgomb (TSpeedButton) – Bevezető 45
A gyorsgomb – Tulajdonságok 45
A képkezelő (TImage) – Bevezető 46
A képkezelő – Tulajdonságok 46
A képkezelő – Használata 47
A címkézett beviteli mező (TLabeledEdit) – Bevezető 53
A címkézett beviteli mező – Tulajdonságok 53
A maszkolható beviteli mező (TMaskEdit) – Bevezető 54
A maszkolható beviteli mező – Tulajdonságok 54
A maszkolható beviteli mező – Használata 55
A szöveges táblázat (TStringGrid) – Bevezető 56
A szöveges táblázat – Tulajdonságok 56
A szöveges táblázat – Használata 58

A COMMON CONTROLS (GYAKORI VEZÉRLŐK) 59

A csúszka (TTrackBar) – Bevezető 59
A csúszka – Tulajdonságok 59
A folyamatsáv (TProgressBar) – Bevezető 60
A folyamatsáv – Tulajdonságok 61
A fastruktúra (TTreeView) – Bevezető 61
A fastruktúra – Tulajdonságok 61
A fastruktúra – Használata 63
Az állapotsáv (TStatusBar) – Bevezető 65
Az állapotsáv – Tulajdonságok 65
Az állapotsáv – A panelek tulajdonságai 66
A fel-le gombcsoport (TUpDown) – Bevezető 66
A fel-le gombcsoport – Tulajdonságok 66
A többlapos form (TPageControl) – Bevezető 67
A többlapos form – Tulajdonságok – Tabs (Fülek): 67
A többlapos form – Tulajdonságok – TabSheets (Lapok): 68
A képlista (TImageList) – Bevezető 68
A képlista – Tulajdonságok 68

A DIALOG-OK (PÁRBESZÉDABLAKOK).. 70

A megnyitóablak (TOpenDialog) – Bevezető.. 70
A megnyitóablak – Tulajdonságok 70
A megnyitóablak – Használata... 71
A mentésablak (TSaveDialog) – Bevezető 72
A mentésablak – Tulajdonságok.. 72
A mentésablak – Használata 72

Előszó a jegyzethez

A jegyzet a Lazarus nevű fejlesztői környezet rövid ismertetését tűzi ki céljául. Vagyis nem egy teljes, átfogó írás, hanem a legfontosabb komponensekről, s azok lényeges elemeiről szól. Bemutatjuk a gyakori tulajdonságokat, szó lesz az eseményekről, és érintőlegesen egy kis objektum-orientált programozásról. Igyekeztünk érthetően, lényegre törően fogalmazni, és példákkal segíteni a megértést.

A Lazarus egy ingyenesen használható, és terjeszthető program, fordítója egy Turbo Pascal klón, a FreePascal, tehát a programkód szintaktikája megegyezik az eddig tanultakkal. Továbbá maga a program is egy klón, méghozzá az ismert fejlesztői környezeten a Delphi-n alapszik. A jegyzet megírásához éppen ezért felhasználtuk a Delphi program súgóját, valamint Marco Cantu „Delphi 7 mesteri szinten" című könyvét. Így az olvasó találkozhat olyan részekkel a jegyzetben, hogy bizonyos Lazarus-beli elemek ugyan ismertetésre kerültek, de megjegyzésként odaírtuk, hogy nem vagy hibásan működik. Ennek oka az volt, hogy a Lazarus folyamatos fejlesztés alatt áll, stabil verziója még nem jelent meg, de az újabb változatokban természetesen igyekeznek a fejlesztők kijavítani a hibákat, hiányosságokat. Sajnos még így is gyakori a program összeomlása, sokszor érthetetlen hibaüzeneteket ad. Mindazon által a Lazarus a Delphi-hez képest több szempontból is előre mutat, például több platformon elérhető (Linux, OS X, Windows, stb.).

Ez a fejlesztői környezet 4GL típusú (Fourth-generation programming language), így segítségével nem csak konzolos (lásd Pascal), hanem vizuális kezelőfelülettel rendelkező programokat készíthetünk. Maga a programírás is egy ilyen vizuális felülettel történik, mely előre megalkotott elemeket, úgynevezett komponenseket tartalmaz. Ezáltal leegyszerűsödik sok feladat megoldása (pl. adatok bekérése, fájlok megnyitása), és felhasznláló-barátabb alkalmazásokat írhatunk.

A jegyzetet a Lazarus 0.9.14 és 0.9.16 béta verziója alapján készítettük, s főként a Programozás Módszertan 3 és 4, valamint az Informatika Alkalmazás Módszertana 1 és 2 tárgyakhoz ajánljuk. Reméljük, hasznosnak találod majd, s a Lazarus-ra való átállást kissé zökkenő mentesebbé tesszük számodra.

Sok sikert kívánnak a jegyzet írói!

Linkek:

A fejlesztők honlapja
A Lazarus letöltése

A fejlesztőkörnyezet bemutatása

A jegyzet ebben a fejezetében a Lazarus felépítésével, fontosabb beállításaival, függvényeivel és eljárásaival fogunk foglalkozni, de szó lesz még néhány típusról is. Mint említettük, a Lazarus fordítója a Free Pascal, így sok olyan elemet használhatunk fel programjaink készítése során, melyet Turbo Pascal-ban már megismertünk (főbb típusok, mint például Integer, String, függvények, mint a Copy, vagy akár az egész típusú változókat növelő és csökkentő eljárások, az Inc és a Dec). De a Lazarus nem csak külalakra lesz más, hanem alapvetően változik meg programkészítés módja, és ehhez új eszközöket is kapunk.

- Konvertálók és egyéb függvények, eljárások
- A fájlkezelés változásai
- A komponensek általánosan
- A Lazarus kezelőfelülete
- Ha túl nagy az exe

❖ ***Konvertálók és egyéb függvények, eljárások:***

1) Konvertálók:

Ilyen új eszköz többek között, néhány beépített függvény és eljárás. Ezek nagy részét a gyakori típusok között konvertáló függvények alkotják. Szintaktikai felépítésük megegyezik: Függvénynév(mit:BemenetTípus):KimenetTípus. Természetesen mindig előfeltétel, hogy a bemenet helyes legyen! A konvertáló függvényekből leggyakrabban a „szöveg-egyéb” és az „egyéb-szöveg” típusúak használatosak, ugyanis a grafikus felület miatt a kiíratás és beolvasás csak szöveges formában történhet, nem úgy, mint a karakteres felületű (konzolos) Pascalban. Ezek közül válogattunk ki néhányat:

Függvény szintaktika	Leírás
StrToInt(**Const** s:**String**):Integer	Szövegből alakít egész számmá.
StrToFloat(**Const** s:**String**):Extended[1]	Szövegből alakít valós számmá.
StrToDateTime(**Const** s:**String**):TDateTime[2]	Csak dátumot és/vagy időt tartalmazó szövegből alakít dátum-idő típusúvá.
StrToDate(**Const** s:**String**):TDateTime	Csak dátumot tartalmazó szövegből alakít dátum-idő típusúvá.
StrToTime(**Const** s:**String**):TDateTime	Csak időt tartalmazó szövegből alakít dátum-idő típusúvá.
IntToStr(value:Integer):**String**	Egész típusú adatot alakít szöveggé.
FloatToStr(value:Extended):**String**	Valós típusú adatot alakít szöveggé.
DateTimeToStr(DateTime:TDateTime):**String**	Dátum-idő típusú adatot alakít dátumot és időt is tartalmazó szöveggé.
DateToStr(Date:TDateTime):**String**	Dátum-idő típusú adatot alakít csak dátumot tartalmazó szöveggé.
TimeToStr(Time:TDateTime):**String**	Dátum-idő típusú adatot alakít csak időt tartalmazó szöveggé.

1: Az Extended (kiterjesztett) típus egy előjeles valós számtípus, a legbővebb ilyen a Lazarus-ban. A szokásos Real típusba is be lehet olvasni ezzel a függvénnyel, hiszen az Extended nagyobb intervallumot foglal magába.
2: A TDateTime valójában egy valós szám, az Extended-nél kisebb, a Real-nél viszont bővebb Double típusú. A szám egész része az 1899. december 30. óta eltelt napok száma. A negatív számok értelemszerűen az alapdátum előtti időpontokra hivatkoznak. A szám tört része mutatja az időt. Például a 38961,75 értékének megfelelő dátum és időpont: 2006. szeptember 1. 18 óra, a nap háromnegyede.

2) Egyszerű felugró ablak (ShowMessage):

Gyakran fordul elő, hogy szeretnénk a program futása közben értesíteni a felhasználót valamilyen információról, amit megtehetünk úgy is, hogy egy felugró ablakban jelenítjük meg ezt a tájékoztatást. Ilyen ablakot a *ShowMessage* eljárással is létrehozhatunk. Az eljárásnak csupán egy bemeneti paramétere van: az a szöveges információ, amit közölni szeretnénk a felhasználóval.

Az ablakot meghívó eljárás forráskódja:

```
procedure Probaablak;
begin
   ShowMessage('Kilépéshez nyomja meg az "OK" gombot!');
end;
```

3) Ablakok bezárása (Close):

Egy programablak kódból történő bezárásához kell meghívnunk ezt az eljárást. Ha a főablakot (lásd a formoknál) bezárjuk, magát a programot is bezárjuk, míg ha egy más típusú ablakot, akkor az alkalmazás továbbra is futni fog. Használata egyszerű, csak meg kell hívni a *Close* eljárást minden paraméter nélkül.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Close;
end;
```

4) Késleltetés (Delay és Sleep):

Gyakran az okoz problémát, hogy a számítógépünk túl gyorsan számol ki valamit, így csak a végeredményt látjuk, a folyamatot nem. Például, ha egy populációgenetikai szimulálós programot írunk, aminek a grafikonja az aktuális évben élő egyedek korbeli összetételét ábrázolja, a program másodpercenként ugorhat akár évszázadokat is a modell bonyolultságától függően. A megoldást a késleltetés eljárás jelentheti. A Lazarus-ban két ilyen eljárás található: a *Sleep* és a Crt unitban levő *Delay*. Közös bennük, hogy mind a kettőnél egy egész szám típusú paramétere van az eljárásnak, mégpedig az, hogy mennyi ideig kell a programnak várakoznia. Ezt az értéket milliszekundumban kell megadnunk. A Delay esetében 0-65535 milliszekundum lehet várakozási idő, míg a Sleep-nél a megadható idő 0-4294967295 milliszekundum.

Alább egy olyan program forráskód részletét látjuk, ami egy gombnyomásra 10 másodpercig várakozik, majd kiírja, hogy „Letelt az idő”:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Sleep(10000);
   ShowMessage(’Letelt az idő.’);
end;
```

❖ ***A fájlkezelés változásai:***

A Pascalhoz képest apróbb változás történt két fájlkezelő eljárás nevében és a szöveges fájl típusának nevében. A megnyitott fájlok bezárására a Close helyett a *CloseFile*, egy fájl és nevének összekapcsolására az Assign helyett az *AssignFile* eljárásokat kell használnunk. Az egyszerű szöveges fájl típusánál Text helyett TextFile-t kell írni. A változásokat alább egy példával is illusztráltuk:

```
Procedure Pelda(Var f:TextFile; Const fNev:String);
Begin
   AssignFile(f,fNev);
   {$i-}
   Reset(f);
   {$i+}
   If IOResult=0 then begin
      ShowMessage('Sikeres megnyitás!');
      CloseFile(f);
      end
      else ShowMessage('Nincs ilyen nevű fájl!');
End;
```

❖ ***A komponensekről általánosan:***

Ebben a programozói környezetben programjaink alapjait, építőköveit alkotják majd a komponensek. Korábban egy konkrét feladatkörhöz tartozó típusokat, eljárásokat, függvényeket unitokba csoportosítottuk, így valósítva meg azok újrahasznosíthatóságát. A komponensek is ezt a célt fogják szolgálni speciális felépítésük révén. Minden ilyen alkotóelemnek saját felhasználási területe van: egyesek megjelenítésre, mások adatbevitelre vagy éppen vezérlésre szolgálnak. Továbbá tartoznak hozzájuk olyan tulajdonságok és függvények, melyek e speciális feladatok elvégzését teszik lehetővé. Alapvetően kétféle komponensről beszélhetünk: látható és nem látható. Ez azt jelenti, hogy egy részük az alkalmazás futásakor is megjelenhet a programablakban, mert tartozik hozzájuk vizuális felület, amely ezt lehetővé teszi. Másik részük ilyennel nem rendelkezik, így ők közvetlenül nem láthatók.

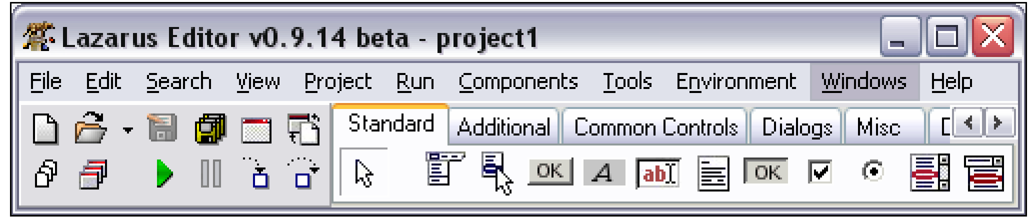
A Lazarus a leggyakrabban felmerülő és komplexebb problémák megoldásához is tartalmaz előre elkészített komponenseket. Mi ezek közül emeltünk ki néhányat, melyek saját tapasztalataink szerint fontosak. A jegyzet további részében a Lazarus által használt csoportosítást követtük, most azonban feladatkörük alapján gyűjtjük össze őket:

- ❖ Szövegbeviteli komponensek
  - ➢ Egyszerű szövegbeviteli mező
  - ➢ Címkézett beviteli mező
  - ➢ Maszkolható beviteli mező
  - ➢ Jegyzettömb
  - ➢ Táblázat
- ❖ Megjelenítésre használt komponensek
  - ➢ Címke
  - ➢ Folyamatsáv
  - ➢ Állapotsáv
  - ➢ Fastruktúra
  - ➢ Képkezelő és rajzoló
- ❖ Csoportosításra való komponensek
  - ➢ Csoportmező
  - ➢ Választógombos csoportmező
  - ➢ Jelölőnégyzetes csoportmező
  - ➢ Többlapos form
  - ➢ Képlista
- ❖ Párbeszédablakok
  - ➢ Megnyitóablak
  - ➢ Mentőablak
- ❖ Vezérlésre alkalmas komponensek
  - ➢ Főmenü
  - ➢ Felugró menü
  - ➢ Gomb
  - ➢ Képgomb
  - ➢ Gyorsgomb
  - ➢ Választógomb
  - ➢ Jelölőnégyzet
  - ➢ Listadoboz
  - ➢ Lenyíló lista
  - ➢ Eseménylista
  - ➢ Csúszka
  - ➢ Fel-le gomb

❖ ***A Lazarus kezelőfelülete:***

1) <u>Főablak:</u>

A Lazarus alapértelmezésben öt ablakkal indul. Közülük az egyik a főablak, ahol a menüsor, a vezérlőgombok és a komponenspaletta is található:



A File-menüben a programoknál általában megszokott menüpontokat találjuk: új elemek létrehozása (esetünkben ez unitot, és formot takar). Ezeket a „New…”-ra kattintva a felugró ablak listájából is elérhetjük. Ezeken kívül itt tudunk új alkalmazást is létrehozni a „Program/Application” részre klikkelve. Az elkészített programok kimentésére a „Save All” (Ctrl+Shift+S) menüpontot válasszuk, így egyszerre menthetjük minden elemet (unit, project). A későbbiekben persze elég lehet csak egy unit mentése a sima „Save”-vel (Ctrl+S).

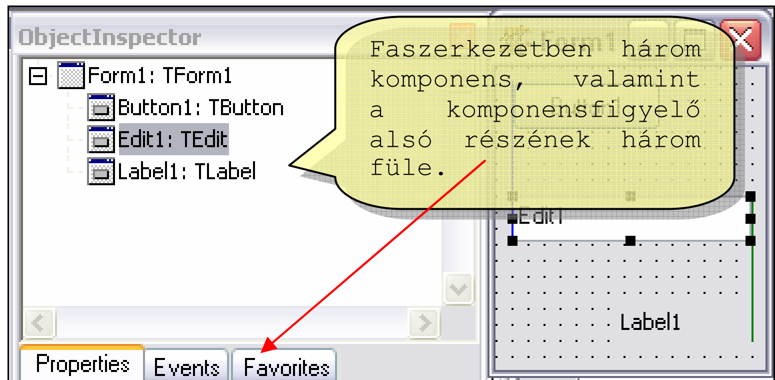
A Run-menüben megtalálhatjuk a Pascalból is ismert, futtatással kapcsolatos menüpontokat. Egy fontos menüpontra hívnánk fel a figyelmet, ez a „Reset debugger”. Mivel a Lazarus-nak még nincs stabil verziója, olykor előfordulhat, hogy szükség van a nyomkövető program (debugger) újraindítására. Ennek hatására az éppen futó (tehát vizsgált, nyomon követett) program leáll.

A Windows-menüben tudunk a Lazarus ablakai közt váltani.

A menüsor alatt található, fülekkel tagolt rész a komponenspaletta. Innen válogathatjuk majd ki az „építőelemeket”, melyekből majd programjaink felépülnek. Szintén a menüsor alatt, a komponenspaletta bal oldalán találhatók a vezérlőgombok, melyekkel a menü gyakran használt elemei érhetők el (pl. a zöld háromszög a Run, a három egymáson lévő floppy a SaveAll).

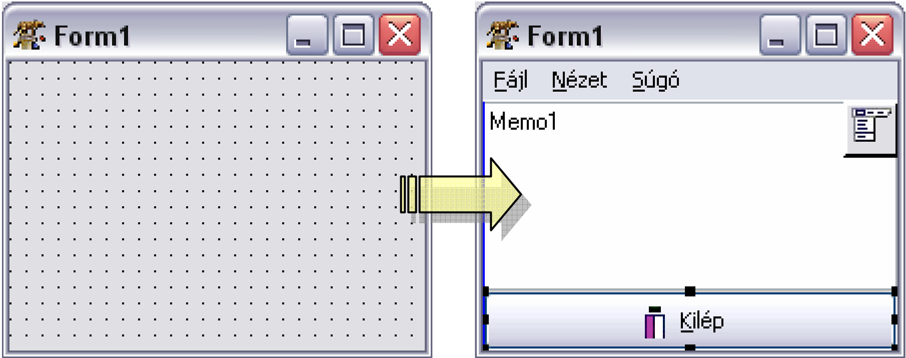
2) <u>Object Inspector (komponensfigyelő):</u>

Egy másik programablak a komponensfigyelő, mely két részre tagolódik. A felső részen a formra helyezett komponenseket tekinthetjük meg fastruktúrába rendezve. Az alsó fele további három részre oszlik: Properties (tulajdonságok), Events (események) és Favorites (kedvencek). Az első kettőnek fontosságuk miatt egy külön fejezetet szenteltünk. A kedvencekbe az előbbi kettő gyakran használt elemeit válogathatjuk ki.



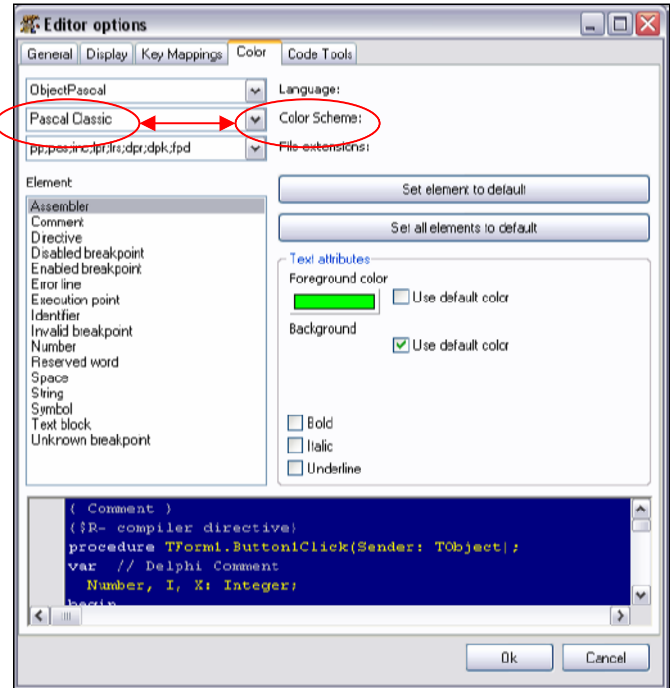
3) <u>Formtervező:</u>

A formtervezővel alakíthatjuk ki programunk ablakainak arculatát úgy, hogy komponenseket helyezünk el rajta, majd azokat a kívánt helyre mozgatjuk, esetleg módosítjuk egyes tulajdonságaikat. Kattintsunk a kiválasztott komponensre, majd a formra: így már is elhelyeztük az elemet. Ezután már egérrel húzva mozgathatjuk a formon, a komponensfigyelőben pedig szerkeszthetjük, módosíthatjuk beállításait.



4) <u>Kódszerkesztő (Source Editor):</u>

Itt írhatjuk a programkódot. A Lazarus, ahogy a Pascal is, színekkel segíti az olvashatóságot. Ezeket az Environment-menü „Editor options” menüelem Color fülén tudjuk beállítani. A „Color schemes”-nél előre beállított színsémák közül választhatunk. Még Pascal-séma is van:

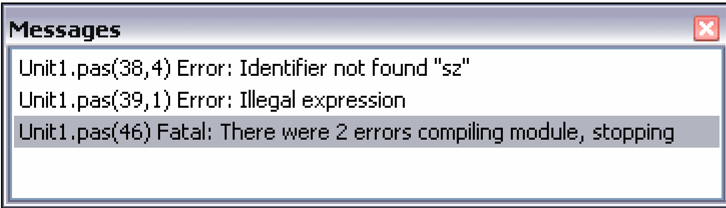


A Lazarus ezen kívül automatikus kódkiegészítéssel is segíti munkánkat. Ez azt jelenti, hogy ha elkezdjük beírni valamely változó, konstans, rekordmező stb. nevét, és megnyomjuk a Ctrl+Space kombinációt, akkor felugró ablakban felajánlja a lehetséges folytatást:

```
Procedure Pelda;
Var szam1:Integer;
    szam2:Real;
    szoveg:String;
Begin
   sz
End var        szam1 : Integer
    var        szam2 : Real
ini var        szoveg : String
   {

end
```

5) Messages:

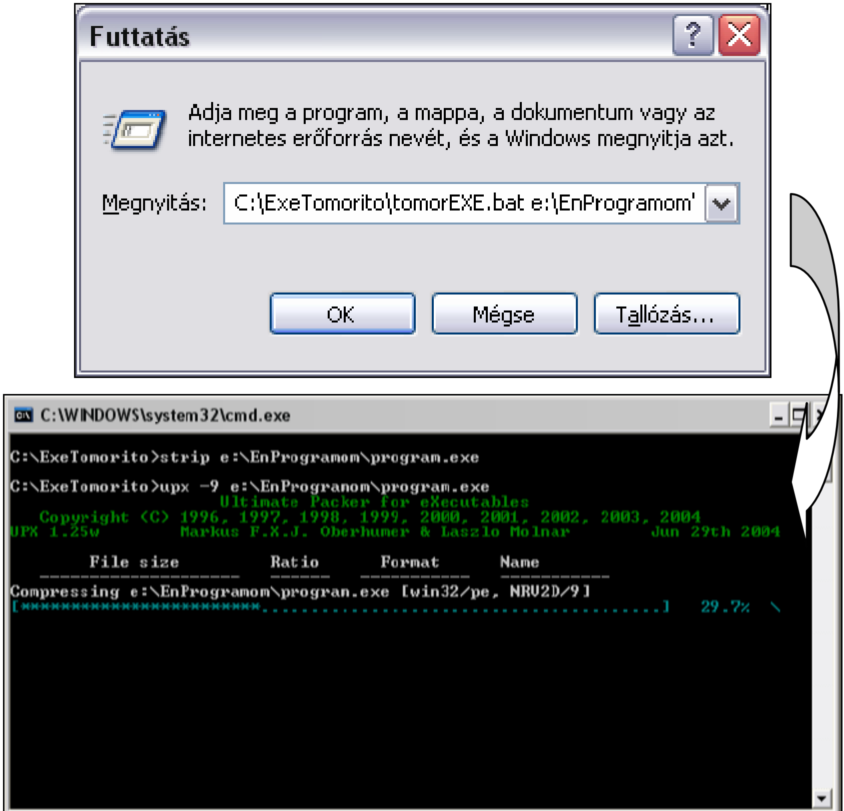
Ebben az ablakban mutatja a Lazarus a kódban található szintaktikai hibákat, esetleg feleslegesnek vélt elemekre hívja fel a figyelmet (pl. ottfelejtett, de már nem használt változók), de a fordítási folyamat állapotáról is itt kapunk tájékoztatást. Például, ha az előző példát kiegészítés nélkül megpróbáljuk lefordítani, akkor ezt láthatjuk:



Azonban, ha sikeresen lefordult programunk, akkor a „Project [projektnév] successfully built." üzenetet kapjuk.

