PaedDr. Végh László

Programozás Delphi-ben

Komárom, 2009. április 30.

Tartalom

1 Mi a Delphi?

Bevezetésként nézzük meg, milyen fő jellemvonásai vannak a *Delphi* programozási nyelvnek. A *Delphi* alapja az **Object Pascal** programozási nyelv, amely az ismert *Turbo Pascal* objektumos felépítménye. Éppen ezért sok mindent, amit megtanultunk a pascal programozási nyelvben, most fel fogunk tudni használni a *Delphi*-ben. Fontos, hogy valamilyen szinten már tudjunk programozni – ismerjük a vezérlési szerkezetek: ciklusok *(for..do*, *while..do*, *repeat..until*), elágozások (*if..then..else*, *case..end*) fogalmát. Tudnunk kéne, hogyan kell a program elején változókat deklarálnunk (*var..*) és ismernünk a pascalban használt változók alaptípusait (a *Delphi*-ben használt egyszerű változók típusait a jegyzet végén levő mellékletben megtalálhatjuk).

Miért jobb a *Delphi* fejlesztői környezetében programoznunk más hozzá hasonló programozási nyelv helyett? Elsősorban a **produktivitás** végett. A *Delphi* az egyik legeffektívebb eszköz, mellyel a *Windows* operációs rendszer alatt alkalmazásokat hozhatunk létre. A rengeteg vizuális eszköznek és integrált környezetnek köszönhetően maximálisan leegyszerűsített fejlesztői fázisa van az alkalmazás létrehozásának. Amit eddig 8-10 órán átt írtunk pascalban, azt *Delphi*-ben létre tudjuk hozni pár óra alatt.

Ezen kívül a programozás *Windows* alatt általában (tehát *Delphi*-ben is) különbözik a szekvenciális programozástól, melyet a pascal programozási nyelvnél megismerhettünk. A *Windows* alatti programozás **eseményekkel irányított** programozás. A program irányítását az operációs rendszer végzi, a programozónak csak a rendszer különféle eseményeire kell reagálnia. Az irányítás tehát

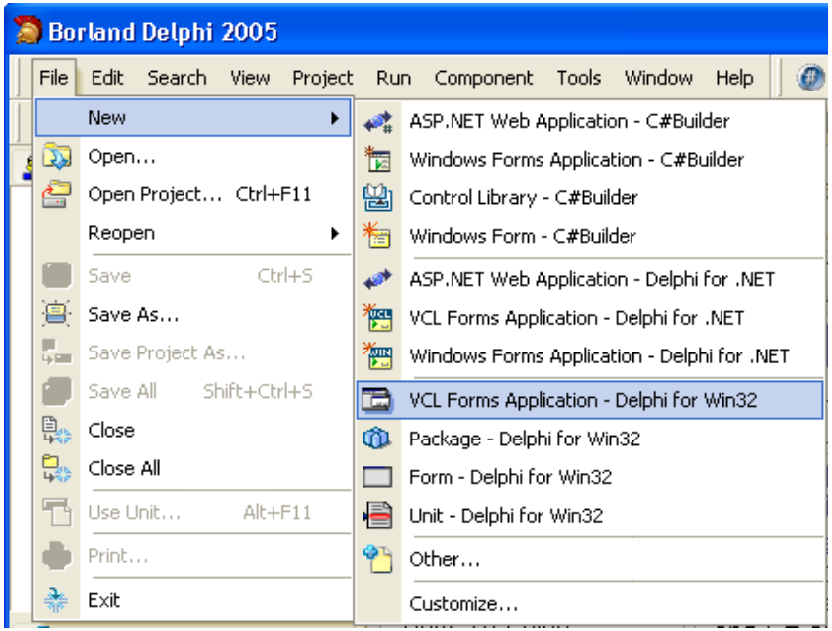
továbbra is az operációs rendszernél marad. Ha „történik valami” (pl. a felhasználó klikkel az egér valamelyik gombjával), a rendszer küld az alkalmazásunknak egy üzenetet, melyet a következőhöz hasonló képen képzelhetünk el: „kedves alkalmazás, a te főablakodban bal egérkattintás történt az X, Y koordinátákon”. Az alkalmazás erre az üzenetre reagálhat (pl. úgy, hogy kiír valamit), vagy figyelmen kívül hagyhatja – a programozónak csak ezt a reakciót kell megfogalmaznia (hogy mit tegyen az alkalmazás). Ezekről természetesen még szó lesz bővebben is a következő fejezetekben.

Ebben a jegyzetben levő ábrák a Delphi 2005-ös (*Delphi* 9) verziójából valók. Természetesen az alkalmazások létrehozhatók a leírtak alapján alacsonyabb, ill. magasabb verziószámú *Delphi*-ben is.

A 2005-ös változatnak négy kiadása létezik – Personal, Professional, Architect és Enterprise. A Personal változat ingyenesen használható nem kommerciális célokra – tanulásra nekünk ez tökéletesen megfelel. Ennek a változatnak az egyik legnagyobb megkötése, hogy nem tartalmaz hálózat és adatbázis támogatást – ezek csak a magasabb (Professional, Architect, Enterprise) változatoknál érhetők el. Ha szeretnénk telepíteni a *Delphi 2005 Personal* változatát, a telepítés után szükségünk lesz egy licensz kódra (CD key), melyet a www.borland.com – *Downloads* – *Delphi* weboldalon ingyenesen kérhetünk egy rövid kérdőív kitöltése után.

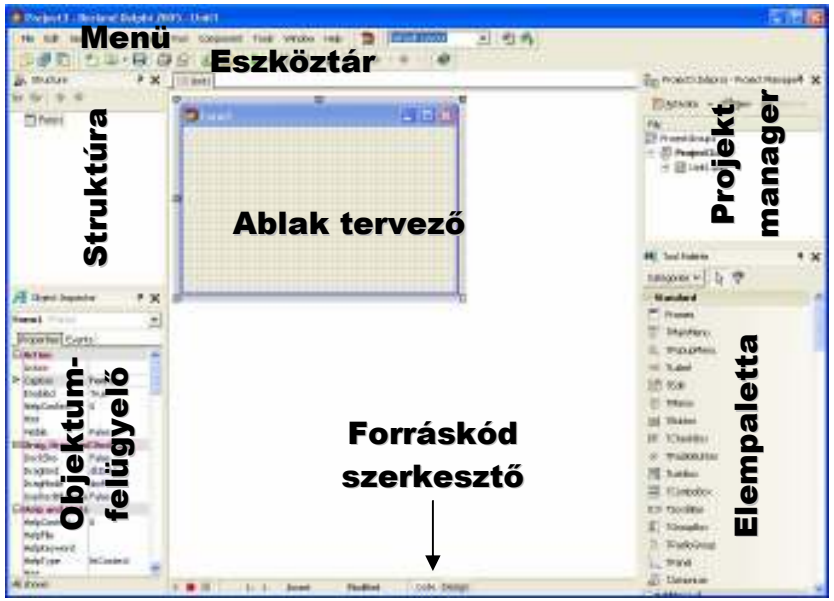
2 Az integrált fejlesztői környezet

A *Delhi* elindítása után új alkalmazás létrehozásához válasszuk ki a **File – New – VCL Form application - Delphi for Win32** menüpontot. (VCL = Visual Component Library = Vizuális komponenskönyvtár)



Itt láthatjuk, hogy *Delphi 2005*-ben nem csak *Delphi Win32* alkalmazást, de *C#*, illetve *.Net* alkalmazásokat is létrehozhatunk. Mi csak *Delphi Win32* alkalmazásokat fogunk létrehozni.

Miután létrehoztunk egy új alkalmazást, az alábbi ábrához hasonlót láthatunk. Nézzük meg részletesebben miből áll a *Delphi* fejlesztői környezete:



Menü: A különféle beállítások, programfuttatások, segítség, keresés, stb. megvalósítását végezhetjük el itt.

Eszköztár: A menüből is hívható funkciók gyors elérését teszik lehetővé. Ha egérrel „rámegyünk" valamelyik ikonra, akkor egy feliratban (buborékban) tájékoztatást kapunk az adott funkcióról.

Ablak tervező: A leendő programunk formáját tervezhetjük meg itt aránylag egyszerű módon. Megváltoztathatjuk az ablak (form)

méretét, komponenseket (nyomógombokat, címkéket, képeket, stb.) helyezhetünk el rajta.

Elempaletta: Itt választhatjuk ki a komponenseket, melyeket elhelyezhetünk az ablakunkon (form-on).

Objektum felügyelő: Ez a *Delphi* egyik legfontosabb része. Segítségével beállíthatjuk az egyes komponensek tulajdonságait (Properties) és a komponensek reakcióit az eseményekre (Events). TIPP: Az Objektum felügyelőben a tulajdonságok és az események kategóriák szerint vannak sorba rendezve. Ezt átállíthatjuk, ha rákattintunk az egér jobb gombjával az Objektum felügyelő tetszőleges mezőjére és kiválasztjuk az „Arrange – by Name" menüpontot. Hasonlóan az „Arrange – by Category" segítségével visszaállíthatjuk a kategóriák szerinti elrendezést.

Forráskód szerkesztő: A *Delphi*-nek az a része, ahova magát a forráskódot (programot) írjuk. Ezt az ablakot kezdetben nem látjuk, az ablak alján levő „code" fül segítségével jeleníthetjük meg. Ha vissza szeretnénk menni a form-unk tervezéséhez, ugyanott klikkeljünk a „design" fülre.

Struktúra: Ebben az ablakban láthatjuk a form-unkon levő komponensek hierarchikus elrendezését.

Project manager: A *Delphi*-ben mindig egy komplex rendszerben (Projektben) dolgozunk. Minden egyes alkalmazásunk egy projektből áll. Egy projekt tetszőleges mennyiségű fájlt használhat. Ezek a fájlok lehetnek programfájlok (unit-ok), a hozzájuk tartozó ablakok (form-ok) és az ablakon levő komponensek elrendezését tartalmazó fájlok, adat-, kép-, hang-, stb. fájlok. Azt, hogy a projektünkhöz milyen fájlok kapcsolódnak és melyik fájl melyik fájlhoz tartozik, a project manager-ben láthatjuk. Kezdetben a projektünkhöz két fájlt kötődik –

egy programkódot tartalmazó fájl (.pas kiterjesztésű) és egy olyan fájl, amely a form-on levő komponensek elrendezését, kezdeti beállításait tartalmazza (.dfm kiterjesztésű).

3 Első programunk Delphi-ben

Az első programunk annyit fog tenni, hogy kiír egy szöveget az ablakunkba. A form-unkon lesz még egy nyomógomb, amely megnyomásával az alkalmazást bezárhatjuk. Pelda01

Az első alkalmazásunk elkészítését egy kicsit részletesebben fogjuk tárgyalni. A további alkalmazások létrehozását a jövőben már ennél tömörebben fogjuk elemezni.

3.1 Komponensek kiválasztása

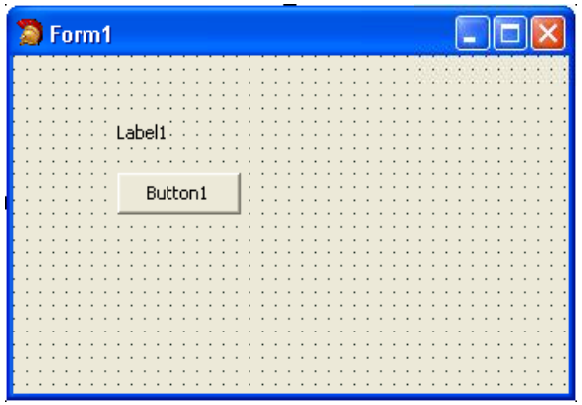
Az első lépések egyike, melyet minden alkalmazás fejlesztésének kezdetén meg kell tennünk, a megfelelő komponensek kiválasztása.

1. Az új alkalmazás létrehozásához, ha még nem tettük meg, klikkeljünk a **File – New – VCL Form Application - Delphi for Win32** menüpontra. A képernyő közepén megjelenik a főablakunk (form-unk).
2. Az elempalettában válasszuk ki a **TLabel** (címke) komponenst. (Megjegyzés: A „T” betű a „type” rövidítése – általában a Delphiben minden osztályt, tehát a komponenseket is így szokás jelölni a nevük előtt, ezzel is segítve a programkód könnyebb megértését. Az osztályokról majd még lesz szó bővebben a későbbi fejezetekben.)
3. Klikkeljünk az ablakunkban arra a helyre, ahová a címkét szeretnénk tenni. A címke elhelyezésekor a *Delphi*

automatikusan az objektumhoz a **Label1** nevet rendeli hozzá.

4. Hasonlóan helyezzünk el az ablakunkon egy **TButton** (nyomógomb) komponenst. A *Delphi* az elhelyezett objektumhoz a **Button1** nevet rendeli hozzá.

Jelenleg az ablakunkon két komponens – Label1 és Button1 található, hasonlóan, ahogy az alábbi ábrán is láthatjuk:

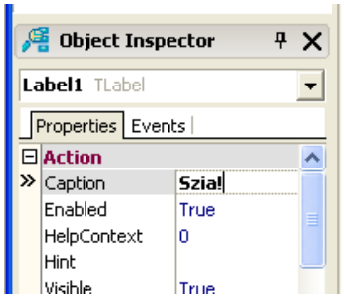


3.2 Komponensek tulajdonságainak beállítása

Miután kiválasztottuk a szükséges komponenseket, beállítjuk azok néhány tulajdonságát. Mi most csak a komponensek feliratait, méreteit, elhelyezéseit fogjuk változtatni. Általában minden komponensnek ennél jóval több tulajdonsága van – ezekkel majd folyamatosan megismerkedünk.

1. Klikkeljünk a **Label1**-re a főablakunkban (Form1-en). Ezzel a kiválasztott komponens aktív lesz az Objektum felügyelő

ablakában. Itt az ablak tetején két választási lehetőségünk van – Properties (tulajdonságok) és Events (események). Ha nincs kiválasztva, válasszuk most ki a **Properties** fület. Ezzel kijelöltük, hogy a komponens tulajdonságait fogjuk beállítani. Az Objektum felügyelőben két oszlopot láthatunk. A bal oldali oszlopban vannak a komponens tulajdonságainak a nevei, a jobb oldali oszlopban a hozzájuk tartozó értékek. Keressük itt meg a **Caption** (felirat) tulajdonságot és klikkeljünk rá. A „Label1” érték helyett írjuk be: „Szia!”.



Észre vehettük, hogy az alkalmazásunk form-ján is rögtön megváltozott a felirat.