❖ ***Ha túl nagy az exe:***

A Lazarus által létrehozott futtatható fájl mérete egy „üres" program esetében is túllépi a 6MB-os méretet, ami igen nagynak mondható, s ez a komponensek számának növekedésével természetesen tovább emelkedik. Ez főként akkor jelenthet gondot, ha programunkat e-mailben szeretnénk továbbítani. Megoldás lehet a problémára egy kis alkalmazás, mely úgy tömöríti be az exe-fájlt, hogy az továbbra is futtatható marad. A programot innen le tudod tölteni. A zip négy fájlt tartalmaz, melyeket tömörítsünk ki egy mappába. A használatára több mód is van, mi az általunk legegyszerűbbnek véltet mutatjuk be: a Start menüből válasszuk a „Futtatás" menüpontot, majd a megjelenő ablakba írjuk be a tömörítő program és a tömörítendő exe elérési útját. Például, ha a tömörítő fájljait a „C:\ExeTomorito" mappába tettük, és alkalmazásunk futtatható fájlja (exe-je) az „E:\EnProgramom\program.exe", akkor az ablakba a következő szöveget kell beírni: „C:\ExeTomorito\tomorEXE.bat E:\EnProgramom\program.exe"



Az objektum orientált programozás lényege

Ahogy a cím is sugallja, nem célunk teljes, átfogó fejezetet írni a témáról, mert az objektumorientált programozás (OOP) jóval túlmutat e jegyzet céljain. Persze, úgy is tudjuk használni a Lazarus-t, hogy nem ismerjük az objektumokat, de a nyelv megértéséhez, és teljes mértékű kiaknázásához elengedhetetlen bizonyos fokú jártasság a témában. A fejezet előbb bemutatja, mi is az objektum, és az osztály, majd megismertet az OOP filozófiájának 3 alappillérével: betokozás, öröklődés, és többalakúság. Az objektumközpontúságot a vizuális fejlesztőkörnyezet követeli meg, mivel minden programablak egy objektum (nem is akármilyen, de erről majd később), és egy ilyen ablak minden alkotóeleme is egy osztálytípus egy példánya.

❖ *Objektum vagy osztály:*

Osztálynak nevezzük egy adattípus belső adatainak, és műveleteinek (tagfüggvényeinek, ez csak egy elnevezés, lehet eljárás is, s még később lesz róluk szó) együttesét. Pongyolán úgy is fogalmazhatunk, hogy a rekordnak egyfajta kibővítése, s így aztán nem meglepő, hogy használata is hasonló lesz. Itt összefoglaltuk a két típust:

Rekord	Osztály
Type TRekord=**Record** mezo1:Tipus1; mezo2:Tipus2; … **end**;	**Type** TOsztaly=**Class** mezo1:Tipus1; mezo2:Tipus2; … **Function** TagFgv1(…):Tipus1; … **Procedure** TagFgv2(…); … **end**;

Az osztályok eljárásait, és függvényeit meg is kell írni:

```
Function TOsztaly.TagFgv1(…):Tipus1;
Begin
   {utasítások…}
End;

Procedure TOsztaly.TagFgv2(…);
Begin
   {utasítások…}
End;
```

Objektumnak nevezzük az adott osztály egy példányát. Az alábbi táblázatban a rekord illetve az objektum elemeire történő hivatkozást hasonlítjuk össze:

Rekord	Osztály
Type TRekord=**Record** mezo1:Tipus1; mezo2:Tipus2; … **end**; **Var** elem:TRekord; valami:Tipus1; … elem.mező1:=valami;	**Type** TOsztaly=**Class** mezo1:Tipus1; mezo2:Tipus2; … **Function** TagFgv1(…):Tipus1; … **Procedure** TagFgv2(…); … **end**; **Var** elem:TOsztaly; valami:Tipus1; … elem.mező1:=valami; valami:=elem.TagFgv1(…);

Megjegyezzük, hogy ha „benne vagyunk” az adott objektumban, jelen esetben az „elem” nevűben, akkor gyakorlatilag saját magára hivatkozunk, amit megtehetünk a „self” szóval is, vagy el is hagyhatjuk az objektum megnevezését, hisz már „benne vagyunk”. Például:

```
Procedure TOsztaly.TagFgv2(…);
Begin
   …
   elem.mezo1:=…
   {vagy}
   Self.mezo1:=…
   {vagy}
   mezo1:=…
End;
```

- ***Ábrázolás:***

Az objektumtípusú változók valójában mutatók, amik a memóriában elhelyezett objektumokra mutatnak, amiknek használat előtt memóriát kell lefoglalni. Ez lehetővé teszi, hogy ugyanarra az objektumra többen is mutassanak, viszont így bonyolultabb az objektumok létrehozása, és megsemmisítése.

- ***Objektum születése:***

Tehát ha deklarálunk egy új (valamilyen osztálytípusú) változót, egy mutatót hozunk létre. De az objektumnak még le is kell foglalni a memóriát. Ehhez egy különleges tagfüggvényt (ún. konstruktor) fogunk használni. Használhatjuk a Lazarus beépített konstruktorát, vagy írhatunk mi is egyet, ha különleges kezdőértéket szeretnénk. A beépített konstruktor az egyes mezőket a típusuk szerinti alapértékre állítja, s használata egyszerű. Példaként, íme egy kódrészlet:

```
Type TPelda=Class
     end;

Var elem:TPelda;

Procedure Megalkot;
Begin
   elem:=TPelda.Create;
End;
```

A saját konstruktor írása pontosan úgy történik, mint egy belső eljárásé, csak a Procedure szó helyett *Constructor*-t írunk.

- ***Objektumok megsemmisítése:***

Alapszabály, hogy amit egy programban létrehoztunk egyszer, azt meg is kell semmisítenünk (természetesen, ha a programot bezárjuk, annak összes lefoglalt helye felszabadul). Ehhez is egy különleges tagfüggvényt (destruktor) fogunk használni. Ezt a célt szolgáló beépített eljárásai is vannak a Lazarus-nak: Free és Destroy. A Destroy sajnos nem ellenőrzi az objektum mutatóinak „nem-nil”-be mutatását, s ez problémákat okozhat. Ezt elkerülhetjük, ha helyette a Free-t hívjuk meg inkább, mely ezeket ellenőrzi, és ezután meg is hívja a Destroy-t. Sajnos ez még nem elég, mivel csak az objektum helyét szabadítottuk fel, és a mutató még mindig értékes helyre mutat. Ez nem olyan érthetetlen, mivel egy objektum nem tudja (nem is tudhatja), hogy mely változó(k) mutattak rá. Tehát a teljes megsemmisítés így néz ki:

```
Procedure Megsemmisit;
Begin
   elem.Free;
   elem:=nil;
   {vagy}
   FreeAndNil(elem);
End;
```

A saját destruktor írása szintén olyan, mint egy belső eljárásé, csak itt a *Destructor* szót írjuk a Procedure helyére. Ezt azonban nem ajánljuk, mert egy ilyen megírása csak felesleges munkát ad nekünk, és igen bonyolult lehet. Használjuk nyugodtan a beépített destruktort.

❖ ***Tagfüggvények:***

A tagfüggvényeknek két típusa van: az eljárás (procedure) és a függvény (function). Azért nem mondunk négyet, mert a konstruktor és a destruktor is eljárás. A deklarálásuk a meghatározási részben (*interface*), a típusdefiníciónál történik a következő módon: a tagfüggvény típusa, neve, paraméterei, esetleg visszatérési értéke, ha függvényről van szó. A kifejtési részben (*implementation*) a deklarált tagfüggvényeket meg kell írni. A tagfüggvény neve elé oda kell írni az osztály nevét is, ugyanis két osztályban lehet két ugyan olyan nevű tagfüggvény.

```
Type TOsztaly=Class
        …
        Procedure TagFgv1(paraméterek);
        …
        Function TagFgv2(paraméterek):TipusFgv2;
        …
     end;
…
Procedure TOsztaly.TagFgv1(paraméterek);
Begin
   {utasítások…}
End;

Function TOsztaly.TagFgv2(paraméterek):TipusFgv2;
Begin
   {utasítások…}
End;
```

A Lazarus támogatja az eljárások és a függvények úgynevezett túlterhelését is. Ez azt jelenti, hogy két függvénynek vagy eljárásnak ugyanaz a neve, de paramétereiben eltérés van. Ilyenkor a deklaráció mögé oda kell írni az *overload* kulcsszót. Ezt a tulajdonságot tagfüggvényeknél is használhatjuk, például így:

```
Type TOsztaly=Class
        …
        Procedure TagFgv1(be1:Tipus1); overload;
        Procedure TagFgv1(be2:Tipus2); overload;
        Procedure TagFgv1(be3:Tipus3; be4:Tipus4; ki:TipusKi); overload;
        …
     end;
```

❖ ***Betokozás:***

1) A lényeg:

Nem feltétlenül hasznos, ha minden elemét látjuk egy osztálynak. A betokozás lényege, hogy adunk-e hozzáférési jogot az adott mezőnek, tagfüggvénynek, vagy sem. Négy típusát különböztetjük meg a hozzáféréseknek, public (nyilvános), private (privát), protected (védett), published (közzétett). Mi az első kettővel fogunk foglalkozni.

2) Public:

Az ilyen fajta elem az osztály-meghatározáson, és kifejtésen kívül is látszik. Pongyolán megfogalmazva mindenki láthatja.

3) Private:

Csak az osztály-meghatározáshoz tartozó kód látja, és tudja módosítani, lekérdezni. Természetesen egy nyilvános tagfüggvénynek (ami ugyanabba az osztályba tartozik) lehet ez a változója, és ő is módosíthatja, különben nem is lenne sok értelme.

Például: készítünk egy adatbázist, amiben eltároljuk azokat az állatokat, amikkel már találkoztunk. Tartalmazza ez az adatbázis többek között az állat családját, és élőhelyét. Ha minden mezőt szabadon változtathatunk, akkor a felhasználó felvihet az adatbázisba egy olyan halat is, amely a szárazföldön él.

A megoldás: a mezők beolvasásához, és írásához tagfüggvényeket használunk, így ellenőrizni tudjuk, hogy mit ír be a felhasználó, és elrejthetünk előle elemeket is, amikhez nem szeretnénk, hogy hozzáférjen. Ennek szellemében a mezőket általában védetté tesszük, a tagfüggvényeket viszont nyilvánossá. Természetesen ez nem törvényszerű. De vigyáznunk kell, hiszen ha nem vagyunk körültekintőek, és minden mezőhöz külön beolvasó, és megjelenítő tagfüggvényt írunk, a betokozás értelmét veszti.

Használatához a típusdefiníciós részben a kívánt mezők és tagfüggvények elé, be kell írni a megfelelő kulcsszót (*public*, *private*), amelynek hatása addig tart, amíg egy másik ilyen kulcsszót be nem vezetünk. A fentebb említett példa így nézhetne ki:

```
Type TAllat=Class
     private
        csalad:String;
        elohely:String;
     public
        Procedure Beolvas(csalad,elohely:String);
        Procedure Kiir;
     end;
```

De miért is jó ez? Ha már megírtunk egy osztályt, írtunk már hozzá programot is, ami megfelelően működik, akkor ezután jön a hatékonyság. Ha megváltoztatjuk a belső ábrázolást az osztályban, és a programunk, ami ezt felhasználja, ismerte a mezőneveket, és nem tagfüggvényekkel, hanem direktben fért hozzájuk, az egész osztályt, és a programot is át kell írnunk. Viszont, ha a program nem látta a mezőket, csak tagfüggvényeken keresztül használta őket, csak e tagfüggvényeket kell átírnunk, magát a programot nem. Ha az előbbi példát átírnánk, és nem szöveget, hanem felsorolás típust használnánk, úgy csak a mezők típusát és az azokat használó tagfüggvényeket kell módosítani.

❖ ***Öröklődés:***

Lehetséges, hogy a már kész osztályunkat kissé módosítva fel szeretnénk használni. Erre a legegyszerűbbnek tűnő megoldás a kódrészlet átmásolása tűnik, és a másolatot változtatjuk. Viszont ez sok problémát felvet, amiből csak a helybeli hatékonyság a legkisebb, mivel így az esetleges hibákat megkétszerezzük, és két helyen kell ugyanazt kijavítani. Erre nyújt megoldást az öröklődés. Elég leírni, hogy kitől származik (a módosított típus „class” kulcsszava után zárójelben odaírjuk, hogy kitől származik), és azt, hogy milyen változások vannak benne.

```
Type TModositott=Class(TEredeti)
        …//a módosítások
     end;
```

Ha megfigyeljük, a Lazarus is így tesz minden form esetében. Van egy „ős” form meghatározás, és ezt egészíti ki mezőkkel (általunk beírt elemek vagy komponensek, melyeket mi helyezünk a formra), és tagfüggvényekkel (események, amik a komponensek műveleteihez kapcsolódnak, és olyan tagfüggvények, amiket mi írunk). A formokról részletesebben egy későbbi fejezetben olvashatsz.

Az öröklődés kapcsán szóba jön egy érdekes típusátadás, ami első pillanatra szokatlannak tűnik, mivel a Pascal-alapú nyelvekben az értékadás erősen típusfüggő. Még, ha két konstrukció, például rekord ugyanolyan mezőkből áll, egy ilyen típusú változót nem adhatunk értékül a másik típusú változónak.

```
Type TValami=Record
        x,y:Integer;
     end;
     TBarmi=Record
        x,y:Integer;
     end;
…
Procedure Pelda;
Var v:TValami;
    b:TBarmi;
Begin
   valami.x:=2;
   valami.y:=3;
   barmi:=valami; //erre a Lazarus hibát jelez
End;
```

Viszont, ha definiálunk egy osztályt (TEredeti), és származtatunk belőle egy másik osztályt (TModositott), akkor egy TEredeti típusú változónak értékül adhatunk egy TModositott típusú változót:

```
Type TEredeti=Class
        …
     end;
     TModositott=Class(TEredeti)
        …
     end;
…
Procedure Pelda;

Var eredeti:TEredeti;
    modositott:TModositott;
Begin
   eredeti:=modositott;
End;
```

❖ ***Többalakúság:***

Lényege, hogy egy származtatott osztályban lévő tagfüggvény felülbírálhat egy ugyanolyan nevű, azonos paraméterű, az „ősosztályban” lévő tagfüggvényt. Azaz, ha van egy osztályunk és származtatunk belőle egy másik osztályt, a második osztály mindegyike tartalmaz egy bizonyos tagfüggvényt. Az ősosztályban ezt a függvényt virtuálisnak nevezzük, a meghatározása mögé egy *virtual* kulcsszót írunk. A származtatott osztályban ezt felbírálhatja a másik függvény, ami mögé az *override* kulcsszót írjuk:

```
Type TEredeti=Class
        …
        Function Fgv:TipusFgv; virtual;
     end;

     TModositott=Class(TEredeti)
        …
        Function Fgv:TipusFgv; override;
     end;
```

Azt, hogy melyik függvényt kell meghívni, az határozza meg, hogy milyen változóval hívjuk meg. Tehát, a „valami.Fgv” függvényhívás attól függ, hogy az a „valami” milyen objektumot tartalmaz. Előbb láttuk, hogy előfordulhat, hogy „valami” TEredeti típusú mégis TModositott objektumot tartalmaz.

❖ ***Zárszó:***

Ebben a fejezetben, nagylépésekben átnéztük az OOP alapelveit, és a megvalósításának szerkezetét, ami nem azt jelenti, hogy teljes lenne a kép, de ebben a pár oldalban nem is lehet. Azonban reméljük, felkeltettük az érdeklődésedet, és néhány problémánál eszedbe fog jutni, hogy objektumok megvalósítva mennyivel logikusabb, átláthatóbb lenne a megoldás, mivel az objektumok használata lecsökkenti a globális változók számát, így átláthatóbbá válik a programkód.

A látható komponensek tulajdonságai (Properties)

A jegyzet e részében, ahogy a cím is mutatja, a látható komponensekkel fogunk foglalkozni. Mivel ezeknek sok közös tulajdonságuk van, így ezeket nem részletezzük ki minden egyes komponensnél, hanem itt foglaljuk össze őket. Persze az itt bemutatott tulajdonságok nem találhatók meg mindegyiknél, pusztán csak gyakori előfordulásuk miatt kerültek különválasztásra. Természetesen a sajátos és számunkra fontos tulajdonságra mindegyiknél ki fogunk térni.

❖ ***Align (egyenesbe hozás):***

Egy komponens tartalmazó objektumában való helyét határozhatjuk meg ezzel a tulajdonsággal oly módon, hogy melyik oldalhoz igazodjon.

Érték	Jelentése
alBottom	Alulra teszi a komponenst, és a szülő (tartalmazó) további méretezésénél is alul marad.
alTop	Felülre teszi a komponenst, és a szülő (tartalmazó) további méretezésénél is fent marad.
alClient	A komponens kitölti a rendelkezésére álló helyet. Ha egy másik komponens már lefoglalt helyet, akkor értelemszerűen csökken a fennmaradó hely.
alCustom	Ha használtunk már valamilyen align-t, és változtattunk a Left vagy a Top (lásd később) tulajdonságokon, akkor alCustom-ra váltva ezek fognak érvényesülni.
alLeft	Balra teszi a komponenst, és a szülő (tartalmazó) további méretezésénél is a baloldalon marad.
alRight	Jobbra teszi a komponenst, és a szülő (tartalmazó) további méretezésénél is a jobboldalon marad.
alNone	Nem határozunk meg speciális igazítást.

❖ ***Anchors (horgonyok):***

Az Align-nal viszonylag primitíven ugyan, de meg tudtuk határozni, hogy a tartalmazó objektum méretének változásával változzanak-e a komponensek helyzetei, és ha igen hogyan. Ennél egy fokkal finomabb, ha horgonyokat helyezünk el. Ez annyit tesz, hogy minden komponensnek meg tudjuk mondani, hogy melyik oldalhoz igazodjon. Igazodáson azt értjük, hogy ha a tartalmazó objektumot méretezzük egy irányba, akkor a komponens is átméreteződik ugyanabba az irányba, ugyanakkora mértékben. Alap érték, hogy a Top (felső) és Left (bal) horgonyok vannak Igaz értékűre állítva. Természetesen bármilyen kombináció elképzelhető. Egy Memo-nál például, hasznos lehet, ha mind a 4 horgonyt igazra állítjuk, mert ez esetben a form növekedésével minden irányban növekszik a Memo.

❖ ***AutoSize (automatikus méretezés):***

Igaz és Hamis értékek között válthatunk. Igaz-ra állítva a komponens méretét automatikusan igazítja a tartalmához (gomb esetében a felirathoz, képkezelő esetében a betöltött képhez, stb.).

Megjegyzés: sajnos tapasztaltuk, hogy nem minden esetben működik.

❖ ***Caption (felirat) és Text (szöveg):***

Az adott komponens tartalmaként fogalmazható meg. Ha a komponens tartalmaz beviteli részt, akkor általában van Text mezője, ez tartalmazza a felhasználó által a beviteli mezőbe írt szöveget. Ha a komponensen csak megjelenítésre szolgáló szöveg van, akkor a Caption tulajdonsággal állíthatjuk annak tartalmát.

❖ ***Color (szín):***

Általában a komponensek háttérszínét változtathatjuk ezzel a tulajdonsággal. Szerkesztés közben egy legördülő listából választhatunk színt, míg kódból történő módosításkor egy másik lehetőségünk is van: BGR formátumú megadásra. Itt nem tévedés a fordított írásmód (a megszokott ugye az RGB), ugyanis Lazarus-ban kék-zöld-piros a sorrend, és elé kell tenni a $00 kódot ($00BGR – ennek okát itt nem tárgyaljuk). Tehát mondjuk a három alapszín: $00FF0000 (kék), $0000FF00 (zöld), $000000FF (piros).

❖ ***Constraints (korlátok):***

Ha az Align-t és az Anchors-t rendszeresen használjuk, érdemes ezzel az összetett tulajdonsággal is megismerkednünk! Előfordulhat, hogy a tartalmazó objektum nagyításánál a komponensek olyan méretet érnek el, ami már nem praktikus. Ezt elkerülhetjük, ha itt beállíthatjuk a komponens maximális/minimális magasságát, szélességét. A megadott értékek pixelben értendők.

❖ ***Cursor (egérmutató):***

Beállíthatjuk, hogy milyen legyen az egér mutatója, ha az adott komponens fölé visszük. Az egyes jelentéseket a Windows-ból kölcsönöztük a könnyebb érthetőség kedvéért.

Érték	Jelentése
crAppStart	Munka a háttérben
crArrow	Normál kijelölés
crCross	Pontos kijelölés
crDefault	Alapállapot
crDrag	Húzás (nem működik)
crHandPoint	Hivatkozás kijelölés
crHelp	Súgókijelölés
crHourGlass	Foglalt
crHSplit	Vízszintes elválasztás
crIBeam	Szövegkijelölés
crMultiDrag	Több elemes húzás (nem működik)
crNo	Nem érhető el
crNoDrop	Nem húzható (nem működik)
crSizeAll	Áthelyezés
crSizeNESW	Átlós átméretezés (ÉK-DNY)
crSizeNS	Függőleges átméretezés
crSizeNWSE	Átlós átméretezés (ÉNY-DK)
srSizeWE	Vízszintess átméretezés
crSQLWait	SQL-foglalt
crUpArrow	Alternatív kiválasztás
crVSplit	Függőleges elválasztás (nem működik)

❖ ***Enabled (engedélyezettség):***

Értékét Igaz vagy Hamis közt választhatjuk. Ha Igaz-ra állítottuk a komponens nem lesz elérhető (például, egy gombra nem lehet rákattintani).

❖ ***Font (betűkészlet):***

A komponensen megjelenő szöveget formázhatjuk a segítségével. Ezt könnyedén megtehetjük a Windows-ban megszokott párbeszédablak segítségével.

❖ ***Height (magasság) és Width (szélesség):***

Ezzel a két tulajdonsággal lehet méretezni a komponensünket, s értékeik pixelben értendők.

- ***Top és Left (bal felső pont-koordináták):***

Értékük szintén pixelben értendő, segítségével az adott komponens helyét határozhatjuk meg az őt tartalmazó komponensben. A Top a tetejétől, a Left pedig a bal széltől való távolság.

- ***Hint (útmutatás) és ShowHint (útmutató engedélyezése):***

A komponenshez segítő információt rendelhetünk (Hint), mely akkor jelenik meg, ha az egeret fölé visszük, és a ShowHint tulajdonságot Igaz-ra állítjuk.

- ***Name (komponensnév):***

Mivel minden komponens egy objektum, így van neki azonosítója, amivel hivatkozni tudunk rá. Ez a Name tulajdonság. Értéke a Pascalban is megszokott azonosító-formátumú lehet.

- ***Parent-tulajdonságok (szülőtulajdonságok engedélyezése):***

Több ilyen tulajdonsággal találkozhatunk. Közös bennük, hogy logikai értékűek. Ha Igaz-ra állítjuk a komponensünk ilyen tulajdonságait, akkor mintegy örökli őket a szülőtől. Például, ha egy címkénél engedélyeztük a ParentFont-ot, akkor az őt tartalmazó komponens betűtípusának formai beállításai lesznek rá is érvényesek, hacsak a komponens saját tulajdonságát nem módosítjuk külön. Ekkor viszont, értelemszerűen a megfelelő Parent-tulajdonság Hamis értékűre vált!

- ***TabOrder (Tab-sorrendbeli hely):***

Amikor a „Tab” billentyűt megnyomjuk, a kijelölés átmegy egy másik komponensre. A TabOrder ebben a hierarchiában elfoglalt helyet tartalmazza. A 0. lesz az alapértelmezésben (vagyis a program indulásakor) kijelölt komponens. Az utolsónak kijelölt elemnél, ha megnyomjuk a „Tab”-ot, a 0. elemre ugrik vissza. A Lazarus a sorrendet a formra kerülés sorrendjében állítja be, amit mi magunk meg is változtathatunk, ha átírjuk ennek a mezőnek a tartalmát. Ilyenkor a Lazarus a többi komponens értékét automatikusan átállítja.

- ***TabStop (Tab-kirekesztés):***

Vannak olyan komponensek, melyeket nem szeretnénk elérhetővé tenni a „Tab” használatával. Ehhez nem kell mást tennünk, mint Hamis-ra állítani ezt a tulajdonságot.

- ***Tag (cédula):***

Ez egy Longint típusú változó, tehát egy 4 bájtos szám. Minden komponensnek van ilyen mezője, de előre meghatározott célja nincs. Tetszésünk szerint felhasználhatjuk bármire, amit jónak látunk.

- ***Visible (láthatóság):***

A komponens láthatóságát állíthatjuk be vele.

Az leggyakoribb események (Events)

Egy fontos részhez érkezünk, az eseményekhez. Szinte minden komponenshez tartoznak ilyen eljárások, melyek előre meghatározott programbeli történésekkor hajtódnak végre. Ilyenek használatával érhető el például egy átlagos böngészőben, vagy szövegszerkesztőben az, hogy gombok megnyomásának hatására a program valamilyen cselekvést hajtson végre (pl. beállított kezdőlapra visszatérés egy böngésző esetén, vagy a Word igazításai). Ebben a fejezetben olyan eseményekről esik szó, melyek sok komponensnél megtalálhatók, és használatuk gyakori.

Az események közös jellemzője, hogy egy „Sender" TObject típusú paramétert kapnak, mellyel lekérdezhető, hogy mely komponens hívta meg az adott eljárást. Ennek oka, hogy több komponens eseménye lehet ugyanaz az eljárás. Például elképzelhető, hogy egy gomb lenyomásakor és egy menüelem kiválasztásnál ugyanazt a műveletsort hajtja végre a program.

Minden egyes esemény létrehozható oly módon, hogy az object inspector-ban a fastruktúrából kiválasztjuk azt a komponenst, melyhez eseményt szeretnénk hozzárendelni, az Events fülre kattintunk, majd a kiválasztott esemény sorának jobb oldalán a „…" feliratú gombra kattintunk. A Lazarus automatikusan beilleszti a form típus-meghatározásába az eljárás fejlécét, és előkészíti a kifejtést (az implementation részbe kódrészletet illeszt a fejléccel és egy „begin-end" párosítással).