2. Klikkeljünk most a form-unkon a **Button1** feliratú nyomógombra. Ekkor az Objektum felügyelőben a Button1 tulajdonságai jelennek meg. Klikkeljünk a **Caption** tulajdonságra és írjuk be: „Kilépés”.

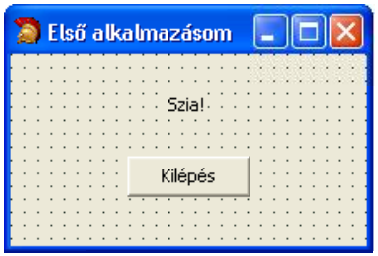
*Jegyezzük meg, hogy a **Caption** beállításával a komponens neve nem változik meg, csak a felirat, amely megjelenik rajta. Például a mi nyomógombunk felirata **Kilépés**, de a programkódban továbbra is **Button1** néven fog szerepelni!*

3. Vegyük észre, hogy az ablakunk (alkalmazásunk) felső sávjában a Form1 felirat szerepel. Ez a főablak alapértelmezett felirata. Változtassuk meg ezt is. Klikkeljünk bárhova a form-unkra (de úgy, hogy ne klikkeljünk se a címkére, se a nyomógombra). Ekkor az Objektum felügyelőben a főablakunk tulajdonságait állíthatjuk be. Válasszuk itt ki ismét a **Caption** tulajdonságot és írjuk be feliratnak: „Első alkalmazásom”.

4. Változtassuk meg a főablak méretét kisebbre úgy, ahogy azt tennénk bármilyen Windows alkalmazásnál – fogjuk meg az alkalmazásunk jobb alsó sarkát (vagy jobb és utána alsó szélét) és húzzuk beljebb. Az ablakunk kisebb lett. Az ablakunk méretét beállíthatjuk az Objektum felügyelőben is a **Width** (szélesség) és **Height** (magasság) tulajdonságok segítségével.

5. Végül rendezzük el az ablakunkban a címke és a nyomógomb komponenseket. Egyszerűen fogjuk meg azt a komponenst, amit máshova szeretnénk tenni és vigyük át egérrel. Természetesen ezt is beállíthatjuk az Objektum felügyelőben is a **Top** (távolság a form tetejétől) és a **Left** (távolság a form bal szélétől) tulajdonságok segítségével. A komponensek elhelyezkedését beállíthatjuk szintén a **Position** kiválasztásával a lokális pop-up menüből, amely a komponensre jobb egérgombbal klikkelve jelenik meg.

Ezzel befejeztük az alkalmazásunk külalakjának tervezését, amely jelenleg így néz ki:



Alkalmazásunk ablaka pontosan így fog kinézni futtatáskor is (természetesen rácspontok nélkül lesz). A következő lépésben már csak be kell állítanunk, hogy a Kilépés gombra kattintással a program befejezze a futását.

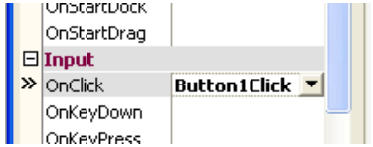
3.3 A reakciók beállítása az eseményekre

A következő fontos lépés a reakciók beállítása az eseményekre. Eseményeknek nevezünk mindent, ami az operációs rendszerben történik és valahogyan összefügg a komponenseinkkel, mint például: kattintás egérrel, billentyű megnyomása, stb.

1. Először is meghatározzuk, milyen eseményekre szeretnénk reagálni. Ezekből most csak egyetlen egy lesz. A **Kilépés** gombra kattintásnál szeretnénk, ha az alkalmazásunk befejeződne. Megnyitjuk ezért az Objektum felügyelőben a **Button1** komponenst. Ez megtehetjük úgy, hogy egyszerűen rákattintunk a komponensre a form-unkon, vagy kiválasztjuk az Objektum felügyelő legördülő listájából.

2. Az Objektum felügyelőben most válasszuk ki az **Events** (események) fület. Mivel mi a „komponensre kattintásra” szeretnénk reagálni, az események közül válasszuk ki az

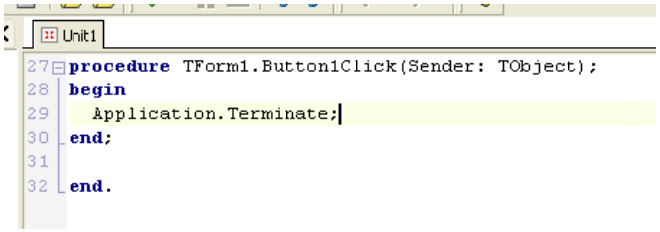
OnClick eseményt. A jobb oldali oszlopban az OnClick mellett levő üres mezőre klikkeljünk rá duplán.



Az Objektum felügyelőnek ebben az üres mezőjében most megjelenik a **Button1Click** felirat. Ez egy eljárás neve, amely mindig meg lesz hívva, ha a felhasználó a Kilépés gombra klikkel.

Továbbá észre vehettük, hogy eltűnt az ablak tervező és helyette a forráskód szerkesztő ablaka jelent meg. Ebbe az ablakba fogjuk megírni a programkódot. A *Delphi* automatikusan létrehozta a **Button1Click** eljárást és a kurzort az eljárás begin..end kulcsszavai közé tette. Nekünk már csak az a dolgunk, hogy ide beírjuk azt a programrészt, amely meghatározza, hogy mit tegyen a program a Kilépés gombra kattintáskor.

A mi esetünkben a programkód beírása egyetlen lépésből fog állni. Írjuk be a begin..end közé, ahol a kurzor villog a következő sort: **Application.Terminate;**



A programrész írásakor észrevehettük, hogy megjelentek a kurzor mellett egy kis ablakban különféle parancsszavak. Ez az automatikus kiegészítés a programozó munkáját szeretné

megkönnyíteni és meggyorsítani. Elég elkezdenünk írni az utasítást, majd kiválasztani a megjelenő listából a megfelelő parancsot. Ha a lista véletlenül nem jelenik meg automatikusan, azt előhívhatjuk manuálisan is a **Ctrl + Space** billentyűkombináció segítségével.

Hasonló módon fogunk a jövőben programozni bonyolultabb események kezelését is. Az egy sornyi programkód helyet (ami most Application.Terminate;) fogjuk beírni a néha hosszú és bonyolultnak tűnő programkódot.

Ezzel az alkalmazásunk létrehozásának fázisa valójában befejeződött!

3.4 Program mentése, fordítása, futtatása

Az első alkalmazásunk kész! Hátra maradt még az alkalmazás lefordítása és futtatása. Mindenek előtt azonban mentsük el az egész projektünket. Bár nem kötelező, de ajánlatos mindig, minden fordítás és futtatás előtt az alkalmazás összes részét elmenteni, ha ugyanis a fordításnál vagy futtatásnál komolyabb hiba lépne fel, elveszhetne az alkalmazásunk el nem mentett része.

1. Az egész alkalmazás elmentéséhez klikkeljünk a **File – Save All** menüpontra. Megjelenik egy ablak, amelyben meg kell adnunk az elmenteni kívánt unit nevét. Ajánlom, hogy minden egyes projektnek hozzunk létre egy új alkönyvtárat, és abba mentsük el a projekt összes állományát – a Delphi minden egyes projekthez több állományt hoz létre, és ha mindig ugyanabba a mappába mentenénk, egy idő után nem igazodnánk ki a mappában található fájlokon.