❖ ***OnClick (kattintáskor):***

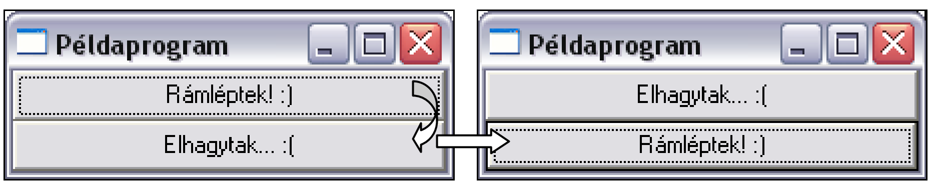
Az egyik leggyakrabban használt esemény, melynek utasításai akkor hajtódnak végre, ha a felhasználó az egér bal gombjával kattint az adott komponensre. Például, ha azt szeretnénk, hogy egy gomb megnyomására kilépjen a programunk:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Application.Terminate;
   //az Application-nel magára az alkalmazásra hivatkozunk, a Terminate
   //eljárás leállítja a futását.
end;
```

❖ ***OnEnter (kiválasztáskor) és OnExit (elváltáskor):***

Az OnEnter esemény akkor hajtódik végre, ha a fókusz (vagyis a kiválasztás) rákerül a komponensre („rálépünk"), míg az OnExit akkor, ha elváltunk róla („elhagyjuk"). Egy egyszerű példa, hogy két gomb feliratát változtatjuk attól függően, éppen melyik a kiválasztott. Kódtakarékosság céljából csak két eljárást írtunk meg, mindkettőt az első gombhoz (Button1). A másik gomb (Button2) esetében az object inspector-ban csak kiválasztottuk a megfelelő eljárásokat. Az eljárásokban a már említett *Sender* paraméter segít eldönteni azt, hogy ki (tehát melyik gomb) hívta meg az adott eljárást.

```
procedure TForm1.Button1Enter(Sender: TObject);
begin
   If Sender=Button1 then Button1.Caption:='Rámléptek! :)'
                     else Button2.Caption:='Rámléptek! :)';
end;

procedure TForm1.Button1Exit(Sender: TObject);
begin
   If Sender=Button1 then Button1.Caption:= 'Elhagytak... :('
                     else Button2.Caption:= 'Elhagytak... :(';
end;
```



❖ ***OnKeyPress (billentyű lenyomásakor):***

Az ilyen eljárások mindig kapnak egy plusz „key” nevű paramétert, ezért szintaktikájuk a következőképp néz ki: *OnKeyPress(Sender: TObject; var key: Char)*. Ez az esemény ASCII kóddal rendelkező billentyűk, és billentyűk kombinációk használatakor fut le. A lenyomott billentyű a „key” paraméterrel kérdezhető le, és módosítható is akár. Példaként adunk egy megoldási lehetőséget arra, hogy egy editbox-ba ne írhasson be szöveget a felhasználó:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
   If not (Key in ['0'..'9',#8]) then key:=#0;
end;
//a #0 az üres karakter, míg a #8 a backspace kódja. Utóbbi azért
//szükséges, hogy a lehessen törölni
```

❖ ***OnMouseDown (egérgomb lenyomásakor):***

Ezek az eljárások négy többletparamétert kapnak, és szintaktikájuk a következőképp néz ki: *OnMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer)*. Akkor fut le az esemény, ha a komponensen van az egér mutatója, és a felhasználó kattint az egér bal, jobb, vagy középső gombjával. Az esemény akkor is lefut, ha az egérgombbal kattintást a „Shift”, „Alt” vagy „Ctrl” billentyűkkel kombináljuk. A Button paraméterrel kérdezhető, hogy melyik volt a használt egérgomb:

Érték	Jelentése
mbLeft	Bal egérgomb.
mbRight	Jobb egérgomb.
mbMiddle	Középső egérgomb.

Az X és Y az egér mutatójának koordinátái a komponensen.
A Shift paraméter mutatja az egérgombok, az „Alt”, „Shift” vagy „Ctrl” billentyűk állapotát:

Érték	Jelentése
ssLeft	Bal egérgomb nyomva van tartva.
ssRight	Jobb egérgomb nyomva van tartva.
ssMiddle	Középső egérgomb nyomva van tartva.
ssDouble	Az egérrel duplán kattintottak.
ssAlt	Az Alt billentyű lenyomva.
ssShift	A Shift billentyű lenyomva.
ssCtrl	A Ctrl billentyű lenyomva.

A Shift paraméter egy halmaztípus, melybe belekerülhetnek ezek az elemek. Lekérdezni nyilván úgy tudjuk, hogy benne van-e valamelyik elem a halmazban. Például, egy címke dupla kattintásra írja ki, hogy „dupla-katt”:

```
procedure TForm1.Label1MouseDown(Sender:TOBject; Button:TMouseButton;
                                 Shift: ShiftState; X, Y: Integer);
begin
   If ssDouble in Shift then Label1.Caption:='dupla-katt';
end;
```

❖ ***OnMouseUp (egérgomb felengedésekor):***

Paramétereiben megegyezik az előző eseménnyel, csakhogy ez az egérgomb, vagy a kombináció felengedésekor fut le.

❖ ***OnMouseMove (egér mozgatásakor):***

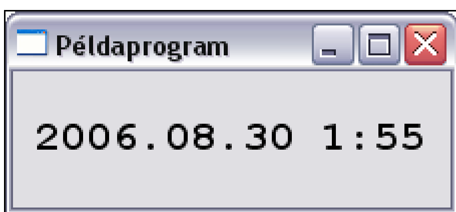
Az ilyen eljárások szintaktikája így néz ki: *OnMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer)*. Az esemény akkor hajtódik végre, ha a komponens fölött mozgatjuk az egeret. Például, kiírathatjuk egy címkére, hogy hol van épp az egér a formon:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
var seged:String;
begin
   seged:='Az egér épp itt van: X: '+IntToStr(X)+', Y: '+IntToStr(Y);
   Label1.Caption:=s;
end;
```

❖ ***OnCreate (létrehozáskor):***

A komponens létrejötte után fut le közvetlenül. Például, a program indulásakor kezdőértékeket adhatunk, vagy akár elmenthetjük a program indulásának idejét is, ahogyan ezt példánkban is tettük (emlékezzünk vissza a konvertáló függvényekre!):

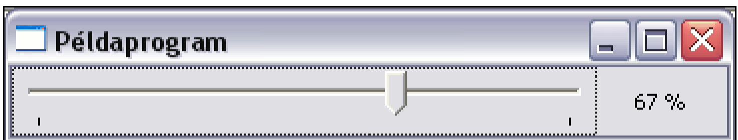
```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Label1.Caption:=DateTimeToStr(Now);
   //a Now függvény az aktuális rendszerdátumot és időt kérdezi le
   //visszatérési értéke TDateTime típusú, amit a DateTimeToStr
   //függvénye segítségével konvertálunk az elvárt String típusúvá.
end;
```



❖ ***OnChange (változáskor):***

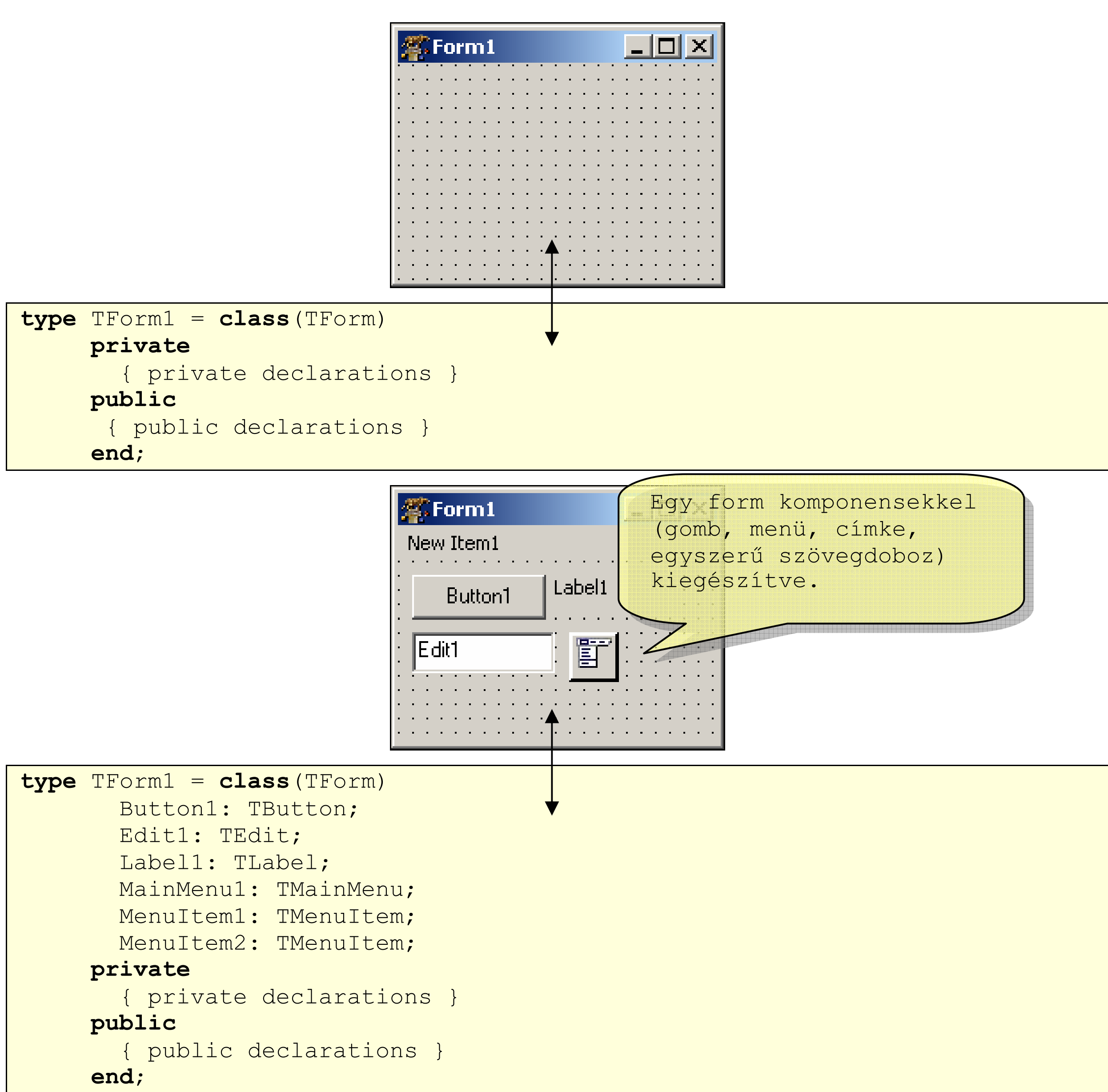
Az esemény akkor hajtódik végre, ha a komponensen a saját feladatát meghatározó tulajdonságában történik változás. Például, egy szövegbevitelre alkalmas komponens esetében, amikor a tartalmát módosítjuk, csúszka esetében, ha jelölőt odébb tesszük.

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
   Label1.Caption:=IntToStr(TrackBar1.Position)+' %';
end;
```



A Formok

Az egy alkalmazáshoz tartozó ablakoknak a program tervezése közben az ún. formok felelnek meg. Ezekre helyezhetjük el a komponenseket úgy, hogy rákattintunk a kiválasztottra, majd a formra kattintunk. Itt alakíthatjuk ki programunk arculatát, divatos szóhasználattal élve design-ját. A form tulajdonképpen egy olyan ősformon alapuló objektum, melyet saját komponensekkel, függvényekkel és változókkal egészíthetünk ki. A Lazarus automatikusan frissíti a formunk osztály-meghatározását, amikor komponenseket helyezünk el a rajta, vagy ezekhez különböző eseményeket rendelünk. A komponensek nevei (azonosítói) automatikusan kerülnek kiosztásra a típustól és az egyforma típusú elemek számától függően. Például, ha két „TValami” típusú komponenst tennénk a formra, akkor a neveik „Valami1” és „Valami2” lenne. A neveket természetesen a Properties fejezetben ismertetett módon meg tudjuk változtatni. Az alábbi képeken és forrás-részleteken látható, hogy milyen egy üres form, egy olyan, amin komponenseket helyeztünk el, és ez miként néz ki lefordítva:




```
type TForm1 = class(TForm)
      private
        { private declarations }
      public
       { public declarations }
      end;
```

```
type TForm1 = class(TForm)
        Button1: TButton;
        Edit1: TEdit;
        Label1: TLabel;
        MainMenu1: TMainMenu;
        MenuItem1: TMenuItem;
        MenuItem2: TMenuItem;
      private
        { private declarations }
      public
        { public declarations }
      end;
```

❖ ***A formok – Tulajdonságok (Properties):***

1) AutoScroll (automatikus gördítők):

Ha ezt a tulajdonságot Igaz-ra állítjuk, akkor a formról esetleg (pl. átméretezés miatt) kilógó részeket automatikusan megjelenő gördítőkkel érhetjük el.

2) BorderIcons (rendszergombok):

Alap állapotban a formok fejlécének jobb oldalán három gomb található, melyek rendre a „Kis méret", „Teljes méret" és a „Bezárás". Itt beállíthatjuk, hogy melyiket szeretnénk engedélyezni, de letilthatjuk akár mindhárom rendszergombot is.

Mező	Jelentése
biSystemMenu	Hamis-ra állítva mindhárom gomb eltűnik.
biMinimize	Hamis-ra állítva a „Kis méret" gombot tilthatjuk le.
biMaximize	Hamis-ra állítva a „Teljes méret" gombot tilthatjuk le.
biHelp	Ha a form kerettípusa (lásd később) bsDialog, vagy a biMinimze és a biMaximize is Hamis, akkor ezt a tulajdonságot Igaz-ra állítva egy „Súgó" gomb jelenik meg. Megjegyzés: ez nem működik.

3) BorderStyle (keret típusa):

Segítségével formunk szegélyének típusát tudjuk beállítani. Néhány módosítás eredményeképp a formot a felhasználó nem tudja átméretezni, mint egy hagyományos ablakot. A lehetséges értékeket táblázatba foglaltuk:

Érték	Jelentése
bsDialog	A form nem méretezhető, a fejléc jobb oldali gombjaiból csak a „Bezárás" marad.
bsNone	A form nem méretezhető, kerete nincs, s így a felhasználó áthelyezni sem tudja.
bsSingle	A form nem méretezhető át, a rendszergombok alapállapotban vannak.
bsSizeable	Az alapbeállítás. A formot szabadon méretezheti a felhasználó, a rendszergombok alapállapotban vannak.
bsSizeToolWindow	Olyan, mint a bsSizeable, de csak bezáró gombbal rendelkezik, és fejléce vékonyított. (Lásd: Object Inspector)
bsToolWindow	Olyan, mint a bsSingle, de csak bezáró gombbal rendelkezik, és fejléce vékonyított.

4) Icon (ikon):

Itt állíthatjuk a form bal felső sarkában, a címsor melletti kis képet.
Megjegyzés: nem működik.

5) Menu (menü):

Lazarus-ban lehetőségünk nyílik menüket szerkeszteni, erről később, a Standard komponenseknél olvashatsz bővebben. Ha több menü van a formon, akkor itt választható ki az aktuális.

6) PopupMenu (felugró menü):

Később szintén lesz szó úgynevezett felugró menükről. Ezek az egér jobb gombjának kattintásával hívható elő. Ha több ilyen menü van a formon, akkor itt kiválasztható az aktuális.

7) Position (formpozíció):

A form első megjelenésekor a méretét és helyzetét befolyásoló tulajdonság. Számunkra most négy érdekes értéke van:

Érték	Jelentése
poDesigned	A form mindig ott jelenik meg, ahova a program szerkesztése közben helyeztük. Alapértelmezettként ez van beállítva.
poDefault	A form elhelyezkedését az operációs rendszer határozza meg. Minden egyes programindításkor a form egy kicsivel jobbra és lefele jelenik meg (így például elkerülhető, hogy egymáson jelenjenek meg ablakok).
poDesktopCenter	Ezzel elérhető, hogy mindig a képernyő közepén jelenjen meg a form.
poMainFormCenter	A form mindig a főformhoz képest középen jelenik meg. A főformot a Project menü, Project Options pontjában tudjuk beállítani. A „Forms” fülre kattintva a formok listájának legelső eleme lesz a főform.

Megjegyezés: poDesigned pozícionáláskor érdemes figyelni a form helyére. Ha több ablakkal dolgozunk, akkor valószínű, hogy félrehúzunk néhányat, netán minimalizálunk a könnyebb átláthatóságért. Viszont ez esetben a fordítás után is ott marad, és megijedhetünk, hogy valami hiba történt.

❖ ***A formok – Használatuk:***

Egy alkalmazás több formot is használhat sőt, inkább ez az általános. Újabb formokat egyszerűen tudunk létrehozni: a Lazarus File-menüjében találunk egy „New Form” menüpontot. Erre kattintva már is kész az új ablak, és egy új unit is, a Lazarus ugyanis minden formhoz külön unitot készít. Új formunkon ismét helyezhetünk el komponenseket, írhatunk függvényeket, eljárásokat stb. A komponensek neveinek automatikus kiosztása – természetesen – elölről kezdődik, tehát külön formokon létezhet ugyanolyan azonosítóval rendelkező elem, függvény stb. Már meglévő formjaink között a Lazarus Windows-menüjében vagy egérrel tudunk váltani.

Megjegyzés: egy programhibára (bug) hívnánk fel a figyelmet. Tapasztaltuk, hogy több form használatakor, ha formot váltunk előfordulhat, hogy nem tudunk újabb komponenseket elhelyezni a kiválasztott ablakon. Ekkor az F12 forróbillentyűvel váltsunk a forrásra, majd ismét megnyomva vissza a formra.

Ha létrehoztunk egy újabb formot, akkor azt nyilván használni is szeretnénk. Az újonnan létrehozott formok Visible tulajdonsága azonban alapértelmezésben Hamis-ra van állítva, így új programablakunk nem lesz látható, ha lefordítjuk a programot. Ennek átállításával láthatóvá tehetjük az új formokat, de így a program indulásakor minden így beállított ablak látható lesz. Ha azt szeretnénk, hogy használat közben valamilyen esemény hatására, felhasználói beavatkozásra (pl. gombkattintás) aktiválódjon egy programablak, akkor ezt kódból tehetjük meg.

Ha egy ablakból a program egy másik ablakát szeretnénk láthatóvá tenni, ezt elsősorban a Show belső eljárással szoktuk megtenni (pl. Form1.Show). Említettük azonban, hogy minden form leírása egy külön unitban van. Ezért minden „hívó” form unitlistájába fel kell vennünk a „mehívandó” formok unitjait. Például, ha létrehoztunk egy Form2-t (melynek unitja Unit2), melyet a már meglévő Form1-ről (melynek unitja Unit1) szeretnénk elérni, akkor a Form1 unitlistájába be kell írni az új form unitját, különben a Unit1 eljárásai, függvényei nem látják a Form2-t, így nem tudjuk meghívni a Show-t:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
  Buttons, Unit2;

…

implementation

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;
```

Ha kihagyjuk a Unit2-t a unitlistából, akkor a fordítás közben a Lazarus érzékeli, hogy ezt nem tettük meg, és ismeretlen objektumként hivatkozik a Form2-re, vagyis a Messages ablakba az „Identifier not found „Form2"" hibaüzenet kerül.

Elképzelhető, hogy két form kölcsönösen el akarja érni egymást. Tehát egy formról meg szeretnénk nyitni egy másikat és fordítva, vagy csak egyszerűen egymás változóit akarják használni. Ehhez szükséges, hogy mindkét form unitjának interface résznél található unitlistába felvegyük a megfelelő unitokat. Előző példánknál maradva a Unit1-be a Unit2-t, és fordítva. Ez azonban kereszthivatkozást eredményez: a Messages ablakban a „Circular unit reference between Unit2 and Unit1" hibaüzenettel figyelmeztet a Lazarus, ha fordítani próbálunk. Ezt el tudjuk kerülni azzal, hogy nem az interface unitlistájába vesszük fel a unitokat, hanem az implementation-nél:

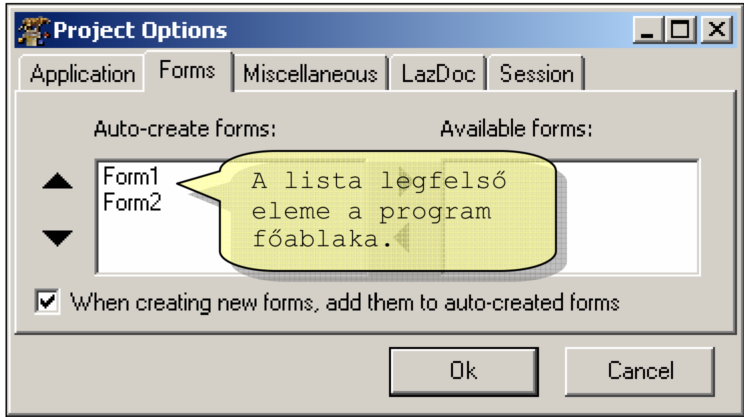
```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs;
```

```
type

  { TForm1 }

  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

…

implementation

uses Unit2;
```

```
unit Unit2;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs;

type

  { TForm2 }

  TForm2 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

…

implementation

uses Unit1;
```

A fenti probléma és megoldása nyilvánvalóan nem szorosan a formokhoz kötődik, hiszen itt a külön unitok okozzák a bajt, nem pedig maguk a formok. Tehát a megoldás bármilyen más unitok kereszthivatkozásánál is segít. Azért említettük itt, mert gyakran fordulhat elő ilyen a formoknál.

Fontos megjegyeznünk, hogy a formok alapértelmezésben egyenrangúak, egy kivételtől eltekintve, ezt hívjuk főformnak (MainForm). A főformot a Project menü, Project Options pontjában tudjuk beállítani. A „Forms” fülre kattintva a formok listájának legelső eleme lesz a főform.



A főform határozza meg, hogy mely form lesz alkalmazásunk főablaka. Ha a főform bezárul, akkor maga a program is kilép, ez azonban nem igaz a többi formra. Például, ha két formunk van (Form1 és Form2, főform legyen a Form1), akkor a Form1-et bezárva kilép a program, a Form2-t bezárva azonban nem. Ilyennel igen gyakran találkozhatunk, hiszen sok program használhat például segédablakokat, s ezek bezárásával nemigen szeretnénk, hogy az alkalmazásból is kilépjünk. Gondolhatunk itt a Total Commander másolóablakára, vagy akár a manapság divatos csevegő-programok egyikre. De ne menjünk messze, maga a Lazarus is több formot használ, és például, a forrás-szerkesztő mégsem tudja bezárni a teljes alkalmazást.

A Standard-paletta (alapkomponensek)

Ez a komponens-paletta azért kapta a Standard nevet, mert az itt található komponenseket szokás a leggyakrabban alkalmazni, használatuk egyszerű, és mégis sok felhasználási lehetőségük lehet. A paletta képe:

Programjaink nagy részét pusztán ezek használatával is sokrétűvé tehetjük, azonban szépítésekre csak korlátozottan nyílik lehetőségünk. Ezek a komponensek kinézetre is egyszerűek, főként szöveg bevitelére és kiírására alkalmas elemek, hagyományos vezérlők (pl. menü, gomb, rádiógomb). A palettáról igyekeztünk olyan komponenseket kiválogatni, melyek valóban használatra kerülnek (kerülhetnek) a gyakorlatok során.

A fejezetből kiderül, hogyan készíthetünk szinte minden programban megtalálható menüsort, szó lesz egy, illetve több soros szövegek használatáról, lenyíló menükről, és komponenseink csoportosításáról is.

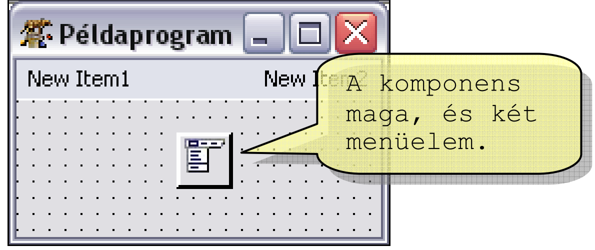
Összesen tizennégy komponenst veszünk át, melyeket itt tematikusan összefoglaltunk, és hivatkozásokat helyeztünk rájuk, hogy könnyebben elérhesd őket:

- Főmenü
- Felugró menü
- Gomb
- Címke
- Egyszerű szövegmező
- Jegyzettömb
- Jelölőnégyzet
- Választógomb
- Listadoboz
- Kombinált lista
- Csoportmező
- Választógomb csoportmező
- Jelölőnégyzet csoportmező
- Eseménylista

❖ *A főmenü (TMainMenu) – Bevezető:*

Segítségével egy hagyományos menüsor szerkeszthető a program fejléce alá. A menü elemei újabb menüpontok listájához vezethetnek, vagy OnClick események eljárásait rendelhetjük hozzájuk. Sok program használ ilyet a logikailag egybe tartozó programműveletek csoportosítására, ergonómiai szempontból előnyös és az alkalmazás könnyebb vezérlését is segíti. Ez egy úgynevezett nem látható komponens, a formon való elhelyezésére azért van szükség, hogy a menüelemeket egybefoglalja. A formon tetszőlegesen elhelyezhetjük.

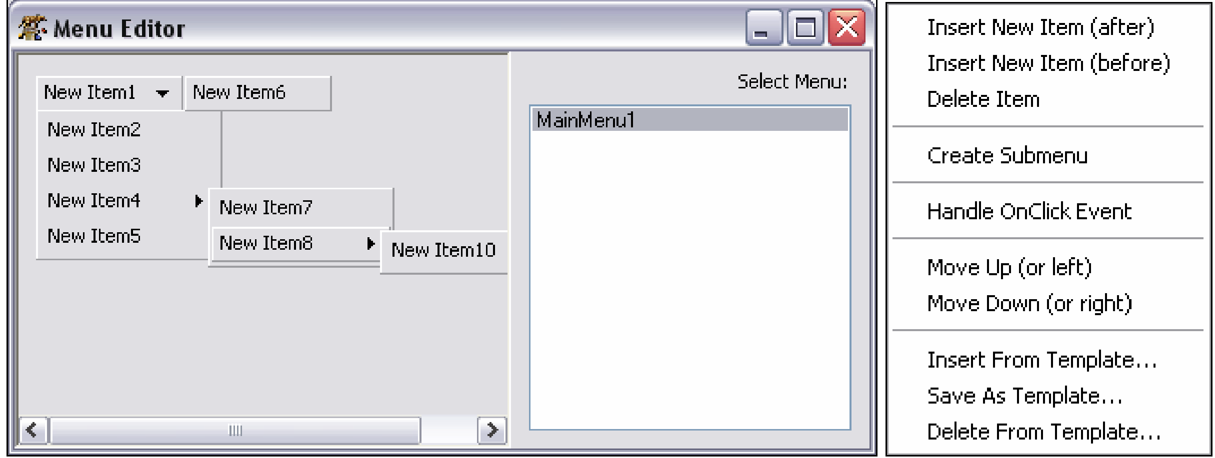
A komponens elhelyezve egy üres formon:



❖ ***A főmenü (TMainMenu) – Tulajdonságok (Properties):***

1) MenuEditor (szerkesztőablak):

Kattintsunk az elhelyezett ikonra kétszer, és megjelenik a menütervező. Nyomjuk meg a jobb egérgombot egy elemen állva, és akkor egy lenyíló-menüből beilleszthetünk egy elem mögé *(Insert New Item (after))*, elé *(Insert New Item (before))*, törölhetjük a kiválasztottat *(Delete Item)*, almenüt hozhatunk létre belőle *(Create Submenu)*. Ha a menüpontok sorrendjét meg szeretnénk változtatni, azt a lenyíló-menü mozgatás fel, vagy balra *(Move Up (or left))*, vagy a másik irányba a mozgatás le, vagy jobbra *(Move Down (or right))* megnyomásával tehetjük meg. A maradék három menüponttal sablonokat szúrhatunk be, menthetünk, vagy törölhetünk. Alább látható a menüszerkesztő, és a lenyíló-menü



A komponensfigyelőben (Object Inspector) láthatjuk, hogy minden menüelem külön komponens, saját tulajdonságokkal, és eseményekkel. Ha elválasztó vonalat szeretnénk elhelyezni két elem közé, illesszünk be egy új menüelemet, és legyen ennek a felirata „-„.

2) Images (képek):

A menü elemeihez lehetőségünk van képeket tenni. Ha azt szeretnénk, hogy a menüelemekhez ne külön-külön kelljen képeket rendelni, akkor használjunk ImageList-et (lásd később). Ez a képekhez sorszámokat rendel, így csak ezeket kell majd az elemek ImageIndex mezőjében megadnunk.

❖ ***A főmenü (TMainMenu) – Elemeinek tulajdonságai:***

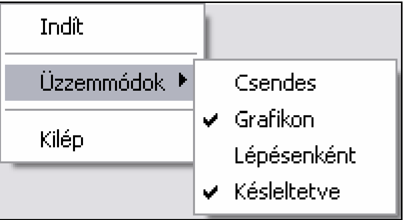
1) Checked (kiválasztottság) és AutoCheck (autómatikus kijelölés):

A menüelemek nem csak eljárások lefutását indíthatják el, olykor visszaigazolásra is használható (pl. be van-e kapcsolva valamilyen szolgáltatás, vagy nem). Ilyenre láthatunk példát a Word-ben is, a Nézet-menüpontban a bekapcsolt vonalzót egy pipa jelzi. Ha valamelyik elem elé ilyen jelölőt szeretnénk tenni, vagy éppen eltüntetni, ehhez állítsuk át a menüelem Checked tulajdonságát! Az alábbi példában egy olyan eljárást látunk, amely a *MenuItem3* nevű menüelem elé tesz egy pipát, vagy vesz el, ha már volt ott.