2. Adjuk meg a unit nevét, tehát annak a forráskódnak a nevét, amelyben a Button1Click eljárásunk is van. Itt hagyhatjuk a **unit1.pas** nevet.

3. Majd megjelenik egy újabb dialógusablak, ahol a projekt nevét kell megadnunk. Ide írjuk be az **elso.dpr** nevet. Ezzel a projektünket elmentettük.

A következő lépés az alkalmazás lefordítása. A **fordítás** alatt a programozó számára érthető forráskódból a számítógép számára érthető állomány létrehozását értjük. A fordítás két lépésben zajlik: egy kompilátor és egy linker segítségével. A **kompilátor** az alkalmazás vagy annak egy részének megírása után a projektet kompillálja egy „közbülső" formába (ekkor minden modulhoz létrejön egy .DCU kiterjesztésű állomány). A **linker** ezekből a kompillált állományokból létrehoz egy futtatható alkalmazást (.EXE kiterjesztésű állományt). Ez az állomány már bármelyik Windows operációs rendszert használó számítógépen futtatható a *Delphi* jelenléte nélkül is.

1. Az alkalmazás lefordításához és futtatásához klikkeljünk az eszköztárban a  ikonra (vagy válasszuk ki a főmenüből a **Run – Run** parancsot, ill. nyomjuk meg az **F9** funkcióbillentyűt).

2. Az első alkalmazásunk elindult. Próbáljunk rákattintani a **Kilépés** gombra. Működik?

Az első alkalmazás létrehozása sikeresen magunk mögött van. Ha belenézünk a mappába, ahová az alkalmazást elmentettük, láthatunk többek között egy **elso.exe** nevű állományt. Ezt az állományt bárhol és bármikor a Windows alatt elindíthatjuk és gyönyörködhetünk az első működő alkalmazásunkban.

Vegyük észre, hogy az alkalmazásunk egy csomó olyan funkcióval is rendelkezik, amelyet nekünk nem kellett beprogramoznunk – az ablakot lehet mozgatni, átméretezni, minimalizálni, maximalizálni, tartalmaz rendszermenüt (melyet a bal felső sarokban levő ikonra klikkelléssel hívhatunk elő), stb. Ezen funkciókat a *Delphi* „programozta" be a *Windows* operációs rendszerrel együttműködve.

*Megjegyzés az első alkalmazásunkhoz: a program befejezésére az **Application.Terminate** függvényt használtuk. Ha valaki régebben már programozott Delphi-ben, lehetséges, hogy erre más metódust használna (pl. **form1.close**) és az Application.Terminate túl erős eszköznek tűnik neki. Az Application.Terminate nem az egyetlen használható megoldás, de elsődlegesen ez a függvény szolgál az alkalmazás befejezésére és használata teljesen korrekt és biztonságos.*

4 A projekt fájlfelépítése

Vizsgáljuk meg, hogyan néz ki a projektünk fájlfelépítése. Ha megnézzük a mappánkat, ahova a projektet mentettük, több állományt találhatunk benne. Elsősorban nézzük meg, melyik állományokat kell átmásolnunk, ha a forráskódot szeretnénk más számítógépre átvinni:

***.DPR** Delphi Project. Minden projektnek létezik egyetlen ilyen fő forrásállománya. Ez elsősorban létrehozza az alkalmazás ablakait és sikeres létrehozáskor elindítja az alkalmazást.

***.BDSPROJ** Borland Development Studio Project fájl. Minden projekthez egyetlen ilyen állomány tartozik. A projekt különféle beállításait tartalmazza.

***.PAS** Unit forráskód. Ez tartalmazza az egyes modulok programkódját. Egy projektnek egy vagy több ilyen állománya lehet. Gyakorlatilag az alkalmazás minden egyes ablakához tartozik egy ilyen állomány, de ezeken kívül a projekt még tartalmazhat további ilyen állományokat (modulokat) is, melyekhez ablak nem tartozik.

***.DFM** Delphi Form. Formleírás. Azokhoz a modulhoz, melyekhez tartozik ablak, léteznek ilyen kiterjesztésű állományok is. Ezek az állományok az ablak és a rajta levő komponensek listáját és tulajdonságait tartalmazzák, tehát mindent, amit az Ablak tervezőben, ill. Objektum felügyelőben beállítottunk (a komponensek elrendezését, méreteit, feliratait, egyéb tulajdonságait

és a komponensek egyes eseményeihez tartozó eljárások neveit is).

***.RES** Resource. Windows erőforrásfájl. Az alkalmazásunk ikonját tartalmazza.

A további állományokat nem szükséges átmásolnunk, ezen állományok többségét a *Delphi* a fenti állományokból hozta létre automatikusan a projekt fordításakor. Ezek közül számunkra a legfontosabb a ***.EXE** kiterjesztésű állomány. Ha alkalmazásunkat más gépre szeretnénk átvinni és futtatni (a forráskód nélkül), elég ezt az állományt átmásolnunk és futtatnunk (ebben a másik gépben nem szükséges hogy legyen *Delphi*). Természetesen, ha a programunk kódját meg szeretnénk nézni, ill. szeretnénk benne valamit javítani, majd újra fordítani, nem elég ez az egyetlen állomány, szükséges hozzá az összes fent említett állomány is.

5 A forráskódok áttekintése

Ebben a fejezetben megnézzük, milyen programkódokat hozott létre a *Delphi* az előző program megírásakor.

5.1 Az ablak forráskódja (.pas)

Amikor megtervezzük, hogy miként nézzen ki az alkalmazásunk ablaka, a *Delphi* automatikusan generál hozzá forráskódot. Nézzük meg most ennek a **unit1.pas** állománynak a szerkezetét:

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```



```
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.Terminate;
end;

end.
```

A **unit unit1;** a modulunk nevét adja meg. Ezt követően észrevehetjük, hogy a unit két részre van bontva. Az első része az **interface** kulcsszóval kezdődik (csatlakozási vagy publikus felület), a második az **implementation** (kivitelezési vagy implementációs rész).

Az **interface** részben fel vannak sorolva azok a típusok, változók, melyeket a unitban használunk, és amelyeket szeretnénk hogy más unitból, programból is elérhetők legyenek (miután ott megadjuk a **uses unit1;** sort).

Az **implementation** részben egyrészt a feljebb felsorolt eljárások, függvények megvalósítását írjuk le – tehát azt, mit is tegyen az adott eljárás vagy függvény. Másrészt ide írhatjuk azokat a további változókat, eljárásokat, függvényeket is, melyeket csak a mi unit-unkon belül szeretnénk használni.

Nézzük meg részletesebben, mi van a programunk **interface** részében. A **uses** parancs után fel vannak sorolva azok a modulok, melyek szükségesek a mi modulunk futtatásához.

A **type** parancs után a **TForm1** típusú osztály definícióját látjuk. Ez valójában a mi főablakunk típusa. Láthatjuk, hogy **TForm** típusú osztályból lett létrehozva. (*Osztály = olyan adattípus, melyet valamiféle sablonnak képzelhetünk el bizonyos objektumok – mi esetünkben főablak – létrehozásához. Az osztály tartalmazhat adatokat, eljárásokat*

és függvényeket. A Delphi-ben szokás az osztályok neveit mindig T betűvel kezdeni.) Továbbá észrevehetjük, hogy a **TForm1** tartalmaz egy nyomógombot (**Button1**) és egy címkét (**Label1**), majd egy **Button1Click** nevű eljárást (ez a mi eljárásunk, amit az OnClick eseményhez hoztunk létre – ez az eljárás kerül futtatásra, ha a felhasználó rákattint a „Kilépés" nyomógombra). Ezek után a **TForm1** osztály **private** (magán – csak az osztályon belül használható) és **public** (nyilvános – az osztályon kívülről is elérhető) változók, eljárások definíciója következhet. Nekünk itt most nincs egyik sem.