```
procedure TForm1.MenuItem3Click(Sender: TObject);
begin
   If MenuItem3.Checked=True then MenuItem3.Checked:=False
                             else MenuItem3.Checked:=True;
end;
```

Viszont felesleges rá eljárást írni, mert minden elemnek van egy AutoCheck tulajdonsága is, amit ha bekapcsolunk, a fenti eljárás elágazásának utasításai automatikusan lezajlanak, ha a felhasználó rákattint az elemre.

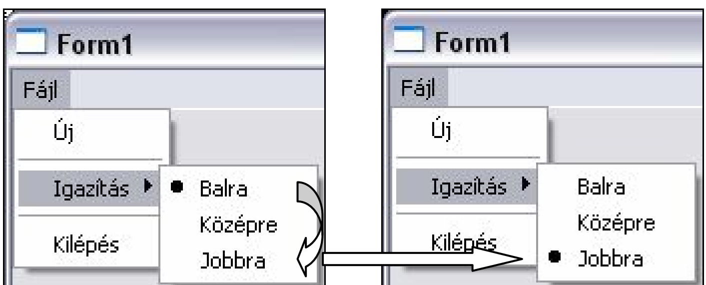
Az alábbi példában egy szimulációs program menüjét láthatjuk, ahol például a grafikont ki, vagy bekapcsolhatjuk.



2) GroupIndex (csoportszám) és RadioItem (választóelem);

Az előző módszerrel olyan kapcsolókat készítettünk, melyeket egymástól függetlenül lehetett ki-be kapcsolgatni. Elképzelhető azonban, hogy egymást kölcsönösen kizáró menüpontokat szeretnénk használni. Ha egy ilyen elemre kattint a felhasználó, az „új" menüpont lesz megjelölve a „régi" helyett. Például, ha egy szövegszerkesztő programnál rákattintunk arra, hogy jobbra szeretnénk igazítani a szöveget, és eddig balra volt, akkor a jelölés a *Jobbra* menüpont elé kerül, és eltűnik a *Balra* menüponttól.

A probléma megoldásához előbb alkossuk meg a (logikailag) egybe tartózó elemekből álló csoportokat, melyeket azonos, nullától különböző (!) GroupIndex jellemez. Például, ha három menüelem e tulajdonságát 1-re állítjuk, akkor ők egy csoportot alkotnak. Állítsuk be minden csoport elemeinek AutoCheck mezőjét Igaz-ra, hogy a program magától állítsa be a jelöléseket. Majd az elemek RadioItem tulajdonságát is módosítsuk Igaz értékűre, ezzel jelezve a programnak, hogy egymást kizáró menüpontokról van szó.



3) Bitmap (ikon rendelése):

Ha csak kevés képet szeretnénk betenni a menünkbe, nem fontos ImageListet használnunk, elég, ha minden menüelemhez külön hozzárendeljük a képet (a Bitmap mezőben). Ezek lehetnek ico; png; xpm; vagy bmp kiterjesztésűek. Természetesen kisméretű képekkel érdemes dolgozni.

4) Default (alapértelmezett):

Az igazán fontos eseményt tartalmazó menüelemre beállíthatjuk ezt a tulajdonságot. Ekkor az őt tartalmazó menüelemre duplán kattintva lefut ez az esemény, és a menüpont felirata vastagon szedett lesz.

Megjegyzés: ez az elvi működése ennek a tulajdonságnak, de sajnos meg kell elégedjünk a kivastagított felirattal.

5) ImageIndex (képsorszám):

Ha kényelmesebb megoldást szeretnénk, mint, hogy minden elemhez külön rendeljük a képet, gyűjtsük össze az összes képet egy ImageList-be. Ez sorszámozza azokat, így már csak az indexüket kell megadnunk, hogy ott álljon a kép a menüelem előtt.

6) RightJustify (jobbra igazítás):

A menüsorban lévő elemeknél van értelme ezt a változót átállítani. Ilyenkor ez, és az utána következő összes elem a menüsorban jobbra lesz igazítva.

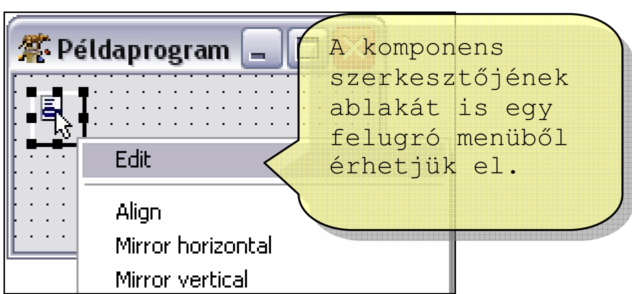
7) ShortCut (gyorsbillentyű):

Ezzel olyan billentyűkombinációkat állíthatunk be, melyek segítségével a menüelemekhez használt eseményeket tudjuk meghívni a menü megnyitása nélkül. Például, a mentésre beállíthatjuk a Ctrl+S kombinációt, ahogy azt sok programban láthatjuk.

- ***A felugró menü (TPopupMenu):***

A felugró menük működésükben szinte teljesen azonosak a főmenüvel, így csak röviden ismertetjük használatát. Az elemeket ugyanúgy lehet szerkeszteni, a tulajdonságai is azonosak, az egyetlen eltérés csupán abban van, hogy a felugró menüket hozzá kell rendelni valamilyen komponenshez, formhoz. Például, ha az utóbbihoz akarjuk hozzárendelni, akkor miután felépítettük a felugró menünket, a form tulajdonságai között keressük meg a PopupMenu mezőt, és a lenyíló-menüből válasszuk ki azt a felugró menüt, amit hozzá akarunk rendelni. Ezek után, ha lefordítjuk a programunkat, a form egy üres részére jobbklikkre előjön a menünk.

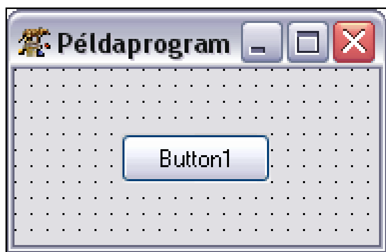
A komponens elhelyezve egy üres formon:



❖ ***A gomb (TButton) – Bevezető:***

A gombokat események előidézésére használjuk elsősorban, így leggyakrabban az OnClick eseményt írjuk meg. Ha a formra helyezett gombra duplán kattintunk, akkor a forrásban elkezdhetjük írni azt az eljárást, amit akkor akarunk lefuttatni, amikor a felhasználó megnyomja a gombot.

A komponens elhelyezve egy üres formon:



❖ ***A gomb (TButton) – Tulajdonságok (Properties):***

1) Cancel (visszavonás):

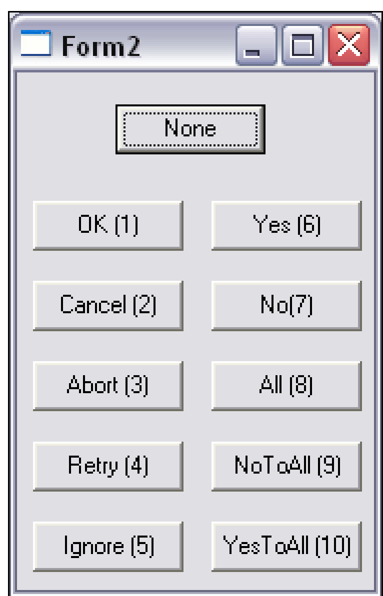
Ha ezt a logikai változót Igaz-ra állítjuk, akkor a gomb OnClick eseménye lefut, ha megnyomjuk az „Esc”-et. Több gombnál is beállíthatjuk, de közülük a Tab-sorrendben legelső gomb eljárása fog csak lefutni.

2) Default (alapértelmezett):

Ha ezt a logikai változót Igaz-ra állítjuk, akkor a gomb eljárása lefut, ha megnyomjuk az „Enter”-t. Ezt is beállíthatjuk több gombnál, de itt is a Tab-sorrendben legelső gomb eljárása fog csak lefutni.

3) ModalResult (a végrehajtás eredménye):

Ha meghívunk egy formról (Pl. Form1) egy másik formot (Pl. Form2), azt megtehetjük úgy, hogy Form2.Show, vagy úgy, hogy Form2.ShowModal. Ez utóbbi egy egészértékkel visszatérő függvény. Ebből megtudhatjuk, hogy milyen gombbal léptünk ki a formról. Az alábbi képen, baloldalt látjuk, hogy milyen lehetséges beállítási módok lehetnek, zárójelben a visszatérési értékeiket. Ha olyan gombbot akarunk használni, amire nem szeretnénk, hogy bezárja az ablakot, állítsuk be az mrNone (ez az alapértelmezett) állapotot:



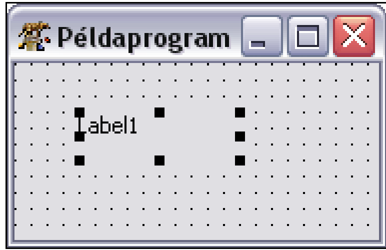
```
procedure TForm1.Button1Click(Sender: TObject);
begin
   ShowMessage(IntToStr(Form2.ShowModal));
end;
```

A fenti kód segítségével egy felugró ablakba írathatjuk ki, milyen gombbal zártuk be az ablakot.

❖ A címke (TLabel) – Bevezető:

Lazarus-ban másképp lehet szöveget kiírni a képernyőre, mint ahogy azt Pascal-ban tettük (Write, és Writeln nem használható, mivel grafikus rendszerről beszélünk). Ehelyett megjelenítésre használhatjuk a címkéket. Ezeket nem változtathatja a felhasználó, tartalmuk módosítása csak kódból lehetséges. Kiválóan alkalmas egy szövegbeviteli komponens (pl. editbox) mellé rakni, hogy tudja a felhasználó, hogy milyen adatra vár a program.

A komponens elhelyezve egy üres formon:



❖ A címke (TLabel) – Tulajdonságok (Properties):

1) Alignment (igazítás):

A címke szövegrésze nem tölti ki teljesen a teret. Itt beállíthatjuk, hogy balra (taLeftJustify), középre (taCenter), vagy jobbra (taRightJustify) legyen igazítva.

2) FocusControl (kijelölés-szabályozás):

Egy speciális karakterrel („&”) megjelölhetjük a címke egy betűjét: pl. „Men&tés”, ami futás közben így jelenik meg: „Mentés” („t” aláhúzva). Ha megnyomjuk az „Alt”+”ez a karakter” (jelen esetben az „Alt+t”) kombinációt, akkor a kijelölés arra a komponensre kerül, amit beállítottunk a FocusControl mezőben.

3) Layout (elrendezés):

Hasonló, mint az Alignment, annyi különbséggel, hogy ez függőlegesen igazítja a szöveget. Felülre rendezhetünk (tlTop), középre (tlCenter), vagy alulra (tlBottom).

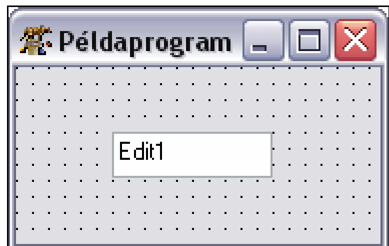
4) ShowAccelChar (aláhúzás engedélyezése):

Ha Igaz-ra állítjuk értékét akkor, amely karakter elé írjuk a „&” karaktert, az aláhúzva jelenik meg, és működik az átirányítás a FocusControl-lal. Ha a címkében szükségünk lenne a & karakterre, akkor kettőt kell beírni a Caption mezőbe egymás mellé. Ha nem engedélyezzük, nem veszi figyelembe a FocusControl-t.

❖ ***Az egyszerű szövegmező (TEdit) – Bevezető:***

Ahogy a Write-ot, és a Writeln-t sem szokás Lazarus-ban használni, úgy a beolvasásra sem a Read, és a Readln eljárást használjuk. Számtalan eszköz található az adatok bevitelére, melyek közül az editbox (szövegdoboz, szövegmező) az egyik legegyszerűbb. Ez egy sor szöveg megjelenítésére, módosítására, beírására használható.

A komponens elhelyezve egy üres formon:



❖ ***Az egyszerű szövegmező (TEdit) – Tulajdonságok (Properties):***

1) CharCase (karakterek típusa):

Ezzel a mezővel beállíthatjuk, hogy amilyen betűket beírunk, azok kisbetűsek (ecLowerCase), nagybetűsek (ecUppercase), vagy vegyesen kis, ill. nagybetűsek (ecNormal) legyenek.

2) EchoMode (visszhang-viselkedés) és PasswordChar (jelszókarakter):

Eredetileg, ha a szövegbeviteli mezőben vagyunk, akkor ott megjelenik az, amit írunk. Ilyenkor az EchoMode-nál az emNormal felirat áll. Ha azt szeretnénk, hogy amit beírtunk az a képernyőn ne jelenjen meg, akkor az EchoMode mezőben válasszuk az emNone lehetőséget. Ha csak „*”-t (vagy más karaktert) szeretnénk látni minden karakter helyett, akkor állítsuk be az emPassword lehetőséget, és a PasswordChar-nál írjuk be, hogy mit szeretnénk látni a valódi karakterek helyett.

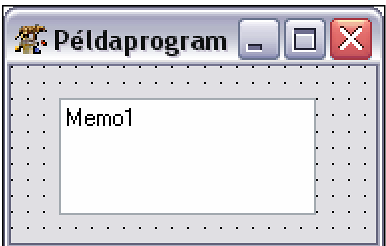
3) ReadOnly (csak olvasható):

Ha ezt a logikai változót Igaz-ra állítjuk, akkor a felhasználó nem tud majd beleírni az editbox-ba.

❖ ***A jegyzettömb (TMemo) – Bevezető:***

Ez a komponens – az előzővel ellentétben – már több soros szöveg bevitelére, megjelenítésére alkalmas. Viszont itt sem lehet a szavakat külön formázni, csak a Memo egészére határozhatjuk majd meg a betűtípust, betűszínt, stb.

A komponens elhelyezve egy üres formon:



❖ ***A jegyzettömb (TMemo) – Tulajdonságok (Properties):***

1) Lines (sorok):

A Memo sorait a Lazarus egy 0-tól indexelhető, szöveg típusú tömbben tárolja. Ha az Object Inspectorban a Lines melletti TStrings-nél lévő gombra kattintunk, megjelenik egy szövegszerkesztő ablak, ahol beírhatjuk a szöveget, amit a felhasználó a program elindulásakor a Memo-ban fog látni.

2) <u>A Lines leghasznosabb függvényei:</u>

Mivel tömbről van szó, lekérdezhetjük a tömb hosszát, ami a „Lines.Count” függvény visszatérési értéke.

Lehetőségünk van szöveges fájlból közvetlen beolvasásra, amit a *Lines.LoadFromFile(s:FileName)* paranccsal oldhatunk meg. A mentést hasonlóan, *Lines.SaveToFile(s:FileName)* eljárás hajtja végre (FileName=String).

Hozzáfűzhetünk a Memo-hoz a *Lines.Add(s:String):Integer* függvénnyel, vagy a *Lines.Append(s:String)* eljárással. A különbség annyi, hogy az előbbi egy olyan függvény, amely visszaadja, hányadik sorba írt bele.

3) <u>ReadOnly (csak olvasható):</u>

Hasonlóan az editbox-hoz ha ezt Igaz-ra állítjuk, nem írhatunk a Memo-ba.

4) <u>ScrollBars (görgetősávok):</u>

Ezzel statikus (mindig látható) vízszintes (ssHorizontal), függőleges (ssVertical), ill. mindkét irányba (ssBoth) görgetősávokat tudunk rendelni a Memo-khoz. Mindegyiknek van „Auto”-s kivitele, amivel a görgetősáv csak akkor válik érzékennyé, ha van értelme.

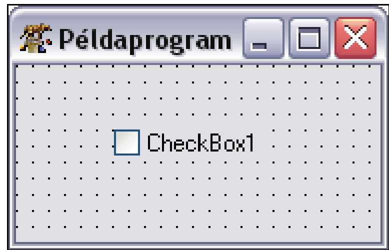
5) <u>WordWrap (sortörés):</u>

Ha bekapcsoljuk ezt a logikai változót, akkor, ha a Memo-ban a szöveg a beviteli komponens széléhez ér, automatikusan új sora ugrik a kurzor.

❖ ***<u>A jelölőnégyzet (TCheckBox) – Bevezető:</u>***

Ezt a komponenst leggyakrabban kapcsolóként használhatjuk. Olyan tulajdonságokat vezérelhetünk vele, aminek két, vagy legfeljebb 3 állapota van. A komponens segítségével leolvashatjuk és beállíthatjuk e tulajdonságokat. Leginkább beállításoknál találhatunk ilyeneket, annak a vezérlésére, hogy valamilyen vezérlő eszköz be van-e kapcsolva, vagy nincs (jelölőnégyzetben ott a pipa, vagy nem).

A komponens elhelyezve egy üres formon:



❖ ***<u>A jelölőnégyzet (TCheckBox) – Tulajdonságok (Properties):</u>***

1) <u>AllowGrayed (szürkítés engedélyezése):</u>

Ezzel engedélyezhetjük, hogy ne két különböző állapota legyen, hanem három.

2) <u>Checked (kiválasztott):</u>

Beállíthatjuk (vagy lekérdezhetjük), a négyzet állapotát. (van-e pipa, vagy nincs)

3) <u>State (állapot):</u>

Hasonlóan, mint az előzővel, ezzel is állapotot határozhatunk meg, csak itt kiegészül a harmadik lehetőséggel, a szürkítettel (cbGrayed).

❖ **_A választógomb (TRadioButton):_**

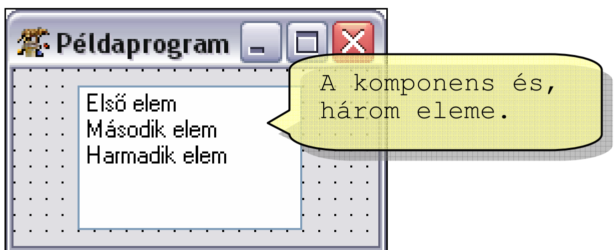
Ez is állapotok kijelzésére szolgál, a különbség csak annyi, hogy itt, az egy csoportban (egy formon, egy GroupBox-ban) lévő komponensek közül csak egyet választhatunk ki, és mindössze két állapot közül választhatunk. A tulajdonságai ugyanazok, mint a jelölőnégyzetéi.

A komponens elhelyezve egy üres formon:

❖ **_A listadoboz (TListBox) – Bevezető:_**

A listadoboz funkciója hasonló, mint a választógomboké, és a jelölőnégyzeteké, itt is állapotokat követhetünk nyomon, állíthatunk be. Lehetőség van egy, vagy több elem kijelzésére is. A program futása közben, az, hogy melyik elem van kiválasztva, az ItemIndex változóba kerül. Ha több elem van kiválasztva, akkor azoknak a számát a SelCount-tal kérdezhetjük le, és a kiválasztottak a Selected tömbbe kerülnek (ez egy logikai tömb, ahol azok az elemek vannak kiválasztva, amelyek indexénél ez a tömb Igaz értéket tartalmaz). A példában látszik milyen hasonló a funkciója a CheckBox-okéval: ezt is megoldhattuk volna 4 jelölőnégyzettel, viszont ott lekérdezni, hogy mely négyzetek vannak bepipálva, már nehezebb lett volna.

A komponens elhelyezve egy üres formon:



❖ **_A listadoboz (TListBox) – Tulajdonságok (Properties):_**

1) ExtendedSelect (kiterjesztett kiválasztás) és MultiSelect (többszörös kiválasztás):

Ha engedélyezzük a többszörös kiválasztást, akkor egyszerre több elemet is kiválaszthatunk a Ctrl, vagy a Shift gomb lenyomva tarásával, és kattintással a kiválasztandó elemekre. Csak ebben az esetben van értelme a kiterjesztett kiválasztásnak. Ha ki van kapcsolva, akkor úgy működnek az elemek, mint a jelölőnégyzetek, több elem kiválasztásához elég rákattintani az elemekre. Ha bekapcsoljuk, akkor a Ctrl lenyomásával kiválaszthatunk olyan elemeket, melyek nem feltétlenül szomszédosak. A Shift nyomva tartásával, és két elemre kattintással pedig „intervallumokat" jelölhetünk ki.

2) ItemHeight (elemmagasság):

Ha a Style-t lbOwnerDrawFixed-re állítjuk, akkor ebben a mezőben beállíthatjuk az egyes listacellák magasságát.(A Style nem megfelelő működése miatt nem használható!)

3) Items (elemek):

Hasonlóan, mint a Memo Lines tulajdonságánál, itt is megszerkeszthetjük a lista elemeit. Az egyes elemeket külön sorba kell írni. Ez is egy szövegtípusú tömb, ugyanúgy 0-túl indexel, és ugyanúgy van egy listaszerkesztő ablaka is.

4) Sorted (rendezettség):

Ha igazra állítjuk, akkor lerendezi (abc rendbe szedi) a listaelemeket.

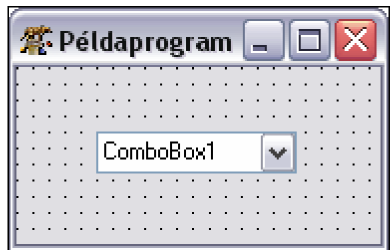
5) Style (stílus):

Az előbb már szó volt róla, hogy beállíthatjuk a listamezők magasságát, ha itt kiválasztjuk az lbOwnerDrawFixed mezőt. Ha változó mezőnagyságot szeretnénk, akkor válasszuk az lbOwnerDrawVariable mezőt. Az lbStandard a betűk nagyságához igazítja a listamezők magasságát. Sajnos ebben a verzióban a Style nem működik megfelelően.

❖ A kombinált lista (TComboBox) – Bevezető:

Ez egy helytakarékosabb, könnyebben kezelhető, és még inkább felhasználóbarát lista, mint a ListBox. Olyan, mintha egy lenyíló listát kiegészítettünk volna egy editbox-szal. Ha meg akarjuk nézni, hogy a felhasználó mit választott, egyszerűen csak a Text tulajdonságot kell lekérni. Ha különleges kívánsága van a felhasználónak, új elemmel bővítheti a listát.

A komponens elhelyezve egy üres formon:



❖ A kombinált lista (TComboBox) – Tulajdonságok (Properties):

1) AutoComplete (automatikus kiegészítés):

Ha a felhasználó elkezdi beírni az elemet, akkor a program megpróbálja a meglévő elemekkel kiegészíteni.

2) AutoDropDown (autómatikus lenyílás):

A működéséhez be kell kapcsoljuk az AutoComplete mezőt. Ha Igaz-ra akkor, amikor elkezdjük beírni a ComboBox-ba az elemet, akkor lenyílik a lista a lehetőségekkel, nem csak kiegészíti velük.

3) DropDownCount (lenyílószámláló):

Meghatározhatjuk, hogy a lenyíló listánk hány elemet jelenítsen meg.

4) ItemHeight (elemmező nagyság):

A lenyíló lista mezőinek nagyságát adhatjuk meg.

5) MaxLength (maximális hosszúság):

Beállíthatjuk a beíró mezőbe maximálisan beírható elemek számát (ha ez 0, akkor nincs megkötés).

Megjegyzés: tapasztalataink szerint nem működik.

6) ReadOnly (csak olvasható):

Az edit mező beleírhatóságát vezéreli. Mivel a stílusok általában felhasználják ezt a mezőt, gyakran nem is tudjuk megváltoztatni ennek a logikai változónak ez értékét.

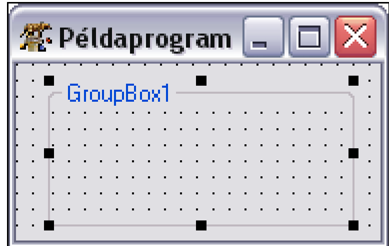
7) Style (típus):

Az egyik legfontosabb tulajdonság, amely a lista-elemek kiválasztását határozza meg.

Érték	Jelentése
csDropDown	Lenyíló lista, ahol az edit-mezőbe be lehet írni elemeket.
csDropDownList	Olyan lenyíló lista, mint az előbbi, csak itt nem lehet beírni az elemet az edit-mezőbe, csak kiválasztani lehet a lehetőségekből.
csSimple	Megjelenik az edit-mező (ahova be lehet írni), és alatta a lista is, ahonnan az elemeket kiválasztva bekerülnek az edit-mezőbe.

❖ ***A csoportmező (TGroupBox) – Bevezető:***

A logikailag egy csoportba tartozó elemeket (például ugyanannak a beállítására vonatkozó kapcsolókat, listákat) szoktuk egy csoportmezőbe rakni.
A komponens elhelyezve egy üres formon:



❖ ***A csoportmező (TGroupBox) – Tulajdonságok (Properties):***

1) ChildSizing (gyerek méretezése):

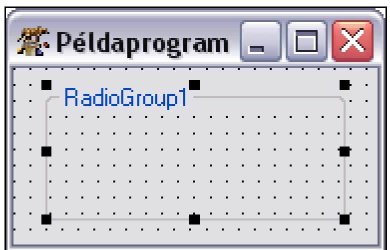
Ha szeretnénk valami rendszert vinni a csoportmezőnk szerkezetébe, akkor használjuk ezt a tulajdonságot, amivel margót határozhatunk meg minden irányban, az elrendezendő elemek sorrendjét, az egy sorban lévő maximális komponensszámot.

2) ClientHeigh és ClientWidth (felhasználható magasság és felhasználható szélesség):

A csoportmező felhasználható területét állíthatjuk be.

❖ ***A választógomb csoportmező (TRadioGroup) – Bevezető:***

Ha a csoportmezőt, egy vezérlővel szeretnénk bővíteni, az elég bonyolult. Itt a csoportmezők egy speciális fajával ismerkedhetünk meg, ezért az azonos tulajdonságokat most sem emelnénk ki újra. Ugyan ide is lehet tetszőleges vezérlőt tenni, mégsem erre való elsődlegesen, hanem arra, hogy választógombokat helyezzünk el rajta:
A komponens elhelyezve egy üres formon:



❖ ***A választógomb csoportmező (TRadioGroup) – Tulajdonságok (Properties):***

1) AutoFill (automatikus kitöltés):

A választógombok egyenletesen fognak eloszlani, ha bekapcsoljuk ezt a szolgáltatást.

2) Columns (oszlopok) és ColumnLayout (oszlopelrendezés):

Az oszlopok számát a Columns változóban állíthatjuk be, és a ColumnLayout tulajdonságban pedig azt, hogy az elemeket oszlop-folytonosan (clVerticalThenHorizontal), vagy sorfolytonosan (clHorizontalThen-Vertical) jelenítse-e meg a program.

3) ItemIndex (a kiválasztott sorszáma):

Ez a változó tartalmazza a kiválasztott elem sorszámát. Ha azt szeretnénk, hogy ne legyen ilyen, állítsuk be ezt az értéket -1-re.

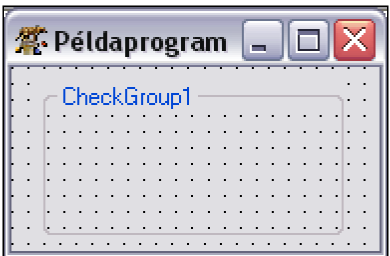
4) Items (elemek):

A választógombokat ebben a (szövegtípusú) tömbben tároljuk. Ha újat akarunk hozzáadni, akkor azt az *Add* függvénnyel, és a gomb nevének megadásával tehetjük meg. RadioGroup1.Items.Add(->hozzáfűzendő választógomb felirata<-).

❖ ***A jelölőnégyzet csoportmező (TCheckGroup):***

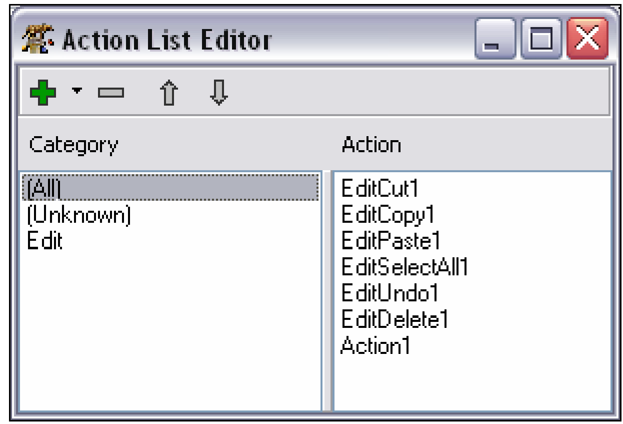
Ugyanaz, mint az előző, annyi különbséggel, hogy itt jelölőnégyzetek vannak. Mivel itt több lehetőséget is kiválaszthatunk, a lekérdezés nem működik az ItemIndex változóval. Ehelyett van egy (logikai) tömbünk, ami tartalmazza, hogy melyik elemek vannak bejelölve: CheckGroup1.Checked[->elemindex<-]. Természetesen nullától indexeljük ezt a tömböt.

A komponens elhelyezve egy üres formon:



❖ ***Az eseménylista (TActionList) – Bevezető:***

Ha ugyanazt az eljárást több vezérlő is meghívja (például jelölőnégyzet, gomb, menüelem, stb.), akkor rendelhetünk hozzá egy eseményt, és elég azt meghívnunk a különböző helyeken. Ehhez még nem kellene az eseménylista. Ennek az ereje abban van, hogy ha a vezérlők, amik meghívják az eljárásokat, azok visszajelzést is adnak, akkor nem kell külön foglalkoznunk a különböző vezérlők állapotainak beállításával, hanem csak az esemény állapotát kell átállítani. Ha például, készítünk egy szövegszerkesztőt, aminek van egy mentés eljárása, amit meghívhatunk menüből is, és egy gomb eljárásaként is, azt kiszürkítjük, ha nincs mit elmentsen, mert nem történt változás. Így elég csak egy eseményt írni, és annak az Enabled tulajdonságát Hamis-ra állítani. Az egyes események szerkesztését, hozzáadását, törlését, áthelyezését az Action List Editorban tehetjük meg, ahol kategóriákba sorolva találhatjuk meg az eseményeket. Ez úgy érhető el, ha a formon az ActionList komponensre kétszer kattintunk.



❖ ***<u>Az eseménylista (TActionList) – Tulajdonságok (Properties):</u>***

1) <u>AutoCheck (automatikus kiválasztás), Checked (kijelölt), Enabled (engedélyezés), GroupIndex (csoportszám), ImageIndex (képsorszám), Visible (láthatóság), ImageIndex (képsorszám):</u>

Amikor menüelemekhez, CheckBox-okhoz, gombokhoz, stb. rendeljük hozzá ezt az eseményt, az azonos nevű tulajdonságukat vezérli. Tehát, ha ugyan azt az action-t rendeltük hozzá egy gombhoz, és egy menüelemhez, akkor az Action Caption mezőjében lévő szöveg fog megjelenni a gombon és menüpont szövegében is.

2) <u>ShortCut (billentyűparancsok):</u>

Billentyűparancsokat állíthatunk be az eseményeinkhez, ilyen például a szövegszerkesztőkben a másoláshoz a Ctrl+C.

3) <u>SecondaryShortCuts (másodlagos billentyűparancsok):</u>

Ha több billentyűparancsot szeretnénk ugyanahhoz az eseményhez rendelni, itt beírhatjuk azokat.

Az Additional-paletta (kiegészítők)

Mint eddig láthattuk az alap komponenskészletünkkel nagyon sok mindent meg tudtunk valósítani. Most azonban olyan eszközöket ismertetünk, melyek nemcsak rendkívül hasznosak tudnak lenni, de sokkal színesebb, átláthatóbb, s nem utolsó sorban még inkább felhasználóbarát programok készítését teszik lehetővé. A paletta képe:

Ahogy a standard-oknál, itt is találunk gombot és szövegek bevitelére alkalmas komponenseket, ezek azonban egyéb lehetőségekkel is szolgálnak „rokonaikhoz” képest, úgy is mondhatnánk, hogy kiterjesztései, kiegészítései azoknak. Az itt meglevő két gombon képeket helyezhetünk el, s az egyik fajtából akár „kapcsolót” is készíthetünk. Szót ejtünk egy olyan szövegbeviteli mezőről, melyhez címke tartozik, és egy olyanról is, melynél a bevitelt általunk megadott szabályokkal korlátozhatjuk. A fejezetben két olyan eszközzel is megismerkedünk, melyekhez hasonlóval eleddig nem találkoztunk: az egyik rajzolásra és képek megjelenítésére alkalmas, a másik, pedig egy táblázat.

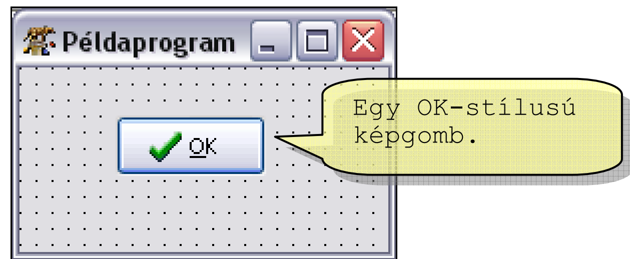
Összefoglalva tehát a paletta hat elemét mutatjuk be, melyek szokásunkhoz híven tematikusan összefoglaltunk:

- Képgomb
- Gyorsgomb
- Képkezelő
- Címkézett beviteli mező
- Maszkolható beviteli mező
- Szöveges táblázat

❖ ***A képgomb (TBitBtn) – Bevezető:***

Látni fogjuk, hogy ez a komponens szinte megegyezik a már korábban, az általános elemek közt megismert egyszerű gombbal (TButton). Mindössze annyival tud többet, hogy lehetőségünk van képet elhelyezni rajta, sőt még a számunkra fontos eseményei is megegyeznek. Éppen ezért csak az extrákat fejtjük ki.

A komponens elhelyezve egy üres formon:



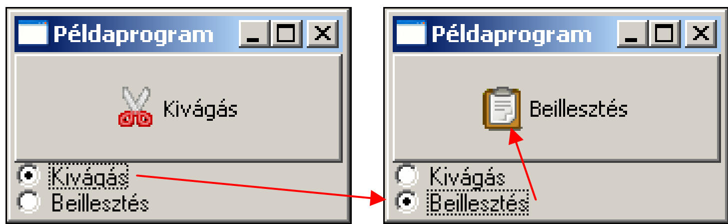
❖ ***A képgomb (TBitBtn) – Tulajdonságok (Properties):***

1) Glyph (díszítés):

Egy ablak segítségével képet helyezhetünk el a gombon. Tapasztalataink szerint a kép először csúnya feketeségként jelenik meg, de ha a Spacing (lásd később) tulajdonságot legalább egyszer módosítjuk – kép beiktatása után – akkor tökéletesen működik. Ez érvényes kódból történő módosításkor is.

Megjegyzés: a „Lazarus\images\22x22” könyvtárban szép, és hasznos képek találhatók.

A futás közben történő módosításhoz mutatunk egy példát:



Mint látható a fenti programképeken, példánkban egy formra helyeztünk két rádiógombot, és egy képgombot. A programot elmentettük egy könyvtárba, majd ugyanebben a mappában létrehoztunk egy „kepek” nevűt is, ahova két ikonfájlt helyeztünk (kivag.ico, beilleszt.ico). A kis alkalmazásunk nem tesz mást, minthogy a gomb képét változtatja a rádiógomboktól függően. A két rádiógomb OnClick eseményeivel ezt könnyen elérhetjük, például így:

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
   BitBtn1.Glyph.LoadFromFile('kepek\kivag.ico');
   BitBtn1.Caption:='Kivágás';
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
   BitBtn1.Glyph.LoadFromFile('kepek\beilleszt.ico');
   BitBtn1.Caption:='Beillesztés';
end;
```

Megjegyzés: egy érdekes, de bosszantó jelenségre hívnánk fel a figyelmet, jelesül arra, hogy míg szerkesztés közben az összes Lazarus által kínált formátumú kép használható, addig a program futása közben csak ikonfájlokat (.ico) illetve tömörítetlen windows-képfájlokat (*.bmp) képes használni. A forrás szerint még tudja a Pixmap (*.xpm) állományokat is használni, de ennek kipróbálására nem volt lehetőségünk.*

2) Kind (fajta):

Ennek a segítségével előre beállított típusú gombfajták közül választhatunk. Alapértelmezésben a semleges (bkCustom) van kiválasztva, amennyiben ezt módosítjuk, megváltozik a gomb képe, és a végrehajtási eredménye (ModalResult). Talán leggyakoribb alkalmazása a kilépésgomb (bkClose), aminek a fentieken túl további előnye, hogy alapértelmezésként hozzárendelődik a Close utasítás, azaz az aktuális ablakot (általában formot) bezárja.

Megjegyzés: ez nem feltétlenül jelenti az alkalmazás végét!

3) Layout (elrendezés):

A gomb képének helyét tudjuk változtatni ennek a tulajdonságnak a segítségével. Négy lehetőség közül választhatunk, és kis angol tudással könnyen rájövünk melyik hova helyezi a képet, de azért mi sorra vettük őket: a blGlyphBottom alulra, a blGylphTop felülre, a blGlyphLeft balra, míg a blGlyphRight jobbra teszi a képet a felirathoz képest.

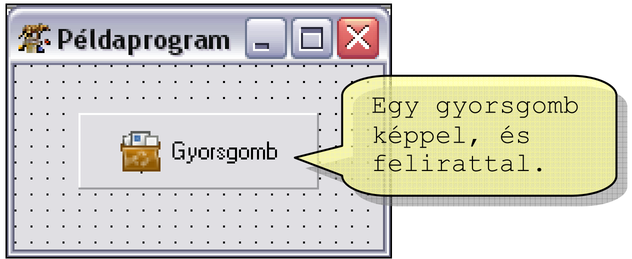
4) Spacing (térköz):

A gomb felirata és beillesztett kép közötti távolságot lehet segítségével megadni, és értéke pixelben értendő.

❖ ***A gyorsgomb (TSpeedButton) – Bevezető:***

Mivel ez a komponens is gomb, így tulajdonságaiban ez sem tér el a sima gombtól nagymértékben, így itt is elegendő lesz a különbségekre felhívni a figyelmet. Ezek a különbségek azonban elegendők arra, hogy szinte teljesen más felhasználási területeken használjuk ezt a komponenst. Például fontos ilyen különbség, hogy nem adható meg hozzá végrehajtási eredmény (ModalResult), valamint azt is érdemes megemlíteni, hogy a gyorsgomb sosem érhető el a Tab billentyűvel, így nincs is Tab-sorrendbeli helye (TabOrder).

A komponens elhelyezve egy üres formon:



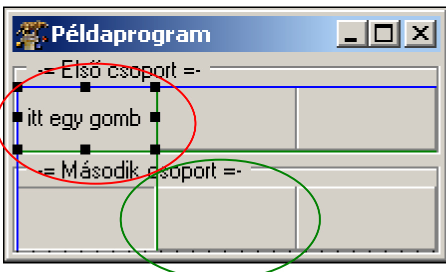
❖ ***A gyorsgomb (TSpeedButton) – Tulajdonságok (Properties):***

1) Caption (felirat):

Igazság szerint egy elég ritkán, mondhatni szinte soha nem használt tulajdonsága. Pusztán ezért említettük, hiszen ha mégis használni szeretnénk, nem különbözik a gombnál megszokottól.

2) Flat (laposság):

A gyorsgomb különlegessége, hogy készíthetünk belőle a háttérbe simuló, úgynevezett lapos gombot. Ehhez mindössze Igaz-ra kell állítanunk ezt a tulajdonságot.



Az ábrán látható, hogy a piros színnel karikázott gomb szinte észrevehetetlen, míg a zölddel karikázott kiemelkedik a háttérből. De ilyen lapos gombbal találkozunk nagyon sok program vezérlőpalettáin, hogy messze ne menjünk, Lazarus-ban is ilyen gombokkal választunk a komponensek között, ilyen gombbal érhetjük el a mentést stb.

3) Glyph (díszítés):

Mindenben megegyezik a képgombnál leírtakkal kivéve talán azt, hogy nem tapasztaltuk azt a furcsa jelenséget a szerkesztés közbeni módosításkor (Spacing-állítás).

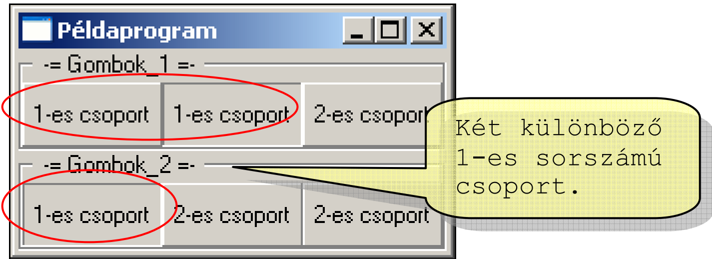
4) GroupIndex (csoportszám), Down (kiválasztottság), AllowAllUp (mindet felenged):

A GroupIndex tulajdonsággal csoportosítani tudjuk az egy komponensen belüli gyorsgombjainkat. Ehhez nem kell mást tenni, mint egy nullánál nagyobb értékű egész számra állítani a tulajdonságot. Az ugyanazon csoportban lévő gombok közül, ha az egyikre kattint a felhasználó, az „benyomva” marad mindaddig, míg egy másik csoportbéli elemre nem kattint. Így a gombok egymást kizáró módon használhatók. Az AllowAllUp tulajdonság Igaz-ra állításával elérhető az, hogy egy csoporton belül a lenyomott gombot ismét „felengedjük” egy kattintással. Fontos tudnunk azt a nyilvánvalóan logikus tényt, hogy ha egy gombcsoport egy elemén megváltoztatjuk az AllowAllUp tulajdonságot, az a többiét is módosítja.

Ha 0-n hagyjuk a GroupIndex értékét, akkor a gomb „hagyományos” módon működik, azaz ha megnyomtuk, „visszarúgja” magát.

A Down tulajdonság segítségével főként lekérdezhetjük, hogy egy adott gyorsgomb ki van-e választva vagy sem, esetenként pedig, kódból módosíthatjuk a gomb kiválasztottságát. Utóbbi lehetőséggel azonban óvatosan bánjunk, hiszen hatását egy másik csoportbeli gombon is éreztetheti.

Egy példát mutatunk csoportosításra:



Láthatjuk, hogy két különböző komponensben már lehetnek azonos sorszámú csoportok, vagyis a már használt számok ismét felhasználhatók lesznek, ha más a tartalmazó komponens.

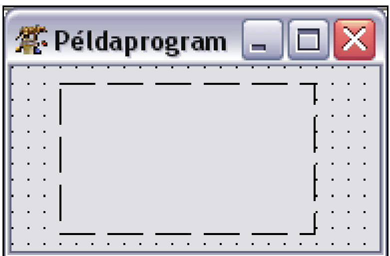
5) Layout (elrendezés), Spacing (térköz):

Szintén megegyezik a képgombnál ismertetekkel.

❖ A képkezelő (TImage) – Bevezető:

A most ismertetésre kerülő komponens alkalmas – elviekben – képek megjelenítésére, és rajzolásra. Sajnos, mint látni fogjuk, a képek megjelenítése még némi gondot okoz neki, viszont rajzolásra kiválóan alkalmas. Fontos egyedi tulajdonságokkal nem találkozunk így ez a rész nem lesz túl hosszú, használatáról viszont igyekeztünk bővebben szólni.

A komponens elhelyezve egy üres formon:



❖ A képkezelő (TImage) – Tulajdonságok (Properties):

1) AutoSize (automatikus méretváltozás):

Ha Igaz-ra állítjuk értékét, a betöltött kép méretéhez fog igazodni komponensünk. Ha az (alapértelmezett) Hamis-t választjuk, akkor a Center és a Strech tulajdonságok (lásd később) határozzák meg az Image viselkedését kép használatakor.

2) Center (középre igazítás):

Ha Igaz az értéke, a betöltött kép vízszintesen és függőlegesen középre igazodik, ha Hamis, akkor kép bal felső sarka az Image bal felső sarkából indul.

Megjegyzés: a középre igazítás nem hat, ha az AutoSize vagy a Strech tulajdonság valamelyike is Igaz-ra van állítva, vagy ikon típusú fájl a használt kép.

3) Stretch (nyújtás):

Egyrészt kicsit megtévesztő a tulajdonság neve, másrészt nagyon találó. Ha az Image-be egy méreteiben kisebb képet töltünk be, és Igaz-ra állítottuk ezt a tulajdonságot, akkor a kép függőlegesen és vízszintesen is „megnyúlik” úgy, hogy pont beleférjen az Image-be. Ha viszont egy nagy képet használunk (és a Stretch Igaz), akkor az „összemegy” úgy, hogy pont beférjen. Egy betöltött kép esetén pedig, ha az Image méretét változtatjuk, a kép is változik az Image-el.

4) Proportional (arányosítás):

Ha Igaz-ra állítjuk ezt a tulajdonságot, akkor a túl nagy képek méretarányosan (tehát nem torzulva, ahogyan a Strech esetében) lesznek olyannyira kicsinyítve, hogy éppen elférjenek az Image-ben, s ha az Image méretét növeljük, a kép addig növekedhet méretarányosan, míg eredeti méretét el nem éri. Az eredetileg „elférő" képek nem módosulnak.

5) Transparent (áttetszősség):

Értéke Igaz vagy Hamis lehet, és akkor éreztetheti hatását, ha képként tömörítetlen állományt, vagyis Bitmap-et (*.bmp), vagy ikonfájlt (*.ico) használunk. Ekkor a kép háttérszíne áttetszővé válik, és láttatni engedi az alatta lévő elemeket. Háttérszín alatt értjük a képen leggyakrabban előforduló színt.

Megjegyzés: tapasztalatunk szerint nem működik, viszont fontos „hibajavító" szerepe van, melyet a használatnál ismertetünk.

6) Canvas (vászon):

Ez a tulajdonság nem érhető el az Object Inspector-ból, de a rajzoláshoz mindig ezt a tulajdonságot kell használnunk. Fontos tudnunk, hogy ez már önmagában is egy objektum, így vannak mezői, eljárásai, függvényei.

A Canvas-nek két számunkra fontos eleme van a *Pen (toll)*, valamint a *Brush (ecset)*, s ezeknek főként a színállításuk lényeges – egyelőre. Általában a Pen felel a rajzolásért, az egyszerű vonalakért, míg a Brush a kitöltésért. Színüket a következőképp módosíthatjuk:

```
Image.Canvas.Pen.Color:=szin1; //szin1 : TColor típusú változó;
Image.Canvas.Brush.Color:=szin2; //szin2 : TColor típusú változó;
```

❖ ***A képkezelő (TImage) – Használata:***

1) Pont rajzolása:

Egy pont megrajzolása nem jelent mást, mint egy pixel színének módosítását. Az Image Canvas része tartalmaz egy *Pixels* nevű tulajdonságot, mely egy szélesség*magasság nagyságú (vagyis oszlop-sorindexelésű) TColor típusú elemekből álló mátrix, ahol a (0,0) pont a bal felső sarkat jelöli, és értelemszerűen legfeljebb a (szélesség-1,magasság-1) koordináta-pár az értelmes. Így világos, hogy a pontrajzolás a következő módon történik:

```
Image.Canvas.Pixels[X,Y]:=szin;
//szin : TColor típusú változó
//X,Y : Egész; Image : az általunk használni kívánt TImage típusú
//változó neve
```

2) Szakasz rajzolása:

Ahelyett, hogy valamilyen algoritmus segítségével pontok egyenkénti kirajzolásával oldanánk meg a feladatot, szerencsénk ehhez az Image Canvas részének két eljárását használhatjuk fel hatékonyan: *MoveTo(X,Y:Integer)* és *LineTo(X,Y:Integer)*. A MoveTo segítségével a kívánt kezdőpozícióba mozgathatjuk a Canvas „tollát", majd a LineTo-val ebből a pontból rajzolhatunk vonalat a megadott koordinátákba. A szakasz színe a Pen színétől függ. Példaként lássunk egy kódrészletet (amennyiben a használt képkezelő neve Image):

```
Image.Canvas.MoveTo(0,0);
Image.Canvas.LineTo(Image.Width-1,Image.Height-1);
//amely a bal felső sarokból a jobb alsó sarokba, tehát átlósan húz
//vonalat
```

3) Téglalap rajzolása:

Téglalap rajzolásához is megtehetnénk, hogy a kívánt pontokat valamilyen algoritmus alapján kirajzolnánk az előbb ismertetett módon. Érezhető azonban, hogy ez nem túl praktikus, és szerencsénkre itt három egyszerűbb megoldás közül is választhatunk, azonban mindegyik kicsit másra szolgál. Nézzük őket sorra:

A *Rectangle(X1,Y1,X2,Y2:Integer)* eljárás segítségével egy olyan téglalapot rajzolhatunk, melynek bal felső pontja éppen az (X1,Y1), jobb alsó sarka pedig az (X2,Y2) pont, keretszínét a

Pen, a töltőszínt pedig a Brush határozza meg. Példaként lássunk egy kódrészletet (amennyiben a használt képkezelő neve Image):

```
Procedure Pelda;
Begin
   With Image.Canvas do begin
      Pen.Color:=clBlack;
      Brush.Color:=clWhite;
      Rectangle(0,0,Image.Width-1,Image.Height-1);
      end;
End;
//ezzel a kóddal fehér hátteret, és fekete keretet adhatunk a teljes
//Image-nek
```

Az eljárás használható egy másik paraméterezéssel is, ehhez azonban meg kell ismernünk a beépített téglalaptípust (a definíció, szemléltető jellegű, nem a valódi implementáció!):

```
TRect=Record(X1,Y1,X2,Y2:Integer or BF,JA:TPoint);
TPoint=Record(X,Y:Integer);
```

Vagyis kétféleképpen adhatunk meg egy ilyen típusú adatot: egyrészt egészekkel dolgozva a pontkoordinátákat egyenként, vagy a bal felső (BF) illetve jobb alsó (JA) pontokat, mint pont-típusokat megadva. Természetesen a két fajta megadási mód valójában nem különbözik – nem is különbözhet – egymástól, így ízlésünk szerint választhatunk közülük. Az előző kódrészletet a következőképp módosíthatjuk ekvivalens módon:

```
Procedure Pelda;
Begin
Var Teglalap:TRect;
Begin
   Teglalap:=Rect(0,0,Image.Width-1,Image.Height-1);
   //ez a beépített függvény egyszerűsíti az egyenkénti megadást
   //Function Rect(X1,Y1,X2,Y2):TRect;
   With Image.Canvas do begin
      Pen.Color:=clBlack;
      Brush.Color:=clWhite;
      Rectangle(Teglalap);
      end;
End;
```

A *FrameRect(teglalap:TRect)* eljárás egy kitöltetlen téglalapot tud rajzolni nekünk, ahol a színt a Brush határozza meg. Példaként lássunk egy kódrészletet (amennyiben a használt képkezelő neve Image):

```
Procedure Pelda;
Begin
Var Teglalap:TRect;
Begin
   Teglalap:=Rect(0,0,Image.Width-1,Image.Height-1);
   Image.Canvas.Brush.Color:=clRed;
   Image.Canvas.FrameRect(Teglalap);
End;
//most egy piros keretet adtunk Image-nek
```

A *FillRect(teglalap:TRect)* eljárás az előzővel pont ellentétes feladatot lát el, azaz egy kitöltött, ám keret nélküli téglalapot lehet vele alkotni, színét azonban szintén a Brush határozza meg. Azonban figyelnünk kell arra, hogy a téglalap meghatározásakor figyelembe kell venni a jobb és az alsó szélt is, viszont ezeket az eljárás nem színezi be! Példaként lássunk egy kódrészletet (amennyiben a használt képkezelő neve Image):

```
Procedure Pelda;
Begin
Var Teglalap:TRect;
Begin
   Teglalap:=Rect(0,0,Image.Width,Image.Height);
   //figyeljünk a változásra!
   Image.Canvas.Brush.Color:=clBlue;
   Image.Canvas.FillRect(Teglalap);
End;
//így a teljes Image-et kékre tudtuk színezni
```

Megjegyzés: mivel minden négyzet téglalap, könnyen látható, hogy a téglalap tulajdonságait megfelelően választva éppen négyzethez jutunk.