A **var** kulcsszó után egyetlen változónk van deklarálva, ez a **Form1** objektum, ami valójában a mi alkalmazásunk főablaka.

Az **implementation** részben találunk egy **{$R *.dfm}** sort. A **$R** egy külső resource fájl beolvasását jelzi. A ***.dfm** most nem azt jelzi, hogy az összes **.dfm** végződésű állományt olvassa be, hanem itt a * csak a mi unitunk nevét helyettesíti, tehát csak a **unit1.dfm** állomány beolvasására kerül sor. Ez a fájl tartalmazza a főablakunk és a rajta található komponensek kezdeti beállításait.

Végül a **TForm1.Button1Click** eljárás megvalósítását láthatjuk, melynek **begin..end** közötti részét mi írtuk be.

*Végül egy megjegyzés a modulokhoz, tehát a **.pas** végződésű állományokhoz: Egy alkalmazáson belül több ilyen állományunk is lehet. Alkalmazásunk minden egyes ablakához tartozó forráskód egy ilyen külön modulban található. Ezen kívül az alkalmazásunk tartalmazhat még további ilyen modulokat is, melyekhez ablak (form) nem tartozik.*

5.2 Alkalmazás projekt fájlja (.dpr)

Nézzük meg, mit tartalmaz az alkalmazás projekt állománya. Valójában ez az állomány nem mást, mint egy hagyományos Pascal fájl más kiterjesztéssel:

```
program elso;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Láthatjuk, hogy ez a program használja ez előbb elemzett **unit1.pas** modult – tehát azt a modult, amely az alkalmazásunk főablakát tartalmazza. Ha az alkalmazásunkban több ablakunk lenne, itt lennének felsorolva az összes hozzájuk tartotó modulok (unitok).

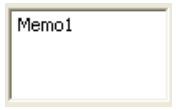
A **{$R *.res}** sor most az **elso.res** állomány csatolását jelzi. Ez az állomány tartalmazza az alkalmazásunk ikonját.

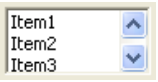
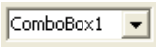
A begin..end közötti főprogram inicializálja az alkalmazást, létrehozza a főablakunkat és elindítja az alkalmazást.

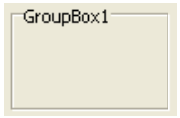
6 Alap komponensek áttekintése

Komponensek alatt azokat az elemeket értjük, melyeket elhelyezhetünk az alkalmazásunk ablakában (form-on). Ezekből a *Delphi*-ben rengeteg van (az Enterprise változatban több mint 200). Amennyiben ez nekünk nem elég, létrehozhatunk saját komponenseket is, ill. sok kész komponenst találhatunk az Interneten is.

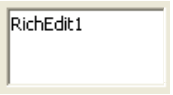
Standard paletta komponensei:

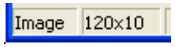
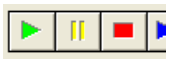
MainMenu, PopupMenu 	A főmenu és lokális pop-up menu létrehozására szolgáló komponens. A *Delphi* rendelkezik egy úgynevezett „*Menu Designer*”-rel, amely segítségével részletesen beállíthatjuk a menü egyes menüpontjait.
Label 	Címke. Ez a komponens csupán szöveg megjelenítésére képes. Ennek ellenére a címkénél több különböző eseményre is reagálhatunk.
Edit 	Beviteli mező. Egysoros szöveg bevitelére vagy megjelenítésére szolgáló komponens.
Memo 	Hosszabb, többsoros szöveg megjelenítésére szolgáló komponens. Használható például egy egyszerű szövegszerkesztő alkalmazás létrehozásánál, ha nem akarjuk a bonyolultabb **RichEdit** komponenst használni.

Button 	Nyomógomb. Ez az egyike a leggyakrabban használt komponenseknek.
CheckBox 	Logikai értékű (igen,nem) információk bevitelére vagy megjelenítésére szolgáló komponens. Egyszerre bármennyi ilyen komponens ki lehet jelölve (pipálva), de nem szükségszerű kijelölni egyetlen komponenst sem.
RadioButton 	A **CheckBox**-hoz hasonló komponens, de itt a felhasználó csak egyet jelölhet ki több ilyen komponens közül. Egy kijelölése mindenképpen szükséges. **RadioButton**-t lehetne pl. használni a szöveg színének kijelölésére (mivel egyszerre csak egy színt választhatunk), **CheckBox**-ot pedig a szöveg félkövér, dőlt, aláhúzott típusának kijelölésére (mivel ezeket bárhogy kombinálhatjuk, egyszerre többet is kijelölhetünk).
ListBox 	Lista. Több hasonló típusú érték kiírására szolgál, melyekből lehet egyet vagy többet kijelölni (a komponens beállításától függően).
ComboBox 	Legördülő lista. Hasonló a **ListBox**-hoz, de ezzel helyet lehet megtakarítani az alkalmazásunkban. A felhasználó választhat a listából, de van lehetősége új érték beírására is, amely a listában nem szerepel.
ScrollBar 	Görgetősáv. Valamilyen egész szám érték beállítására szolgálhat.

GroupBox, RadioGroup, Panel 	Komponensek, melyek más komponensek logikai csoportokba való sorolására szolgálnak. Ezen komponenseknek nem csak vizuális jelentősége van, de logikai is.

Néhány az **Additional, Win32, System, Dialogs, Samples** palettákról:

BitBtn 	Nyomógomb, mely a **Button**-tól eltérően bitképet is meg tud jeleníteni magán, így segítségével könnyen létrehozhatunk bármilyen külalakú nyomógombot.
SpeedButton 	Eszköztáron használható gombok. A gomb lehet lenyomott állapotban is, továbbá beállítható kölcsönös kizárás is lenyomott állapotban (gondoljunk például a Word szöveg igazítási gombjaira – balra, középre, jobbra, sorkizárás)
Image 	Kép. Az alkalmazásban ennek a komponensnek a segítségével tudunk megjeleníteni képet. A komponens rajz létrehozására is szolgálhat (pl. egy rajzprogramban).
RichEdit 	Az **Memo** komponens bővített változata, mely jóval több tulajdonsággal rendelkezik. Segítségével bonyolultabb szövegszerkesztő is létrehozható.

StatusBar 	Állapotsáv. Az alkalmazásunk ablaka alján írhatunk ki segítségével a felhasználónak különféle információkat.
Timer 	Időzítő. Ha az alkalmazásunk periodikus időközönként fog valamilyen műveletet végezni, szükségünk lesz erre a komponensre.
MediaPlayer 	A komponens segítségével hang- és videófájlokkal dolgozhatunk.
OpenDialog, SaveDialog, … 	Standard dialógusablakok. Ha szeretnénk megnyitni vagy menteni egy állományt, nem kell külön dialógusablakokat készítenünk a fájl megkeresésére, hanem helyette használhatjuk ezeket. Hasonlóan léteznek standard dialógusablakok szín és betűtípus kiválasztására, nyomtatásra, vagy szó keresésére egy szövegben.
SpinEdit 	Praktikus komponens, amely alkalmas például egész számok bevitelére. A klasszikus beírás mellett megengedi, hogy a felhasználó az értéket a jobb szélén található fel és le nyilak segítségével állítsa be.

Néhány komponens a tervezésnél az ablakunkban már a végleges állapotában jelenik meg (pl. Label, Button, Edit, …), némelyik azonban egy kis négyzettel van ábrázolva (pl. Timer, MainMenu, OpenDialog, …). Az utóbbiak olyan komponensek, melyek az alkalmazás futtatásakor mindig másképp nézhetnek ki, egyáltalán nem

láthatók, vagy egy saját ablakot hoznak létre. Ide tartozik például a Timer komponens is, amely az alkalmazás futásakor nem látható, de a programban ott van és használhatjuk az összes funkcióját (pontosabban metódusát).

Néhány komponensnek van saját tervezője (Designer-je) is, amely segítségével könnyebben beállítható a komponens külalakja és tulajdonságai. Ilyen komponens például a MainMenu, PopupMenu, vagy a StatusBar.

7 Komponensek tulajdonságaik

Minden komponensnek vannak tulajdonságaik (melyek valójában az adott osztály – komponens – attribútumai). A tulajdonságok nem csak a komponens külalakját határozzák meg, de a viselkedését is. Sok tulajdonság közös több komponensnél is, de vannak olyan egyedi tulajdonságok is, melyek csak egy-egy komponensnél találhatók meg.

Az alkalmazás létrehozása alatt a tulajdonságok értékeit az **Objektum felügyelő** segítségével tudjuk megváltoztatni, az alkalmazás futása alatt pedig a programkód segítségével egyszerű hozzárendeléssel (írással ill. olvasással). Pl.: **Label1.Caption := 'Címke új felirata';**

Az Objektum felügyelőben a komponenseknek csak azokat a tulajdonságokat találjuk meg, melyek hozzáférhetők a tervezés alatt. Ezen kívül léteznek még úgynevezett **run-time** tulajdonságok is, melyek csak az alkalmazás futása alatt érhetők el.