4) Ellipszis rajzolása:

Ellipszist rajzolni a képzeletbeli határoló téglalap segítségével tudunk, keretező színe a Pen, míg kitöltő színe a Brush beállításától függ. A rajzolást végző eljárásnak, ahogyan a normál téglalap (Rectangle) esetében is, kétféle paraméterezése lehetséges: négy egész típusú adattal, illetve egy TRect típusúval: *Ellipse(X1,Y1,X2,Y2:Integer)* vagy *Ellipse(teglalap:TRect)*. Példaként lássunk egy kódrészletet (amennyiben a használt képkezelő neve Image):

```
Procedure Pelda;
Begin
   With Image do begin
      Canvas.Pen.Color:=clBlue;
      Canvas.Brush.Color:=clRed;
      Ellipse(0,0,Width-1,Height-1);
      //ha deklarálunk egy Teglalap:TRect változót, és megfelelően
      //beállítjuk, akkor az eljárást meghívhatjuk így is:
      //Ellispse(Teglalap);
      end;
End;
```

Megjegyzés: mivel minden kör ellipszis, itt is könnyen látható, hogy ha négyzet alakúra választjuk határoló téglalapot, akkor pont kört kapunk.

5) Sokszög rajzolása:

Mivel a sokszög pontokat összekötő egyenesekből áll, így helyes az első gondolat, hogy szakaszok egymás után rajzolásával valósítsuk meg a feladatot. Azonban érezhető, hogy nem túl praktikus megoldás, és ismét „szerencsénk” van, hiszen két sokszögrajzoló eljárás is van. Ehhez azonban szükséges kicsit jobban megismernünk a téglalapnál már futólag megjelent ponttípus. Ahogyan a téglalapnak is, ennek is van beépített típusa, és található olyan függvény is, amely ilyen típusú adatot hoz létre egy koordináta-párból:

```
TPoint=Record(X,Y:Integer);
Function Point(X,Y:Integer):TPoint;
```

A kis bevezető után megismerkedhetünk a *Polygon(**Const**. points:**Array** of TPoint)* és a *Polyline(**Const**. points:**Array** of TPoint)* eljárásokkal. Előbbi a Pen színével keretezett, Brush színével kitöltött sokszöget rajzol. A rajzolás úgy történik, hogy a paraméterként kapott tömb elemeit (hiszen ők pontok) összeköti, lezárásként az utolsót az elsővel. A Polyline azonban csak az alakzat körvonalát rajzolja meg a Pen-nél beállított színnel, és nem köti össze az utolsó pontot az elsővel. Lássunk két egyszerű kódrészletet a könnyebb érthetőségért:

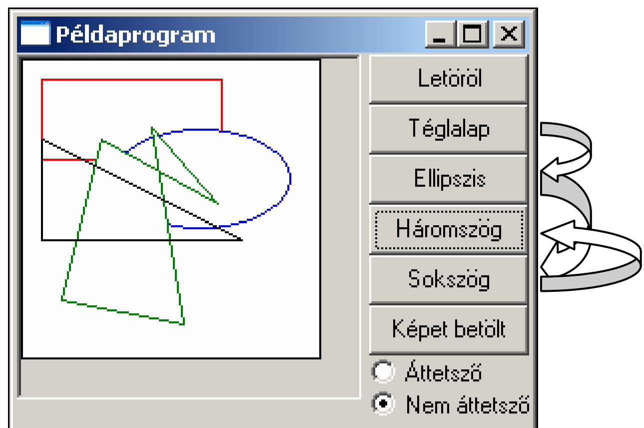
```
Procedure Pelda;
Begin
Var pontok:Array of TPoint;
Begin
   SetLength(pontok,6);
   pontok[0]:=Point(30+10,30);
   pontok[1]:=Point(30+10*Round(Cos(60)),30-10*Round(Sin(60)));
   pontok[2]:=Point(30-10*Round(Cos(60)),30-10*Round(Sin(60)));
   pontok[3]:=Point(30-10,30);
   pontok[4]:=Point(30-10*Round(Cos(60)),30+10*Round(Sin(60)));
   pontok[5]:=Point(30+10*Round(Cos(60)),30+10*Round(Sin(60)));
   With Image1.Canvas do begin
      Pen.Color:=clBlack;
      Brush.Color:=clBlue;
      Polygon(pontok);
      end;
End;
//A most bemutatott kód egy 10 pixel sugarú, (30,30) középpontú körbe
//próbál írni egy nagyjából szabályos hatszöget fekete kerettel, és
//kék belsővel. A pontokat egyszerű szögfüggvényekből kaptuk.
```

A következő példa eredménye egy szintén gyakori alakzat, a háromszög. Itt azonban nem deklarálunk előre egy pontokból álló tömböt, hanem közvetlenül adjuk meg annak elemeit, rögtön paraméterként, és most nem szeretnénk kitölteni az alakzatot, hanem csak a vonalait megrajzolni. Ezért a Polyline eljárást használjuk, de szeretnénk, hogy lezárt legyen a háromszög, így egy plusz („fiktív") pontot teszünk be, melynek koordinátái megegyeznek a kezdőpontéval:

```
Procedure Pelda;
Begin
   With Image1.Canvas do begin
      Pen.Color:=clBlack;
      Polyline([Point(10,10),Point(10,40),Point(70,40),Point(10,10)]);
      end;
End;
```

6) Kicsit összefoglalva:

Egy alkalmazásban összefoglaltuk az eddig bemutatott rajzolási módokat. A gombokra OnClick eseményeket írtunk, s ezek kódjait itt be is mutatjuk.



Elhelyeztünk egy 150*150 nagyságú Image-t. A Form OnCreate eljárását használtuk arra, hogy induláskor fehér háttérrel, és fekete kerettel jelenjen meg az Image:

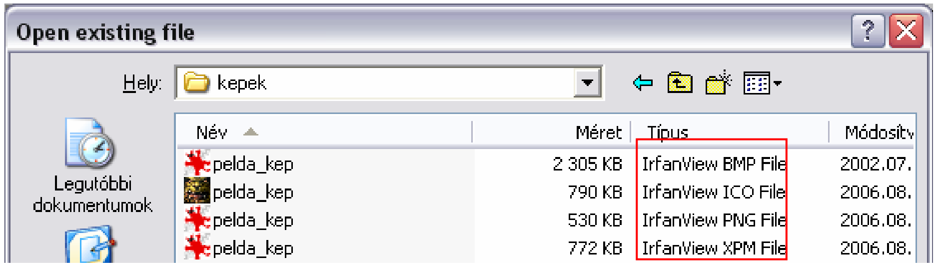
```
procedure TForm1.FormCreate(Sender: TObject);
begin
   With Image1.Canvas do begin
      Pen.Color:=clBlack;
      Brush.Color:=clWhite;
      Rectangle(0,0,149,149);
      end;
   Image1.Transparent:=False;//ennek magyarázatát lásd alább!
end;
```

A négy rajzoló gomb (Téglalap, Ellipszis, Háromszög, Sokszög) előre beállított alakzatokat rajzolhatnak erre a felületre, ha nincs betöltve kép. A sorrend tetszőleges, sőt, nem kell feltétlenül mindegyik alakzatot megrajzoltatni, az ábrára mutató nyilak csak az aktuális példát illusztrálják. A Letöröl gomb segítségével, visszaállíthatjuk az indulási állapotot (kép használata után is). Az eddigi öt gomb OnClick eseményeinek kommentezett kódja:

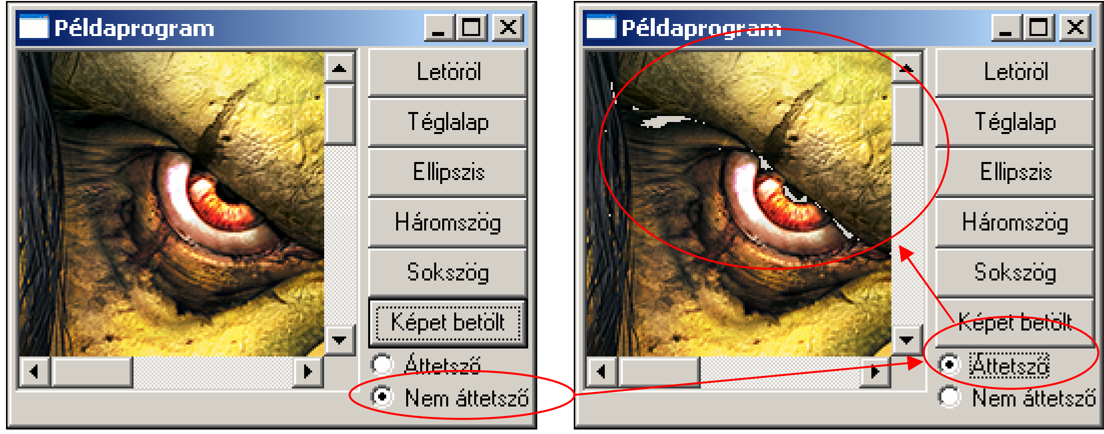
```
procedure TForm1.Button1Click(Sender: TObject);//Letöröl gomb
begin
   If Image1.Picture<>nil then begin;//ha az Image-ben van egy kép
      Image1.Picture.Clear;//azt ki kell törölnünk
      Image1.Width:=150;//majd a szélességet és a magasságot
      Image1.Height:=150;//az eredetire vissza kell állítanunk
      end;
   With Image1.Canvas do begin
      Pen.Color:=clBlack;
      Brush.Color:=clWhite;
      Rectangle(0,0,149,149);
      end;
end;
procedure TForm1.Button2Click(Sender: TObject);//Téglalap gomb
begin
   With Image1.Canvas do begin
      Pen.Color:=clRed;
      Rectangle(10,10,100,50);
      end;
end;

procedure TForm1.Button3Click(Sender: TObject);//Ellipszis gomb
begin
   With Image1.Canvas do begin
      Pen.Color:=clBlue;
      Ellipse(45,35,135,85);
      end;
end;

procedure TForm1.Button6Click(Sender: TObject);//Háromszög gomb
begin
   With Image1.Canvas do begin
      Pen.Color:=clBlack;
      Polyline([Point(10,40),Point(10,90),Point(110,90),Point(10,40)]);
      end;
end;

procedure TForm1.Button4Click(Sender: TObject);//Sokszög gomb
begin
   With Image1.Canvas do begin
      Pen.Color:=clGreen;
      Polygon([Point(20,120),Point(40,40),Point(98,72),Point(65,34),
               Point(81,132)]);
      end;
end;
```

7) Kép használata:

Használtunk egy OpenDialog-ot is (lásd később) a képek könnyebb megnyitására, és beállítottunk néhány szűrőt is annak érdekében, hogy az összes támogatott formátumú kép elérhető legyen. Így a „Képet betölt” gombot használva nyithatunk meg képeket. A gombra kattintva egy párbeszédablak jelenik meg, ahol kiválaszthatjuk használni kívánt képet.



A támogatott formátumok a képen is jól láthatók: Bitmap (*.bmp), ikonfájlok (*.ico), Portable Network Graphics (hordozható hálózati kép, *.png) és PixMap (*.xmp). Ezek közül kettővel (PixMap, PNG) gond nélkül megbirkózik a Lazarus, ám a másik kettő némi problémát jelent. Mindenáron megpróbálja őket áttetszővé tenni (Transparent), viszont ez által a legtöbbször (nem feltétlen egybefüggő felületként) előforduló szín nem fog látszani (áttetszővé válik), és csúnya lesz a kép. Sajnos valami oknál fogva ezen az sem segít, ha az Object Inpsectorban Hamis-ra állítjuk ezt a tulajdonságát az Image-ünknek, mert nem veszi figyelembe fordításkor a program. Egy megoldási javaslatunk, hogy a Form (vagy az Image-et tartalmazó komponens) OnCreate eljárásába elhelyezzük ennek kódját. Azonban, hogy látható legyen ennek fontossága két rádiógombot is tettünk a programba, melyek a Transparent tulajdonság Hamis illetve Igaz állapotai közt képesek váltani. Az alábbi két programképen látható a két állapot közti különbség:

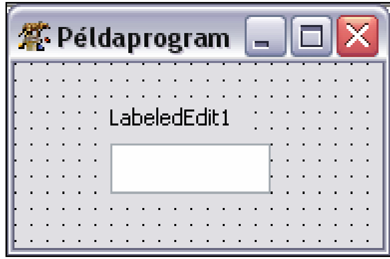


❖ A címkézett beviteli mező (TLabeledEdit) – Bevezető:

Sokszor előfordul, hogy egy beviteli mező (TEdit) szerepe nem egyértelmű, és így a felhasználó nehezen, vagy egyáltalán nem tudja használni a programunkat. Ezt elkerülhetjük azzal az aprócska, és gyakran használt „trükkel”, hogy a környezetébe elhelyezünk egy címkét (TLabel), melyen feltüntetjük a beviteli mező szerepét. Ez azonban két komponens elhelyezését jelenti, ami bonyolíthatja a programot. A kiegészítők palettája kínál erre egyszerűbb megoldást: egy olyan komponenst helyezhetünk el, mely a beviteli mező és a címke egybeolvasztásával keletkezett.

A tulajdonságok listája szinte teljes mértéken megegyezik a TEdit-nél ismertetekkel, mindössze néhány tulajdonságtól (pl. Align, Action) kellett megválnunk, és három új tulajdonsággal kell megismerkednünk, melyek a rátett címke miatt szükségesek.

A komponens elhelyezve egy üres formon:



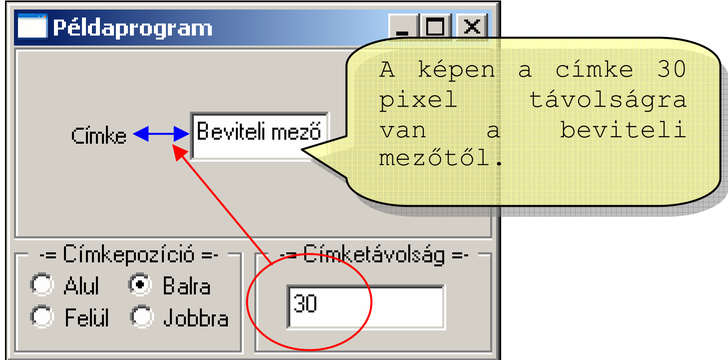
❖ A címkézett beviteli mező (TLabeledEdit) – Tulajdonságok (Properties):

1) LabelPosition (címkepozíció):

Ez határozza meg, hogy a beviteli mezőhöz képest hol helyezkedjen el a címke. Négyféle lehetőség közül választhatunk: lpAbove felülre, az lpBelow alulra, míg az lpLeft és lpRight balra, illetve jobbra teszi a címkét.

2) LabelSpacing (címketérköz):

A címke beviteli mezőtől vett távolságát határozza meg, s értéke pixelben értendő. Tehát „x” érték esetén a címke x pixel távolságra lesz a beviteli mezőtől a címkepozíciónál meghatározott irányban. A könnyebb érthetőség kedvéért, íme egy programkép:



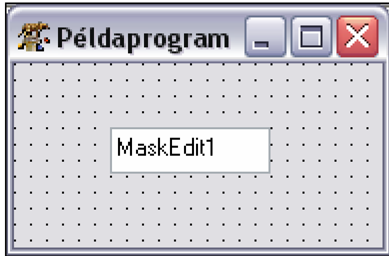
3) EditLabel (a címke):

Ez a tulajdonság maga a címke. Itt lehet módosítani annak beállításait majdnem úgy, mint a hagyományos címkénél. Ami inkább említésre méltó, hogy találunk itt eseményeket is. Ezek a címkére vonatkoznak, tehát nem a beviteli mezőre. A komponens Events részében találhatók azok az események, melyek az Edit-re vonatkoznak.

❖ A maszkolható beviteli mező (TMaskEdit) – Bevezető:

Sokszor fordul elő az is, hogy speciális formában szeretnénk megkapni a bemeneti adatokat, mint például dátum, telefonszám. Ezeket többféleképpen megadhatja a felhasználó, mely nehézkessé teszi az adatok feldolgozását. Megoldás lehet az ilyenekről említést tenni alkalmazásunk súgójában, de a most bemutatásra kerülő komponens is remek lehetőségeket nyújt. Az egyszerű beviteli mezőhöz képest a számunkra fontos tulajdonságok listája mindössze egy elemmel bővült, így használata könnyedén elsajátítható. Ennek a tulajdonságnak a használatával elérhető az, hogy a beviteli mező speciálisabban működjön, mint eddig.

A komponens elhelyezve egy üres formon:



❖ A maszkolható beviteli mező (TMaskEdit) – Tulajdonságok (Properties):

1) EditMask (a maszk):

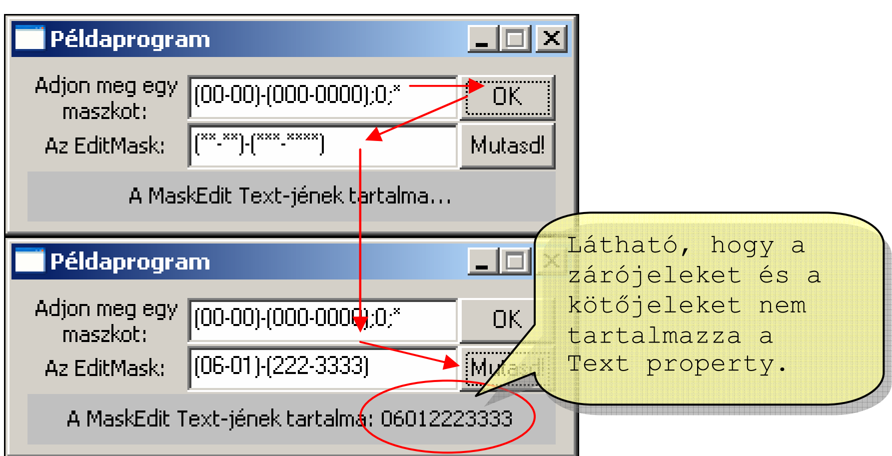
A kívánt formátum beállítására egy úgynevezett maszkot kell alkalmazni. Ez egy speciális jelentésű jelekből álló karaktersorozat, mely azt eredményezi, hogy a beviteli mező csak bizonyos – az adott jel által megengedett – karakterekre fog reagálni. A maszk három részből áll: az első rész maga a maszk, a második rész 0 vagy 1 lehet, és azt határozza meg, hogy a maszk szöveges karakterei bekerüljenek-e Text tulajdonságba vagy sem. A harmadik rész pedig a beírandó karakterek helyét mutató jel. A maszkok megadásában egy jól használható szerkesztőablak hivatott nekünk segíteni.

Nézzük a maszk speciális karaktereinek listáját, és jelentésüket:

Jel	Szerepe
>	Ha szerepel a maszkban, az őt követő karakterek nagybetűsként jelennek meg, amíg nincs vége a maszknak, vagy nem következik egy < karakter.
<	Ha szerepel a maszkban, az őt követő karakterek kisbetűsként jelennek meg, amíg nincs vége a maszknak, vagy nem következik egy > karakter.
<>	Ez után a kettős jel után semmilyen ellenőrzés nincs nagy, illetve kisbetűség szempontjából.
\	Ezt a jelet mindig egy szöveges karakter követi, mely meg is jelenik a beviteli mezőben. Segítségével a maszkolásra használt karakterek is megjeleníthetők.
L	A jel megköveteli egy ábécébeli karakter használatát az adott helyen.
l	A jel (kis L betű) megengedi, de nem követeli meg egy ábécébeli karakter

	használatát az adott helyen.
A	A jel megköveteli egy karakter (ábécébeli vagy szám) használatát az adott helyen.
a	A jel megengedi, de nem követeli meg egy karakter (ábécébeli vagy szám) használatát az adott helyen.
C	A jel megköveteli egy tetszőleges karakter használatát az adott helyen.
c	A jel megengedi, de nem követeli meg egy tetszőleges karakter használatát az adott helyen.
0	A jel megköveteli egy számkarakter használatát az adott helyen.
9	A jel megengedi, de nem követeli meg egy számkarakter használatát az adott helyen.
#	A jel megengedi, de nem követeli meg egy számkarakter vagy egy + vagy egy - jel használatát az adott helyen.
:	Ez a jel arra használható, hogy az időformátumban elválassza az órát, percet, és másodpercet. Ha a számítógép területi beállításainál más jel van megadva erre a célra, akkor automatikusan az a jel fog megjelenni a beviteli mezőben.
/	Ez a jel arra használható, hogy a dátumformátumban elválassza a hónapokat, napokat, és az éveket. Ha a számítógép területi beállításainál más jel van megadva erre a célra, akkor automatikusan az a jel fog megjelenni a beviteli mezőben.
;	Ez a jel választja szét a maszk három részét.
_	A jel automatikusan szóközt ír a szövegbe. Amikor a felhasználó ír a beviteli mezőbe, a kurzor átugorja a _ karaktert.

Mint említettük, a második rész azért felel, hogy a szöveges karakterek megjelenjenek-e a MaskEdit Text tulajdonságában. Ha 0-ra állítjuk értékét, akkor nem kerülnek be. Például a magyar 7-jegyű (plusz ország és körzethívó) telefonszámok maszkja a következő lehetne: „(00-00)-(000-0000);0;*”. Két programképen mutatjuk az eredményt: az elsőn látható, hogy miként néz ki a maszkolható beviteli mező, ha a fenti maszkot alkalmazzuk, a másodikon pedig az eredmény.



❖ ***<u>A maszkolható beviteli mező (TMaskEdit) – Használata:</u>***

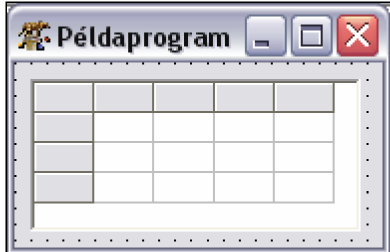
A komponens használatát ismét egy általunk készített mintaalkalmazáson keresztül mutatjuk be. A programban két segítő és egy megjelenítő célú címkét, két gombot, egy beviteli mezőt, és egy MaskEdit-et helyeztünk el. Az „OK” gombra kattintva az Edit-be írt szöveg lesz új komponensünk maszkja. Ha helyesen töltjük ki a MaskEdit-et a „Mutasd!” gombra kattintva a legalsó címkén megjelenik a MaskEdit Text tulajdonságának tartalma. Mindössze a két gombhoz írtunk OnClick eseményeket:

```
procedure TForm1.Button1Click(Sender: TObject);//OK gomb
begin
   Label3.Caption:='A MaskEdit Text-jének tartalma...';
   //a megjelenítő címkén visszállítja a kezdőtartalmat
   MaskEdit1.EditMask:=Edit1.Text;
end;

procedure TForm1.Button2Click(Sender: TObject);//Mutasd! gomb
begin
   Label3.Caption:='A MaskEdit Text-jének tartalma: '+MaskEdit1.Text;
end;
```

❖ ***A szöveges táblázat (TStringGrid) – Bevezető:***

A táblázatok, sokféle felhasználási területük miatt gyakori elemek a programokban. Ezzel a komponenssel mi is könnyen készíthetünk ilyet. Látni fogjuk, hogy tulajdonképpen használata sem túl bonyolult, a szövegeket egy kétdimenziós tömbben tárolja.