Továbbá megkülönböztetünk még **read-only** (csak olvasni lehet) és **write-only** (csak írni lehet) tulajdonságokat. Ezek a tulajdonságok általában csak a program futásakor érhetők el.

Ennek a fejezetnek a további részében csak azokkal a tulajdonságokkal fogunk foglalkozni, melyek a tervezés alatt is elérhetők, tehát, megtalálhatók az Objektum felügyelőben. Most csak a közös tulajdonságokat soroljuk fel, amely minden komponensnél léteznek, a többi „egyedi" tulajdonságot az egyes komponenseknél fogjuk külön tárgyalni.

Ha szeretnénk tudni valamelyik tulajdonságról többet, klikkeljünk rá az adott tulajdonságra az Objektum felügyelőben, majd nyomjuk meg az **F1** funkcióbillentyűt. Ennek hatására megjeleni a *Delphi* súgója a kijelölt tulajdonságra.

7.1 Komponens neve és felirata

Minden komponensnek a *Delphi*-ben van neve (**Name** tulajdonság). Ha a komponens nevét nem állítjuk be, a *Delphi* automatikusan beállít neki egy nevet, amely a komponens típusából (pl. Button) és egy sorszámból áll, pl. Button5. A komponens nevének egyedinek kell lennie a tulajdonosán belül. Egyszerűbben megfogalmazva az alkalmazásunkban lehet két ablak (form), amelyeken ugyanolyan nevű komponens van, de nem lehet ugyanolyan nevű komponens egy ablakon belül. A komponens neve egy azonosító, amellyel az alkalmazásban a komponensre hivatkozni tudunk.

A névvel ellentétben a komponens felirata (**Caption** tulajdonság) bármilyen lehet, tartalmazhat szóközöket, és lehet ugyanolyan is, mint egy másik komponensé. A felirat például az ablak tetején jelenik meg a címsorban (Form komponensnél), vagy egyenesen rajta a komponensen (Button). Felirattal nem lehet ellátni olyan komponenseket, melyeknél ennek nincs értelme (pl. görgetősáv-nak nincs felirata).

A felirat segítségével lehet beállítani a komponens gyors elérését is a felhasználó számára. Ha a komponens feliratában valamelyik betű elé **&** jelet teszünk, akkor ez a betű a feliratban alá lesz húzva, és a felhasználó ezt a komponenst kiválaszthatja az **Alt + aláhúzott betű** billentyűkombináció segítségével. Ha a feliratban az & jelet szeretnénk megjeleníteni, meg kell azt dupláznunk (&&).

7.2 A komponens mérete és elhelyezkedése

A komponens elhelyezkedését a **Left** (bal szélétől) és **Top** (tetejétől) tulajdonságok adják meg. A tulajdonságok a koordinátákat nem az egész képernyőhöz viszonyítva tartalmazzák, hanem a tulajdonoshoz (szülőhöz) viszonyítva. Ha például egy nyomógombot helyezünk el közvetlenül az ablakunkon (form-on), akkor a tulajdonosa az ablak (form) és ennek bal felső sarkához képest van megadva a nyomógomb elhelyezkedése (Left és Top tulajdonsága).

A komponens méretét a **Width** (szélesség) és **Height** (magasság) tulajdonsága határozza meg. Hasonlóan a Left és Top tulajdonságokhoz az értékük képpontokban (pixelekben) van megadva.

Néhány komponensnél beállíthatjuk, hogy a komponens mindig az ablak (form) valamelyik részéhez illeszkedjen (ragadjon). Ezt az **Align** tulajdonság segítségével tehetjük meg. Ennek megadásával a komponenst nem fogjuk tudni onnan leválasztani, az ablak átméretezésénél is ott marad az ablak teljes szélességében (ill. magasságában).

De mit tehetünk, ha a komponenst valamilyen kis távolságra szeretnénk elhelyezni a form szélétől úgy, hogy mindig ugyanakkora távolságra legyen tőle, az ablak átméretezésekor is? Erre szolgál az **Anchors** tulajdonság. Segítségével megadhatjuk, hogy a komponens a form melyik széléhez (vagy széleihez) illeszkedjen.

Az utolsó mérettel és elhelyezkedéssel kapcsolatos érdekes tulajdonság a **Constrains**. Ennek a tulajdonságnak négy altulajdonsága van, melyek segítségével megadhatjuk a komponens lehetséges minimális és maximális méretét. Ha például beállítjuk ezt a tulajdonságot egy alkalmazás ablakánál, akkor az ablakot az alkalmazás futtatásakor nem lehet majd a megadott méretnél kisebbre, illetve nagyobbra méretezni.

7.3 A komponens engedélyezése és láthatósága

A komponens engedélyezését az **Enabled** tulajdonság segítségével tudjuk beállítani. Alapértelmezésben ez mindig igaz (true). Ha átállítjuk hamisra (false), tervezési módban nem történik látszólag semmi, de az alkalmazás futásakor a komponens „szürke” lesz és nem reagál majd a rákattintásra.

A másik hasonló tulajdonság a **Visible**. Segítségével beállíthatjuk, hogy a komponens látható legyen-e az alkalmazás futásakor. Az alapértelmezett értéke ennek a tulajdonságnak is igaz (true). Tervezési időben itt sem fogunk látni különbséget, ha átállítjuk hamisra (false), csak az alkalmazás futtatásakor vehetjük majd észre, hogy a komponens nem látható.

Programunkban ahol lehet, inkább használjuk csak az **Enabled** tulajdonságot, mivel a felhasználóknak zavaró lehet, ha például nyomógombok tűnnek el és jelennek meg. Sokkal áttekinthetőbb a felhasználó számára, ha az alkalmazásunk éppen nem állítható (a felhasználó számára nem elérhető) komponensei szürkék, tehát nem használhatók, de a helyükön vannak és láthatók.

Megjegyzés: Ha a **Visible** tulajdonság értéke igaz egy komponensnél, az még nem jelenti feltétlenül azt, hogy a komponensünk látható a képernyőn. Ha ugyanis a komponens tulajdonosának (tehát amin a komponens van, pl. TPanel, TForm, stb.) a **Visible** tulajdonsága hamis, akkor sem a tulajdonos, sem a rajta levő komponensek nem láthatók. Ezért létezik a komponenseknek egy **Showing** tulajdonsága, amely egy *run-time* (csak futási időben elérhető) és *read-only* (csak olvasható) típusú tulajdonság. Ennek a tulajdonságnak az értéke megadja, hogy a komponensünk valóban látható-e a képernyőn.

7.4 A komponensek „Tag” tulajdonsága

A **Tag** tulajdonság (lefordítva: hozzáfűzött cédula, jel) a komponensek egy különös tulajdonsága. Ennek a tulajdonságnak a beállítása semmilyen hatással nem jár. Ez csak egy kiegészítő memóriaterület, ahol különféle felhasználói adatok tárolhatók. Alapállapotban ebben a tulajdonságban egy LongInt típusú értéket tárolhatunk. Szükség esetén áttipizálással bármilyen más 4 bájt hosszúságú értéket írhatunk bele (pl. mutatót, karaktereket, stb.).

7.5 Komponensek színe és betűtípusa

A komponensek **Color** (szín) és **Font** (betűtípus) tulajdonságaik segítségével beállíthatjuk a komponensek háttérszínét, ill. a komponenseken megjelenő feliratok betűtípusát (ha a komponensen megjeleníthető felirat).

A **Color** tulajdonság értékét megadhatjuk előre definiált konstansok segítségével: **clXXX** formában. Az XXX helyére vagy a szín nevét írhatjuk angolul (pl. **clRed**, **clGreen**, **clBlue**, stb.), vagy a *Windows* által definiált, a rendszerelemekre használt színek neveit (pl. **clBtnFace**, **clWindow**, stb.).

A színt ezeken a konstansokon kívül megadhatjuk az összetevőik (piros, zöld, kék) segítségével is. Ebben az esetben a szín megadására egy 4 bájtos hexadecimális számot használunk, melynek formája: **$AABBCCDD**, ahol:

- **AA** – a színpalettát határozza meg, ez általában 00,
- **BB** – a kék összetevő mennyiségét határozza meg,
- **CC** – a zöld összetevő mennyiségét határozza meg,
- **DD** – a piros összetevő mennyiségét határozza meg.

Például:

$00FF0000 – telített kék szín (clBlue),
$0000FF00 – telített zöld szín (clGreen),
$000000FF – telített piros szín (clRed),
$00000000 – fekete szín (clBlack),
$00FFFFFF – fehér szín (clWhite),
$00609025 – sötétzöld szín,
$003050A0 – barna szín, stb.

A **Font** tulajdonság értéke egy TFont típus lehet. A TFont osztály egyes elemeit beállíthatjuk az Objektum felügyelőben, ha a **Font** mellett rákattintunk a „**+**“ jelre.

Ha a program futása alatt szeretnénk beállítani a **Font** tulajdonság valamelyik elemét (altulajdonságát), például egy nyomógombon a betű méretét, azt a következő paranccsal tehetjük

meg: **Button1.Font.Size := 18;** A betű stílusát hasonlóan állíthatjuk be, csak ezt halmazként kell megadnunk, tehát ilyen formában: **Button1.Font.Style := [fsBold, fsItalic];**

A legtöbb komponens tartalmaz egy **ParentColor** (szülő színe) és egy **ParentFont** (szülő betűtípusa) tulajdonságot is. Ezekkel beállíthatjuk, hogy a komponens a tulajdonosának (ami leggyakrabban az alkalmazás ablaka - form) a színét és betűtípusát használja. Így be tudjuk egyszerre állítani az ablakunkon levő összes komponens színét és betűtípusát a form-unk **Font** és **Color** tulajdonságainak beállításával.

7.6 Komponens lebegő súgója

A komponens **Hint** (javaslat) tulajdonságának köszönhetően az objektum felett egérrel elhaladva egy sárga téglalapban információt közölhetünk a felhasználóval (ha megnyomja pl. a gombot, akkor mi fog történni). A kiírandó segítséget a komponens **Hint** tulajdonságához kell hozzárendelnünk (megadnunk az Objektum felügyelőben).

A komponens **ShowHint** (javaslatot megjelenít) tulajdonságával megadható, hogy ez a segítség megjelenjen-e a felhasználónak.

A **ParentShowHint** tulajdonsággal meghatározhatjuk, hogy a komponenshez a javaslat akkor jelenjen meg, ha a komponens tulajdonosának (ami általában a form) a ShowHint tulajdonsága igaz. Így egyetlen tulajdonság átállításával (a form ShowHint tulajdonságával) beállíthatjuk, hogy az ablak összes komponensére megjelenjen-e a javaslat vagy nem.

7.7 Az egérmutató beállítása

Sok komponens rendelkezik **Cursor** (egérmutató) tulajdonsággal. Ennek segítségével beállíthatjuk, hogy az egérmutatónak milyen alakja legyen, ha az adott komponens felett áll. Lehetséges értékek: **crHourGlass** (homokóra), **crCross** (kereszt), **crHelp** (nyíl kérdőjellel), **crUpArrow** (felfelé mutató nyíl), stb.

7.8 Tabulátor

Ha az alkalmazásunknak több komponense van, jó ha intelligensen működik a TAB billentyű. Azt, hogy a TAB billentyű megnyomásakor milyen sorrendben legyenek aktívak a komponensek a **TabOrder** (TAB sorrend) tulajdonság segítségével állíthatjuk be. Ide egy számot kell beírnunk, amely azt jelenti, hányadik lesz a komponens a sorrendben. A számozás 0-tól kezdődik.

A **TabStop** (TAB áljon meg) tulajdonság segítségével beállíthatjuk, hogy az adott komponensre lehet-e egyáltalán a tabulátor segítségével eljutni (ha a TabStop értéke igaz, akkor lehet, ha hamis, akkor nem lehet – a tabulátor nem áll meg a komponensen, hanem a sorban következőre ugrik át).

8 Események

A legtöbb komponensnél nem elég, ha csak a tulajdonságait állítjuk be. Sokszor szükségünk van rá, hogy az adott komponens valamilyen tevékenységet végezzen, ha pl. rákattintunk egérrel, megnyomunk egy billentyűt, mozgatjuk felette az egeret, stb. Erre szolgálnak az események. Ahhoz, hogy egy eseményere a komponens úgy reagáljon, ahogy mi azt szeretnénk, meg kell írnunk az eseményhez tartozó programkódot (eljárást).

Hasonlóan, ahogy a komponenseknek vannak olyan **tulajdonságaik**, amelyek szinte minden komponensnél megtalálhatók, vannak olyan **események** is, melyek majdnem minden komponensnél előfordulnak. Ezek közül a legfontosabbak a következők:

Komponensek eseményeik:

Esemény	Mikor következik be	Megjegyzés
OnChange	Ha a komponens vagy annak tartalma megváltozik (pl. a szöveg az Edit komponensben).	Gyakran használatos az Edit és Memo komponenseknél. Összefügg a **Modified** tulajdonsággal (*run-time*, *read-only*), amely megadja, hogy a komponens tartalma megváltozott-e.

OnClick	A komponensre kattintáskor az egér bal gombjával.	Ez az egyik leggyakrabban használt esemény. Ez az esemény nem csak egérkattintáskor, hanem Enter, ill. Space billentyűk megnyomásakor is bekövetkezik, ha a komponens aktív (például egy aktív nyomógomb).
OnDblClick	A komponensre duplakattintáskor az egér bal gombjával.	Duplakattintáskor az első klikkelésnél **OnClick** esemény következik be, majd ha rövid időn belül (ahogy a *Windows*-ban be van állítva) érkezik második klikkelés is, akkor bekövetkezik az **OnDblClick** esemény.
OnEnter	Amikor a komponens aktiválva lett.	Itt nem az ablak (form) aktiválásáról van szó, amikor az egyik ablakból átmegyünk a másikba, hanem a komponens aktiválásáról, például ha Edit komponensbe kattintunk.
OnExit	Amikor a komponens deaktiválva lett.	Az előző esemény ellentettje. Például akkor következik be, ha befejeztük a bevitelt az Edit komponensbe és máshova kattintunk.

OnKeyDown	Amikor a komponens aktív és a felhasználó lenyom egy billentyűt.	Felhasználhatjuk az eljárás **Key** paraméterét, amely megadja a lenyomott billentyű virtuális kódját (virtual key codes). Továbbá a **Shift** paraméter (amely egy halmaz típusú) segítségével meghatározhatjuk, hogy le volt-e nyomva az Alt, Shift, vagy Ctrl billentyű (ssAlt, ssShift, ssCtrl). *Megjegyzés*: Ha azt szeretnénk, hogy a lenyomott billentyűt a form kapja meg (méghozzá a komponens előtt), és ne az éppen aktív komponens, akkor a form **KeyPreview** tulajdonságát át kell állítanunk igazra (true).
OnKeyPress	Amikor a komponens aktív és a felhasználó lenyom egy billentyűt.	A különbség ez előző eljárástól, hogy itt a **Key** paraméter char típusú, amely a lenyomott billentyűt ASCII jelét (betűt, számot, írásjelet) tartalmazza. Ez az esemény csak olyan billentyű lenyomásakor következik be, amelynek van ASCII kódja (tehát nem Shift, F1 és hasonlók).
OnKeyUp	Amikor a komponens aktív és a felhasználó felenged egy billentyűt.	A gomb felengedésénél jön létre, **Key** és **Shift** paramétere hasonló, mint az **OnKeyDown** eseménynél.
OnMouseDown	Amikor a felhasználó lenyomja valamelyik egérgombot.	Általában annak a komponensnek az eseménye, amely éppen az egérmutató alatt van.

OnMouseMove	Amikor a felhasználó megmozdítja az egeret a komponensen.	Hasonlóan az előzőhöz, annak a komponensnek az eseménye, amely éppen az egérmutató alatt van.
OnMouseUp	Amikor a felhasználó felengedi valamelyik egérgombot.	Ha több egérgomb van lenyomva, akkor mindegyik felengedésénél létrejön ez az eljárás.

Ablak (form) eseményei:

Esemény	Mikor következik be	Megjegyzés
OnActivate	Amikor az ablak aktívvá válik.	Akkor van generálva ez az eljárás, ha a felhasználó egy másik ablakból (vagy alkalmazásból) erre az ablakra klikkel.
OnDeactivate	Amikor az ablak inaktívvá válik.	Ha a felhasználó egy másik ablakra (vagy alkalmazásra) klikkel, tehát elhagyja a mi ablakunkat.

OnCloseQuery, OnClose	Ha az ablakot bezárjuk (Alt-F4, X a jobb felső sarokban, rendszermenü segítségével, stb.).	Az ablak bezárásakor először az **OnCloseQuery** esemény következik be, utána az **OnClose**. Az első esemény szolgálhat megerősítésre (pl. „Biztos hogy kilépsz?”) vagy az adatok elmentésének figyelmeztetésére. Az alkalmazás bezárásának elkerülésére még az **OnClose** eseménynél is van lehetőségünk. Itt a paraméterben megadhatjuk azt is, hogy az ablakunk bezárás helyett csak elrejtve vagy minimalizálva legyen.
OnCreate, OnDestroy	Az ablak létrehozásakor ill. megszüntetésekor.	Az **OnCreate** esemény kezelésében lehetőségünk van dinamikusan létrehozni objektumok, melyeket ne felejtsünk el megszüntetni az **OnDestroy** eljárás kezelésében.
OnShow, OnHide	Az ablak megmutatásakor, ill. elrejtésekor.	Ezek az eljárások szorosan összefüggenek az ablak **Visible** tulajdonságával.

Látható ablakok (melynek a visible tulajdonságuk igaz) létrehozásakor az események bekövetkezéseinek a sorrendje a következő: **OnCreate, OnShow, OnActivate, OnPaint**.

9 Hibakeresés

Mindenekelőtt készítsünk egy egyszerű programot, amelyen bemutatjuk a hibakeresést. Pelda02

A programon két nyomógomb (*Számítások* és *Kilépés* felirattal) és egy címke legyen. A *Kilépés* megnyomásakor fejeződjön be az alkalmazás (**Form1.Close;**) a *Számítások* gomb megnyomásakor pedig a következő számítás menjen végbe, melynek végeredményét kiíratjuk a címkébe:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i,j: integer;
begin
  j:=0;
  for i:=1 to 10 do
    j:=j+i;
  Label1.Caption:=IntToStr(j);
```



```
end;
```