A komponens elhelyezve egy üres formon:



❖ ***A szöveges táblázat (TStringGrid) – Tulajdonságok (Properties):***

1) StringGrid Editor (táblázatszerkesztő):

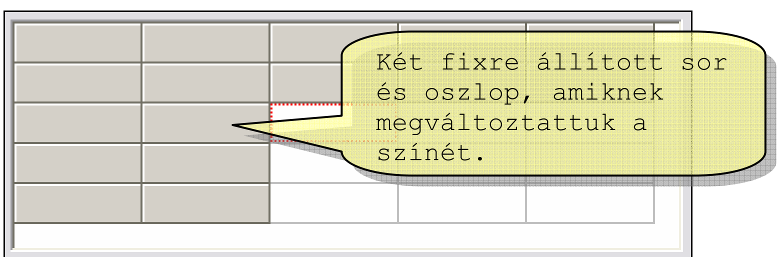
A komponensre jobb egérgombbal kattintva a felugró menüből érhető el. Használatával a program szerkesztése közben tudjuk feltölteni a meglévő táblázatunkat adatokkal.

2) RowCount (sorok száma) és ColCount (oszlopok száma):

Ez a két tulajdonság határozza meg a StringGrid oszlopainak és sorainak számát.

3) FixedRows (rögzített sorok), FixedCols (rögzített oszlopok) és FixedColor (rögzítettek színe):

Rögzítettnek nevezzük az olyan sorokat, és oszlopokat, melyek celláit a felhasználó akkor sem módosíthatja, ha Options-ben (lásd kicsit később) ezt beállítjuk. Ezek számát állíthatjuk be e két tulajdonsággal, s színüket a FixedColor-nél változtathatjuk meg.



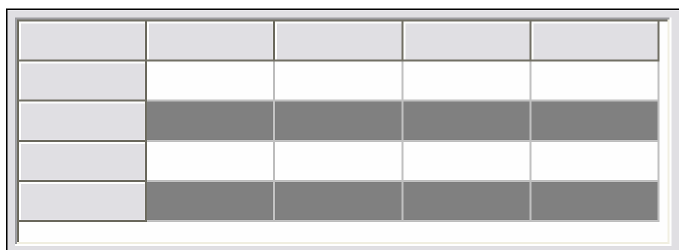
4) Options (beállítások):

Itt különböző logikai kapcsolókat találunk, melyekkel jobban testre szabható a táblázatunk.

Mező	Jelentése
goFixedVertLine	A fixre állított, függőleges cellák közti elválasztó vonalakat kapcsolhatjuk be (Igaz), vagy ki (Hamis).
goFixedHorzLine	A fixre állított, vízszintes cellák közti elválasztó vonalakat kapcsolhatjuk be (Igaz), vagy ki (Hamis).
goVertLine	A nem fix beállítású, függőleges cellák közti elválasztó vonalakat kapcsolhatjuk be (Igaz), vagy ki (Hamis).
goHorzLine	A nem fix beállítású, vízszintes cellák közti elválasztó vonalakat kapcsolhatjuk be (Igaz), vagy ki (Hamis).
goRangeSelect	Igaz értéken a táblázatban tartományokat jelölhetünk ki, Hamis értéken mindig csak egy elem kiválasztása lehetséges.
goDrawFocusSelected	Ha Igaz-ra állítjuk ezt a mezőt, akkor a cella egyszerű kijelölésekor speciális háttérszínt kap, nem csak kerettel. A háttér színét az aktuális Windows-os színbeállítás határozza meg.
goRowSizing	Engedélyezhetjük (Igaz), vagy letilthatjuk (Hamis) a sorok átméretezhetőségét.
goColSizing	Engedélyezhetjük (Igaz), vagy letilthatjuk (Hamis) az oszlopok átméretezhetőségét.
goRowMoving	Engedélyezhetjük (Igaz), vagy letilthatjuk (Hamis) a sorok áthelyezhetőségét.
goColMoving	Engedélyezhetjük (Igaz), vagy letilthatjuk (Hamis) az oszlopok áthelyezhetőségét.
goEditing	Alapállapotban a felhasználó nem módosíthat a cellák tartalmán. Ezt engedélyezhetjük itt, ha Igaz-ra állítjuk.
goTabs	Ha Igaz-ra állítjuk értékét, akkor a felhasználó a „Tab” használatával is válthat cellát.
goRowSelect	Igaz értéken mindig egy teljes sor van kijelölve.
goAlwaysShowEditor	Ha Igaz-ra állítjuk értékét, az első cella-szerkesztés után mindig látható a kurzor egészen addig, míg a StringGrid-en kívülre nem kattint a felhasználó.
goThumbTracking	Igaz-ra állítva, a táblázat gördítéskor is frissül a tartalom, míg Hamis értéken csak a gördítő felengedésekor aktualizálódik.
goDblClickAutoSize	Igaz értékűre állítva hatását akkor fejti ki, ha van egy olyan oszlop, melynek van egy kilógó tartalmú cellája. Ekkor a fejlécen, az oszlop jobb oldali elválasztójára duplán kattintva az oszlop szélessége a kilógó elem nagyságához igazodik (ha be van kapcsolva az oszlopok méretezhetősége).
goFixedRowNumbering	Ha Igaz-ra állítjuk, számozott lesz a fix-re állított első oszlop.

5) AlternateColor (váltószín):

Ezzel a tulajdonsággal beállíthatjuk a nem fix (lásd később) típusú, páros sorszámú sorok színét.



6) AutoAdvance (automatikus haladás):

Ha „Tab”-bal vált cellát a felhasználó, vagy a cellamódosítást követően „Enter”-rel halad tovább, akkor az itt beállított irány szerint mozdul a kijelölés.

Érték	Jelentése
aaDown	A kijelölés lefelé halad a legalsó sorig.
aaLeft	A kijelölés balra halad a bal szélen ki választható utolsó oszlopig.
aaLeftDown	A kijelölés balra halad a bal szélen ki választható utolsó oszlopig, utána sort ugrik, ha van még sor, s annak a jobb oldali végénél kezd.
aaRight	A kijelölés jobbra halad a jobb szélen ki választható utolsó oszlopig.
aaRightDown	A kijelölés jobbra halad a jobb szélen ki választható utolsó oszlopig, utána sort ugrik, ha van még sor, s annak a bal oldali végénél kezd.
aaNone	A kijelölés nem mozdul.

7) AutoFillColumns (automatikus oszlopkitöltés):

Ha Igaz-ra állítjuk ezt a tulajdonságot, akkor az oszlopok olyan szélességet vesznek fel, hogy kitöltsék a rendelkezésre álló helyet.

8) DefaultColWidth (oszlopok szélessége) és DefaultRowHeight (sorok magassága):

Pixelben adhatjuk meg az oszlopok szélességének, illetve a sorok magasságának alapértékeit. Minden új sor és oszlop ilyen paraméterekkel keletkezik.

9) Flat (laposság):

A rögzített cellákra vonatkozik ez a tulajdonság. Alapértelmezésben ezek kiemelkednek a síkból. Ezt módosíthatjuk, ha ezt a kapcsolót Igaz-ra állítjuk.

10) ScrollBars (gördítők):

Ahogy a Memo esetében láthattuk, beállíthatunk vízszintes, illetve függőleges görgethetőséget. A beállítási lehetőségek megegyeznek a Memo-nál leírtakkal.

11) TitleFont (rögzítettek betűtípusa):

A rögzített cellák szövegét formázhatjuk itt.

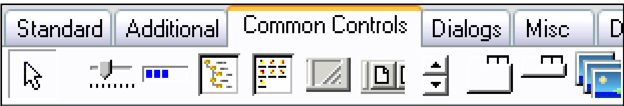
❖ *A szöveges táblázat (TStringGrid) – Használata:*

Mint említettük, a komponens cellái egy kétdimenziós, String típusú adatokból álló tömböt, vagyis mátrixot alkotnak: ezt testesíti meg a Cells belső mező. A mátrix oszlop-sor folytonosan indexelődik, a sorszámozást 0-tól kezdve. Így tudjuk egy cella tartalmát lekérdezni, módosítani. Példa:

```
StringGrid1.Cells[oszlop,sor]:='szöveg';
//0<=oszlop<=oszlopszám-1 [vagyis StringGrid1.ColCount-1]
//0<=sor<=sorszám-1 [vagyis StringGrid1.RowCount-1]
```

A Common Controls (gyakori vezérlők)

Ahogy az Additional-ben lévő komponensekkel, úgy az itt található vezérlőkkel is színesíthetjük, érdekesebbé és kezelhetőbbé tehetjük alkalmazásainkat. Itt viszont nem elsősorban esztétikai célokra gondolunk. Ezen elemeknek főként funkcionális szerepük van: a felhasználót segíthetik az eligazodásban, tájékoztathatják a program belső folyamatairól, vagy könnyíthetik az alkalmazás irányítását. Tehát e komponensek használatával programunkat professzionálisabbá tehetjük. Például, mi is szívesebben használunk olyan vírusirtót, ami valamilyen ízléses formában tudatja velünk, hogy éppen hol tart a vizsgálódásban. A paletta képe:



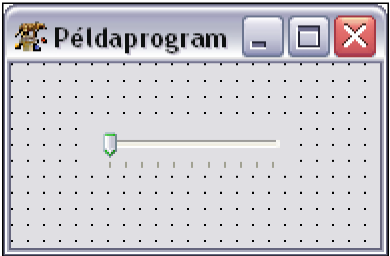
A vezérlők közül hetet emelünk ki ebben a fejezetben: ismertetjük a már többször említett ImageList-et, bemutatunk egy olyan komponenst, mellyel tagoltabb programablakokat alakíthatunk ki, s megismertetünk az említett tájékoztató és irányító szerepű vezérlőkkel is.

- Csúszka
- Folyamatsáv
- Fastruktúra
- Állapotsáv
- Fel-le gomb
- Többlapos form
- Képlista

❖ ***A csúszka (TTrackBar) – Bevezető:***

Olykor hasznos lehet, ha a felhasználónak nem kézzel kell megadnia adatot, hanem például, az egér mozgatásával tudja változtatni azt. Például fizikai programokban bizonyos paraméterek (sebesség, különböző erők nagysága, stb.). Erre nyújthat megoldást ez az egyszerű komponens. Egy jelölőt mozgathatunk egy egyenesen, s így egy intervallumon belül állíthatunk be értékeket. Természetesen mi adhatjuk meg, hogy mi legyen a kezdő, és a végérték, s azt is lekérdezhetjük, hogy a felhasználó hova állította be éppen a jelölőt. Ugyan a komponensen helyezhetünk el jelöléseket, hasznos lehet valamilyen egyéb módon is tudatni a felhasználóval, milyen értéket állított éppen be (pl. label).

A komponens elhelyezve egy üres formon:



❖ ***A csúszka (TTrackBar) – Tulajdonságok (Properties):***

1) Frequency (frekvencia):

Ezzel beállíthatjuk, hogy hány lépésenként tegyen jelölő vonalkát. Például minden másodikhoz.

2) LineSize és PageSize (lépésközök):

Itt beállíthatjuk, hogy hány lépést tegyen meg a csúszkán a jelölő, ha a kurzormozgató billentyűket, illetve ha a PgUp vagy PgDown billentyűket használjuk. A kurzormozgatókkal tett lépésközt a LineSize határozza meg, míg a másik két billentyűét a PageSize.

3) Max és Min:

A csúszka intervallumának maximális, és minimális értékeit állíthatjuk be ezekkel a tulajdonságokkal.

4) Orientation (irány):

Beállíthatjuk, hogy függőleges (trVertical), vagy vízszintes (trHorizontal) irányú legyen a csúszkánk.

5) Position (jelölő-pozíció):

A jelölő aktuális pozícióját meghatározó mező. Segítségével módosíthatjuk, vagy lekérdezhetjük a jelölő helyét.

6) TickMarks (jelölések elhelyezkedése):

A jelölések jelölőhöz viszonyított elhelyezkedését állíthatjuk itt be. A tmBoth mindkét oldalra, a tmBottomRight vízszintes elhelyezkedésnél alul, függőlegesnél jobbra, míg a tmTopLeft vízszintes elhelyezkedésnél felül, függőlegesnél balra teszi a vesszőket.

Megjegyzés: nagy kár, hogy nem működik.

7) TickStyle (jelölések típusa):

A TrackBar vonalkáinak elhelyezési módját állíthatjuk itt be. A tsAuto beállításnál a Frequency alapján automatikusan megjelennek a jelölő vonalkák. A tsManual csak a kezdő, és a befejező állapothoz tesz jelöléseket, ha többet szeretnénk, azt magunknak kell kitenni a *SetTick* eljárással, mely a komponens része. A tsNone beállításnál nincsenek jelölések. Példa:

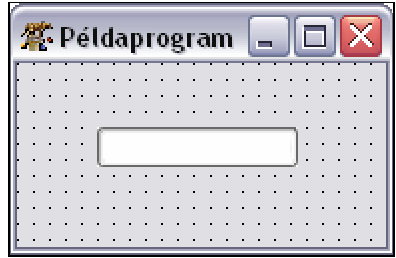
```
procedure TForm1.Button1Click(Sender: TObject);
begin
   TrackBar1.SetTick(x);//x egy intervallumbeli egész
end;
```

Megjegyzés: szomorúan, de itt is azt kell mondanunk, hogy nem működik.

❖ ***A folyamatsáv (TProgressBar) – Bevezető:***

Elképzelhető, hogy alkalmazásunk lassú folyamatokat tartalmaz, s a felhasználó azt gondolhatja, valami nincs rendben. Ilyenkor hasznos lehet annak a visszajelzése, hogy hol tart a program az adatok feldolgozásában. Ezzel a komponenssel ezt egyszerűen megvalósíthatjuk. Sajnos sok tulajdonság még nem működik, így pár hasznosságról le kell mondanunk.

A komponens elhelyezve egy üres formon:



❖ *A folyamatsáv (TProgressBar) – Tulajdonságok (Properties):*

1) BarShowText (felirat):

Ezzel a logikai tulajdonsággal lehetne beállítani, hogy a folyamatsávon jelenjen meg felirat. Két feliratra utaló tulajdonságot is találunk kódszerkesztőben (Caption, Text), de sajnos egyik értéke sem válik láthatóvá, azaz ez sem működik.

2) BorderWidth (margóvastagság):

A kitöltő oszlopok (vagy sáv), és a keret közti távolságot lehet itt beállítani, s az érték pixelben értendő.

3) Max, Min, Orientation (irány) és Position (helyzet):

Ezt a négy tulajdonságot azért nem részletezzük, mert szerepük is és beállításuk is megegyezik a TrackBar-nál leírtakkal.

4) Smooth (simítás):

A folyamatsáv megjelenítését tudjuk módosítani ezzel a tulajdonsággal. Ha Hamis, akkor oszlopokat (alsó folyamatsáv), ha Igaz, akkor egyenletes sávot (felső folyamatsáv) látunk visszajelzésül.

5) Step (lépés):

Beállíthatjuk, hogy egy lépés hány egységből álljon. Egy lépés a belső *StepIt* eljárással valósítható meg (Például: ProgressBar1.StepIt).

❖ *A fastruktúra (TTreeView) – Bevezető:*

Ezzel a komponenssel faszerkezeteket tudunk ábrázolni. Az Object Inspector-ban is ilyet látunk, hogy követhessük a komponensek egymásra épülését. A faszerkezet azt jelenti, hogy kiindulunk egy csomópontból, amely elágazhat újabb csomópontokat hozva létre, azok megint elágazhatnak, és így tovább. A „végső", nem elágazó elemeket levélelemeknek hívjuk.
A komponens elhelyezve egy üres formon:

❖ *A fastruktúra (TTreeView) – Tulajdonságok (Properties):*

1) AutoExpand (automatikus kibontás):

Ha Igazra állítjuk, akkor a fa éppen kiválasztott eleme kinyílik, a többi mind összecsukódik.

Megjegyzés: sajnos Lazarus-ban ez még nem működik.

2) BackgroundColor (háttérszín):

Segítségével módosíthatjuk a komponens hátterének színét.

3) BorderStyle (szegélystílusa) és BorderWidth (szegély szélessége):

A fastruktúrát szegélyező külső keret meglétét (bsSingle) illetve elhagyását (bsNone) állíthatjuk be a BorderStyle tulajdonságnál. A BorderWidth-tel a belső keret nagyságát adhatjuk meg, s értéke pixelben értendő.

4) ExpandSignColor (nyitó jelek színe) és ExpandSignType (nyitójel típusa):

Az elágazások előtti jelek színét és típusát (tvestArrow – nyilak, tvestPlusMinus – plusz-mínusz) állíthatjuk be.

5) HideSelection (kiválasztás elrejtése):

Ha a kapcsolót Igen-re állítjuk, és egy másik komponensre váltunk, akkor az utoljára kiválasztott csomópontot kijelölése elrejtődik. Ha pedig Hamis-ra állítjuk, akkor a kijelölés ott marad.

Megjegyzés: sajnos nem működik megfelelően, mert mindig megmarad a kijelölés.

6) HotTrack (forrónyom):

Amikor egy csomópont fölé visszük az egerünket, s ezt a tulajdonságot Igaz értékűre állítottuk akkor, mint egy linkként aláhúzza, és kékre színezi a feliratot a program.

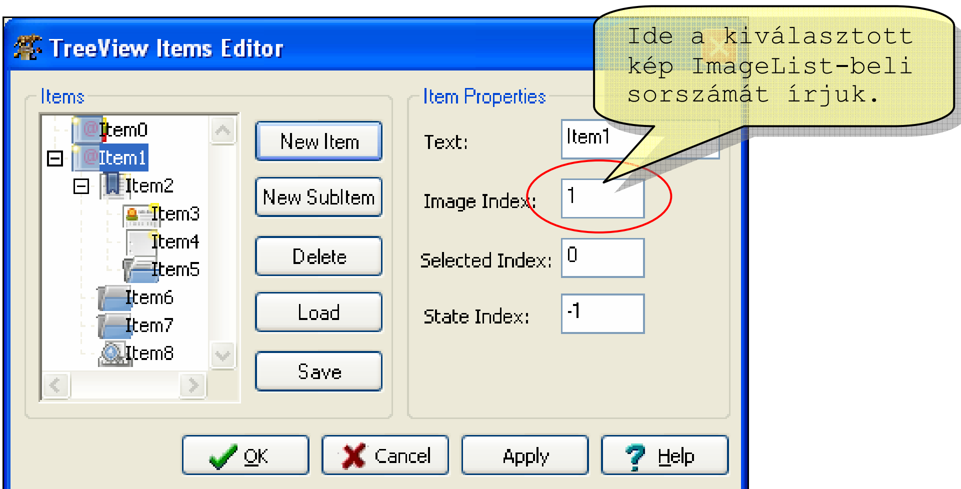
Megjegyzés: tapasztalataink szerint szintén nem működik Lazarus-ban.

7) Items (elemek):

Elérhető itt egy ablak (TreeView Items Editor), mellyel szerkesztés közben tudunk egyszerűen felvenni elemeket a fába, módosítani azok tulajdonságait. Ezt a használati részben részleteztük.

8) Images (képek):

Hasonlóan a menüelemekhez, itt is helyezhetünk a feliratok elé képeket. Ehhez szükségünk van egy ImageList-re (lásd később), és mindegyik csomópontnak meg kell adni, hogy melyik kép tartozik hozzá.



9) Indent (bekezdés):

Megadhatjuk, hogy mennyivel (hány pixellel) kezdődjenek beljebb (a saját szülőjüktől) az egyes gyerek-elemek.

10) ReadOnly (csak olvasható):

Ha Hamis, akkor a meglévő csomópontok szövegét közvetlenül át tudjuk írni úgy, hogy kijelöljük, majd még egyszer rákattintunk, és beírjuk azt, amire módosítani szeretnénk. Ha Igaz, akkor ezt nem írhatjuk át közvetlenül a csomópontokat, csak kódból történő módosítás lehetséges.

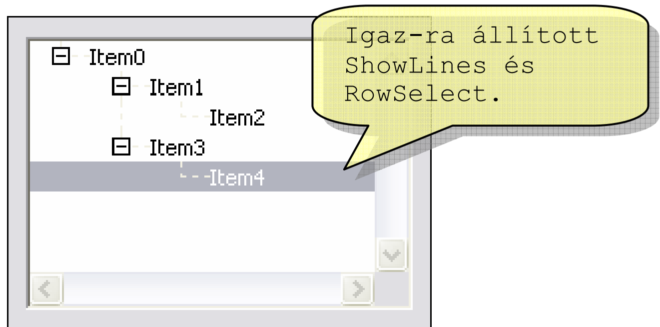
Megjegyzés: sajnos ez sem működik.

11) RightClickSelect (jobb-egérgombbal kiválaszthatóság):

Beállíthatjuk, hogy a csomópontokat jobb egérgombbal is ki lehessen választani, vagy ne.

12) RowSelect (sorkiválasztás) és ShowLines (látható vonalak):

Ha a ShowLines Igaz, akkor a fában látjuk az összekötő vonalakat a csomópontok között, Hamis értéknél nem. Ha a RowSelect Igaz, akkor nem csak magát az elemet színezi ki a program, hanem az egész sorát, ha Hamis, akkor csak az elemet.



13) ScrollBars (gördítők):

Ahogy a Memo esetében láthattuk, beállíthatunk vízszintes, illetve függőleges görgethetőséget. A beállítási lehetőségek megegyeznek a Memo-nál leírtakkal.

14) SelectionColor (kijelölési szín):

Kiválaszthatjuk, hogy milyen színnel szeretnénk látni a kijelölést.

15) ShowButtons (látható gombok):

Ezzel szabályozhatjuk, hogy akarunk-e nyitójeleket látni, vagy nem. Természetesen nyitójelek nélkül is ki tudjuk nyitni, és be tudjuk zárni a fa ágait dupla egérkattintással.

16) SortType (rendezési fajták):

Lehetséges a fában az elemek rendezése. Ha már egyszer rendeztünk, akkor azt nem vonhatjuk már vissza! A lehetséges értékek közül kettőt említünk:

Érték	Jelentése
stNone	Nincs rendezés
stText	Ha megváltozik egy elem szövege, vagy a rendezés fajtája, újrarendezi az elemeket.

17) TreeLineColor (az összekötő vonalak színe):

A hierarchiában egymáshoz kapcsolódást jelölő szaggatott vonalak színét választhatjuk meg itt.

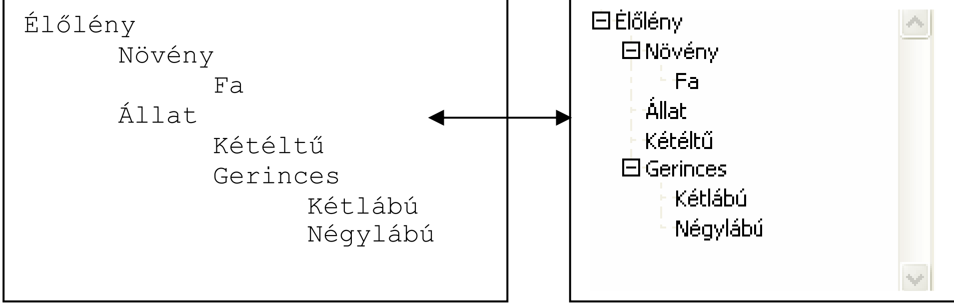
❖ ***A fastruktúra (TTreeView) – Használata:***

1) Node (csomópont):

Az egyik legfontosabb fogalom, amivel meg kell ismerkednünk, az a Node (csomópont, típusa TTreeNode). Ezek speciális mutatók, minden elemnek a fában vannak ilyen Node-jai, ezeknek a segítségével tudjuk bővíteni a fánkat. Pongyolán így tudunk elemek után, illetve elemek alá beszúrni újabb elemeket. Egy Node tartalmazza ezen kívül a csomópont szövegét is. Ha tehát egy Node szövegét szeretnénk lekérdezni vagy módosítani, azt a Node.Text-tel tehetjük meg.

2) Beolvasás fájlból:

Fákat nagyon könnyű fájlból beolvasni, elég hozzá egy olyan szöveges fájl, ami tartalmazza az egyes faelemek szövegeit. Ha azt szeretnénk, hogy valaminek a gyereke legyen, azt új sorba, egy „Tab”-bal beljebb írunk, például így:



3) Elemek hozzáadása a fához:

Az első elemet a nil értékű mutatóhoz kell hozzáfűzni! Ha az *Add(SiblingNode:TTreeNode; s:String):TTreeNode* (SiblingNode jelentése testvér) függvényt használjuk, akkor a „testvér”-ben megadott elemmel egy szintre, és „s” szöveggel kerül a fába az új elem. Az ilyen szintű elemek közül az utolsó helyre fog kerülni, hacsak nincs bekapcsolva a Text-szerinti SortType. Példa:

```
TreeView1.Items.Add(Node, ’Elem szövege’);
//ahol Node egy fa-beli elem, tehát TTreeNode típusú, vagy nil.
```

Ha azt szeretnénk, hogy közvetlen egy elem elé szúrjunk be (ugyanarra a szintre, amelyiken ő van), akkor használjuk az *Insert(NextNode:TTreeNode; s:String):TTreeNode* függvényt (NextNode jelentése következő). Példa:

```
TreeView1.Items.Insert(Node,’Node feletti elem szövege’);
//ahol Node az a fa-beli, TTreeNode típusú elem, aki elé be
//szeretnénk szúrni.
```

Ha valamihez a hierarchiában alatta levőt akarunk rendelni, az *AddChild(ParentNode:TTReeNode; s:String):TTreeNode* (ParentNode jelentése szülő) függvény használandó. Miután már adtunk „gyereket” egy elemnek, erre a szintre már nem kell tovább az AddChild, itt ugyanúgy használható az Add, és az Insert is. Példa:

```
TreeView1.Items.AddChild(Node, ’Node leszármazottjának szövege’);
//ahol Node az a fa-beli, TTreeNode típusú elem, akinek
//„leszármazottat” szeretnénk.
```

Összefoglalásképp, íme egy kódrészlet és az eredménye. Egy teljesen üres TreeView-ba szúrunk elemeket, melyre egy gomb OnClick eseményét használtuk:

```
procedure TForm1.Button1Click(Sender: TObject);
var  Node, Node2, Node3: TTreeNode;
begin
  Node:=TreeView1.Items.Add(nil, 'Első Szint');
  Node2:=TreeView1.Items.Add(Node, 'Alatta kellene legyen');
  Node3:=TreeView1.Items.Add(nil, 'Hova kerül');
  TreeView1.Items.AddChild(Node, 'Második szint');
  Treeview1.Items.AddChild(Node, '???');
  Treeview1.Items.Add(Node, 'A végére kell kerüljön');
  TreeView1.Items.Insert(Node2,'MM');
end;
```

Első Szint
Második szint
???
MM
Alatta kellene legyen
Hova kerül
A végére kell kerüljön

4) Elem törlése:

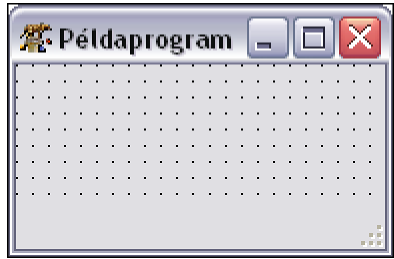
Szinte magától értetődően lehetőségünk van elemek törlésére is, amire a *Delete(Node:TTreeNode)* eljárás ad lehetőséget. A „Node” paraméter annak az elemnek a Node-ja, amit ki szeretnénk törölni. De vigyázzunk ezzel, mert ennek eredményeképpen az összes gyereke elvész annak, akit kitöröltünk!

```
TreeView1.Items.Delete(Node);
```

❖ ***Az állapotsáv (TStatusBar) – Bevezető:***

Egy nagyon hasznos komponenssel ismerkedhetünk meg, melynek a segítségével az ablak alsó részére a program állapotával kapcsolatos információkat írathatunk ki. Egy szövegszerkesztőben például, kiírhatjuk a felhasználónak, hogy hányadik oldalnál tart, hányadik sornál, hányadik karakternél.

A komponens elhelyezve egy üres formon:



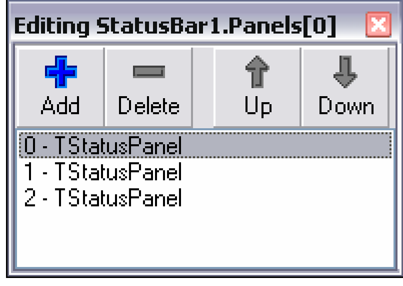
❖ ***Az állapotsáv (TStatusBar) – Tulajdonságok (Properties):***

1) Panels (panelek):

Az állapotsáv egyetlen sorból áll ugyan, viszont azt feloszthatjuk panelekre, az olvashatóság, és a könnyebb kezelhetőség érdekében. A panelekre, mint egy tömb elemeire hivatkozhatunk, tehát például, az első panel szövegét ezzel a paranccsal állíthatjuk be:

```
StatusBar1.Panels[0].Text:='Szöveg';
```

Paneleket egy szerkesztő segítségével egyszerűen létre tudunk hozni, törölni, és sorrendjüket is könnyedén módosíthatjuk. Innen, ha az Object Inspector-ra váltunk tudjuk változtatni az egyes panelek tulajdonságait.



❖ ***Az állapotsáv (TStatusBar) – A panelek tulajdonságai:***

1) Alignment (elrendezés):

A szöveg a rendelkezésére álló helyet kitöltheti balra (taLeftJustify), jobbra(taRightJustify), ill. középre(taCenter) igazítva is.

2) Bevel (kiemelkedés):

Mivel az állapotsor megjelenése 3D szerű, játszhatunk a megjelenéssel, hogy az alapszinthez képest az elemek süllyesztve (pbLowered), azzal egyszintben (pbNone), esetleg kiemelve (pbRaised) jelenjenek meg.

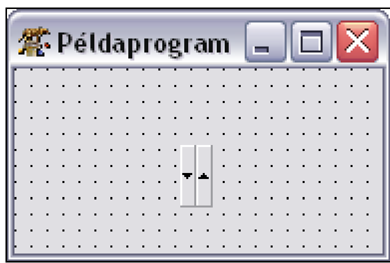
3) SimpleText (sima szöveg) és SimplePanel (egyszerű panel):

Ha a SimplePanel kapcsolót Igaz-ra állítjuk, csak egy panelt használhatunk, aminek a feliratát a SimpleText mező tartalmazza. Ha több panelre szeretnénk felosztani a StatusBar-t, akkor ezt a kapcsolót Hamis-ra kell állítanunk.

❖ ***A fel-le gombcsoport (TUpDown) – Bevezető:***

Ezt a komponenst értékek növelésére és csökkentésére használhatjuk, oly módon, hogy hozzárendeljük egy másik komponenshez. Ily módon például olyan számtartalmú editbox-ot készíthetünk, melynek értékét a gombokra kattintva is módosíthatjuk.

A komponens elhelyezve egy üres formon:



❖ ***A fel-le gombcsoport (TUpDown) – Tulajdonságok (Properties):***

1) AlignButton (gombok elhelyezkedése):

Meghatározhatjuk, hol helyezkedjenek el a gombok, a hozzárendelt komponenstől jobbra (udRight), balra (udLeft), alatta (udBottom), vagy esetleg felette (udTop).

2) ArrowKeys (nyílbillentyűk):

Igaz állapot esetén, a hozzárendelt komponens értékének megváltoztatása nem csak a gombok rákattintásával, hanem a fel-le billentyűk használatával is elérhető.

3) Associate (hozzárendelés):

Itt rendelhetjük hozzá a komponenst a gombcsoporthoz. A legördülő menüből kiválaszthatjuk, amit szeretnénk.

4) Position (helyzet):

Ezzel a tulajdonsággal lekérdezhetjük, és beállíthatjuk azt az értéket, amin a komponensben mutatott érték kell, hogy álljon. Sajnos ennek típusa Smallint, mely a hagyományos Pascal-ból ismert Integer (-32768..32767) intervallumában veheti fel értékeit.

5) Increment (lépték):

A növelés, és csökkentés mértéke állítható be itt.

6) Min (minimális) és Max (maximális értékek):

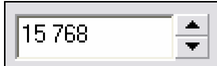
Ezen tulajdonságokkal állítható be az UpDown intervallumának határa.

7) Orientation (irány):

Ha nem föl-le gombokra (udVertical) van szükségünk, beállíthatjuk, hogy jobbra-balra (udHorizontal) mutassanak.

8) Thousands („ezresek”):

A könnyebb olvashatóság érdekében állítsuk ezt a kapcsolót Igaz-ra, és így a számjegyeket hármasával fogja csoportosítani a program. Az alábbi programképen látható erre egy példa:



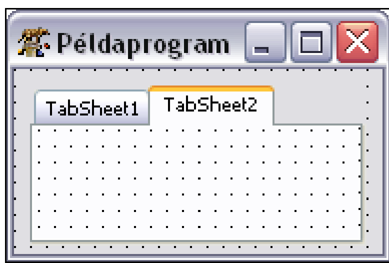
9) Wrap (ciklikusság):

Ha Igaz-ra állítjuk ezt a kapcsolót, és elérünk a maximum értékhez, a program a növelést a minimum értéktől kezdi ismét. Ha pedig a minimális értéket érjük el, akkor a maximálist kezdi csökkenteni.

❖ A többlapos form (TPageControl) – Bevezető:

A praktikus helykihasználásra ötletes megoldás lehet a TPageControl vezérlő. A komponensnek két része van: egy fülek (Tab) rész, és minden fülhöz egy olyan terület (TabSheets), ahova komponenseket helyezhetünk el. Ezek között a területek között tudunk majd fülekkel kalandozni hasonlóan, mint egy jegyzetfüzet lapjai között.

A komponens elhelyezve egy üres formon:



❖ A többlapos form (TPageControl) – Tulajdonságok (Properties) – Tabs (Fülek):

1) ActivePage (aktív oldal):

A lapok között válthatunk ezzel a mezővel, mindig az aktívat látjuk, azon dolgozhatunk felhasználóként, vagy fejlesztőként. A legördülő menüből ki tudjuk választani, melyik oldalon szeretnénk dolgozni éppen.

2) Images (képek):

A füleknek is feldobhatjuk a hangulatát kis képek segítségével. Itt is használnunk kell egy ImageList-et, ami a képeinket, s azok sorszámait fogja tartalmazni. Ha itt a legördülő menüből kiválasztjuk ezt az ImageList-et, akkor mindegyik laphoz hozzárendelhetjük annak a képnek az indexét, amelyiket a címke felirata mellé szeretnénk elhelyezni. Ezt az egyes lapoknál tehetjük meg (lásd kicsit később).

Megjegyzés: sajnos egyelőre ennek használatáról is le kell mondanunk.

3) TabIndex (címkesorszám):

Az éppen aktív címke sorszáma. Vagyis szerepe ugyan az, mint az ActivePage tulajdonságnak, csakhogy itt a lapok nevei helyett sorszámaikkal dolgozhatunk.

4) TabPosition (címkék elhelyezkedés):

Ha nem felül (tpTop) szeretnénk látni a füleket, hanem lent (tpBottom), jobb oldalt (tpRight), vagy esetleg balra (tpLeft), azt itt lehet beállítani.

Megjegyzés: a Lazarus-nak ez a verziója nem képes kezelni az utolsó három lehetőséget.

❖ A többlapos form (TPageControl) – Tulajdonságok (Properties) – TabSheets (Lapok):

1) ImageIndex (képsorszám):

Itt állíthatjuk be, hogy a kiválasztott kép-listából melyik kép tartozzon a fülhöz. Sajnos, ahogy írtuk korábban, ez még nem elérhető.

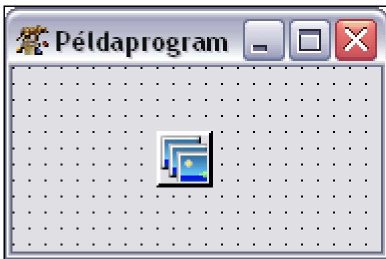
2) TabVisible (lap láthatósága):

Ha el szeretnénk tüntetni egy lapot, állítsuk ezt a kapcsolót Hamis-ra.

❖ A képlista (TImageList) – Bevezető:

Eddig is lépten-nyomon felhasználtuk ezt a komponenst, ha arról volt szó, hogy több képet kellett kezelnünk (például, amikor menünket akartuk kis képekkel feldobni). Ez nem egy vizuális komponens, nincs látható felülete, csak arra való, hogy képeinket megnyissunk, s azokat, mint egy tömb elemeit tároljuk benne.

A komponens elhelyezve egy üres formon:

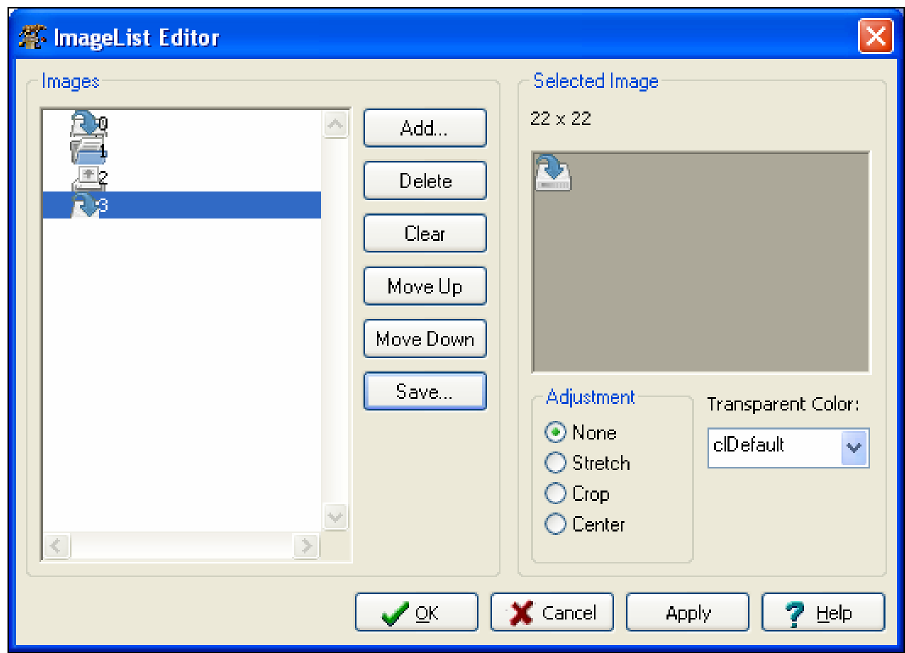


❖ A képlista (TImageList) – Tulajdonságok (Properties):

1) ImageList Editor:

A formon lévő ImageList-re jobbgombbal kattintva hívható elő. Képeinket az Add gombot használva tudjuk kiválasztani, a Delete-el törölhetünk egy elemet, vagy akár az egész listát a Clear-el. A MoveUp és a MoveDown gombokkal mozgathatunk elemeket, s a képeket el is menthetjük. A képek jobb alsó sarkában ott találjuk az indexeiket, amivel később majd hivatkozunk rájuk. Ha például egy menüben fel szeretnénk használni a második képet, akkor előbb a menühöz hozzárendeljük a használni kívánt ImageList-et, és a menüelem ImageIndex tulajdonságában a „-1”-t átírjuk kettesre.

A SelectedImage részen a kép tulajdonságait látjuk: az eredeti felbontást, s magát a képet. Mivel e képeket olyan helyeken használjuk, ahol nem szépek a nagy képek, ezért ajánljuk, hogy a felbontást 16*16-ra állítsuk át. Erre az Adjustment rész három lehetőséget kínál fel: a Strech lekicsinyíti a képet, Crop a bal felső részéből, a Center pedig a közepéből vág ki egy 16*16-os darabot. A TransparentColor-ral pedig a kép háttérszínét választhatjuk ki.

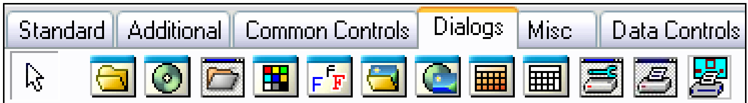


2) Height (magasság) és Width (szélesség):

Ezzel szabályozhatjuk a képek maximális magasságát és szélességét. Ez azt jelenti, hogy ha nagyobbat olvasunk be, lekicsinyíti az általunk megadott méretre.

A Dialog-ok (párbeszédablakok)

Egy olyan részhez éreztünk, mely szintén nagyon fontos szerepet tölt be alkalmazásainkban. Ezen a palettán különböző párbeszédablakokat kezelő komponensek találhatók. Egyesek az operációs rendszer megfelelő ablakait kezelik, mások a Lazarus-ban előre megírtakat használják fel. Közös tulajdonságuk, hogy a nem látható komponensek közé tartoznak, tehát az ábrázoló ikont bárhová helyezhetjük egy formon. Egy ilyen komponens nem rendelkezik vizuális felülettel, pusztán a vezérlés a feladata. Ezek segítségével egyszerűbben nyithatunk meg ill. menthetünk fájlokat, könyvtárakat, betűtípusokat választhatunk ki, elérhetjük a nyomtatót, vagy a beépített számológépet. A paletta képe:



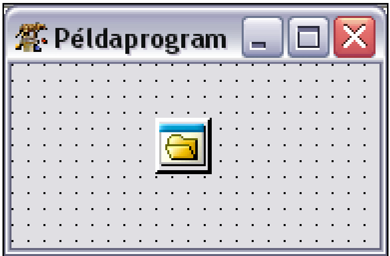
A jegyzetben csak a két legfontosabb komponenst ismertetjük, amikre szinte biztosan szükségünk van ahhoz, hogy egy komolyabb felhasználóbarát programot tudjunk írni (pl. beadandó). Ezek rendre a következők:

- Megnyitóablak
- Mentőablak

❖ ***A megnyitóablak (TOpenDialog) – Bevezető:***

Segítségével elérhetjük a Windows-ban megszokott fájlkiválasztó ablakot. Itt szándékosan nem a tükörfordításból jövő megnyitás szót használtuk, ugyanis ez valójában tényleg csak kiválasztja a fájlt, meg nem nyitja, arra később lesz példa.

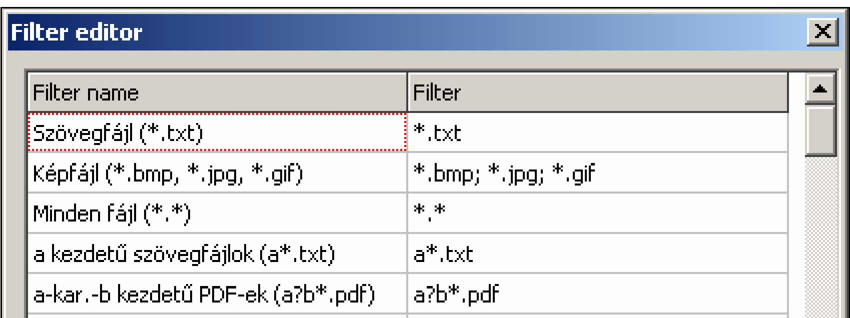
A komponens elhelyezve egy üres formon:



❖ ***A megnyitóablak (TOpenDialog) – Tulajdonságok (Properties):***

1) Filter (szűrő):

Itt elérhető egy filter-, vagyis szűrő-szerkesztő. Ezek a szűrők azért lehetnek hasznosak, mert csak az itt megadott kiterjesztésű fájlokat jeleníti meg a felugró ablak. Egy szűrő két részből áll: a névből, és magáért a szűrésért felelős részből. Névnek természetesen bármi választható, de célszerű a szűrés mikéntjére utalót választani. A szűrőben akár több kiterjesztést is megadhatunk a „;" jellel elválasztva őket. Egy adott szűrés megadásában pedig használhatók a szokásos helyettesítő jelek a ? és a *. Lássunk pár példát az alábbi képernyőkép-részleten:



Filter name	Filter
Szövegfájl (*.txt)	*.txt
Képfájl (*.bmp, *.jpg, *.gif)	*.bmp; *.jpg; *.gif
Minden fájl (*.*)	*.*
a kezdetű szövegfájlok (a*.txt)	a*.txt
a-kar.-b kezdetű PDF-ek (a?b*.pdf)	a?b*.pdf

Természetesen példánk nem valósághű, hiszen elég valószínűtlen, hogy egy alkalmazás ilyen kiterjesztésű állományokat egyszerre kezeljen, pusztán az érzékeltetést tartottuk szem előtt.

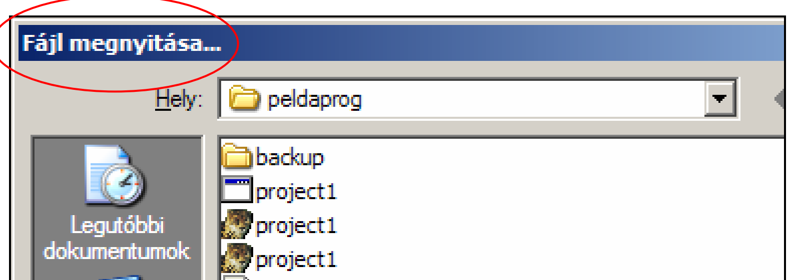
2) InitialDir (kezdőkönyvtár):

Ezzel lehet beállítani az alapértelmezett könyvtárat, vagyis azt, hogy a megjelenő ablak melyik könyvtárból induljon. Ha üresen hagyjuk, akkor a program könyvtára az alapértelmezett (ez utóbbit tapasztalataink szerint a filter használata úgy módosítja, legalábbis WindowsXP esetében, hogy az aktuális felhasználó Dokumentumok könyvtára áll). Ha nem ezt szeretnénk, akkor egy könyvtárútvonalat kell megadnunk, melyet mind relatív, mind abszolút módon megtehetünk a DOS-ban megszokott módokon. Utóbbival hordozandó programok, mint például beadandó, órai programok, esetén ugye célszerű vigyázni, hiszen nagyon is elképzelhető, hogy az útvonal, melyre hivatkoztunk egy másik gépen egyáltalán nem érhető el. Ha nem létező útvonalat adunk meg, akkor ezt a Lazarus semmisnek tekinti, és az alapértelmezett módon működik. Ismét felhívnánk a figyelmet egy furcsa hibajelenségre, jelesül arra, hogy ha a programunk könyvtárába helyezünk egy alkönyvtárt (pl. Inputok, vagy bármilyen más, a működéshez szükséges fájlokat tartalmazó könyvtárat), akkor ezt relatív módon elviekben meg tudnánk adni nagyon egyszerűen (InitialDir-nek egyszerűen beírjuk, a könyvtár nevét, pl.: „Inputok”). ez azonban nem működik. Minden egyéb megadási módról úgy tapasztaltuk, hogy alkalmazható. Nézzünk egy-két példát:

Például	Jelentése
C:\Program	A „C” meghajtó „Program” könyvtárával indul az ablak;
..\MásKönyvtár	Az exe-hez képest egy könyvtárat visszalépve, a „MásKönyvtár”-ral indul;
[x-1 db ..\]..\EgyKönyvtár	Az exe-hez képest x könyvtárnyit visszalépve, az „EgyKönyvtár”-ral indul;
Inputok	Sajnos nem megy ☹;

3) Title (cím):

A Title, azaz cím tulajdonsággal az ablak címsorában lévő szöveg módosítható. Típusa String, tehát akár szabadon is engedhetjük fantáziánkat. Felhasználóbarát programunktól egyébként is elvárható, hogy ne angolul kommunikáljon, így ennek módosításával tovább segíthetjük ezt is. Például „Fájl megnyitása…” esetén az ablak címsora így néz ki:



4) FileName (fáljnév):

Ez az egyik olyan tulajdonság, amit nem igazán éri meg kézileg módosítani. Ide kerül ugyanis a kiválasztott fájl elérési útja, ráadásul így csak megnehezítjük a felhasználó dolgát, hiszen sokkal egyszerűbb az ablakban válogatni. Ennek a tulajdonságnak a segítségével lehet majd megnyitni valójában a fájlt, úgy használva a tulajdonságot, mint egy String típusú adatot, hiszen az is.

❖ ***A megnyitóablak (TOpenDialog) – Használata:***

A TOpenDialog-nak van egy logikai értékű, paraméter nélküli függvénye, az Execute (végrehajtás, megvalósítás). Meghívva megjeleníti az ablakot, és ha a „Megnyitás” gombbal zárjuk be, akkor Igaz, egyéb esetben Hamis értékkel tér vissza. Könnyebb megértés kedvéért a Pascalból ismert fájl-megnyitást, kombináljuk az új lehetőséggel úgy, hogy egy gomb OnClick eseményét használjuk:

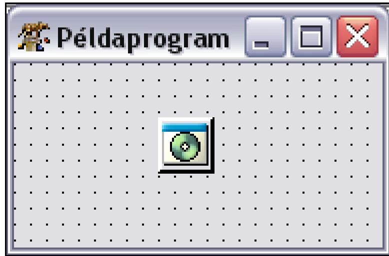
```
procedure TForm1.Button1Click(Sender: TObject);
var f:TextFile;
begin
   If OpenDialog1.Execute then begin
      AssignFile(f,OpenDialog1.FileName);
      Reset(f);
      //…további utasítások…
      CloseFile(f);
      end;
end;
```

Példánkban fontos az elágazás használata. Gondoljuk csak meg, ha Hamis értékkel tér vissza a függvény, akkor a FileName nem biztos, hogy ki van töltve!

❖ ***A mentésablak (TSaveDialog) – Bevezető:***

Ez a komponens nagyon hasonló a megnyitóablakhoz mind felépítésében, beállítási lehetőségeiben, mind használatában, így amikről nem esik itt szó, azok megegyeznek az ott leírtakkal. Említést teszünk azonban egy gyakorlatban gyakrabban alkalmazható tulajdonságáról, és használatát is bemutatjuk.

A komponens elhelyezve egy üres formon:



❖ ***A mentésablak (TSaveDialog) – Tulajdonságok (Properties):***

1) DefaultExt (hiba-kiterjesztés) és FileName (fájlnév):

Az előző komponensnél nem feltételeztük azt, hogy a felhasználó maga írja be az ablakba a kiválasztott fájl nevét, mentés esetén ez azonban nagyon is valószínű. Így fontos megemlíteni, hogy ha beírtuk a nevet, akkor nem biztos, hogy tettünk hozzá kiterjesztést is, ekkor a FileName-ben található útvonal egy kiterjesztés nélküli fájlra mutat. Ez általában problémát jelenthet, de ezt könnyen elkerülhetjük a DefaultExt használatával. Az itt megadott (persze helyes alakú, tehát „.kit") kiterjesztést megkapja a FileName. Megjegyezzük, hogy ha a fájlnév pontot tartalmaz, az utolsó pont utáni részt a Lazarus automatikusan kiterjesztésnek érzékeli.

2) Filter (szűrő):

Használata teljes mértékig megegyezik a megnyitóablak e részével, mindössze felhívnánk a figyelmet, hogy ugyan itt is adott a lehetőség több szűrő egy név alatti használatára, ezt azonban könnyen látható egyértelműségi problémák miatt nem javasoljuk.

3) FilterIndex (szűrőmutató):

Segítségével többféle szűrő használata esetén el tudjuk dönteni, melyiket választotta a felhasználó, így akár automatikusan megadhatjuk a megfelelő kiterjesztést a fájlhoz. Az indexelés egytől indul, tehát ha például, három szűrőt használunk, akkor az 1, 2, ill. 3 számok döntik melyik a kiválasztott.

❖ ***A mentésablak (TSaveDialog) – Használata:***

Használata szinte teljes mértékben megegyezik a megnyitóablakéval, mégis adunk egy példakódot. Itt a Pascalból ismert fájllétrehozást vettük alapul.

```
procedure TForm1.Button2Click(Sender: TObject);
var f:TextFile;
begin
   If SaveDialog1.Execute then begin
      Case SaveDialog1.FilterIndex of
         1 : SaveDialog1.FileName:=SaveDialog1.FileName+'.txt';
         2 : SaveDialog1.FileName:=SaveDialog1.FileName+'.doc';
         3 : SaveDialog1.FileName:=SaveDialog1.FileName+'.pdf';
         //…stb
         end;
      AssignFile(f,SaveDialog1.FileName);
      ReWrite(f);
      //…további utasítások…
      CloseFile(f);
      end;
end;
```