Ha ezt az alkalmazást elmentjük, majd lefordítjuk és futtatjuk, helyesen fog működni. Ilyen ideális eset azonban ritkán fordul elő. Ezért a *Delphi* tartalmaz egy **integrált debugger**-t rengetek eszközzel hibák megkeresésére. Mi ezek közül fogjuk bemutatni a leggyakrabban használtakat.

A programunkban előforduló hibákat durván két csoportra oszthatjuk:

- olyan hibákra, melyeket a fordító kijelez (ide tartoznak a szintaktikai hibák – elírt parancsok, és a szemantikai hibák – parancsok logikailag rossz sorrendbe használata),
- és olyan hibákra melyeket a fordító nem jelzi (logikai hibák).

Azokkal a hibákkal, melyeket a **fordító kijelez**, most nem fogunk foglalkozni. Az ilyen hiba esetében a program nem fut le, a kurzor pedig mindig a hibás sorban áll és megjelenik egy hibaüzenet. Ha rákattintunk a hibaüzenetre és megnyomjuk az F1 funkcióbillentyűt, elolvashatjuk a hiba részletes leírását.

Nehezebb azonban megtalálni az olyan hibákat, melyeket a **fordító nem jelez**. Az ilyen hibáknál a program elindul és mi abban a meggyőződésben élünk, hogy a programunk hiba nélkül fut. Némely esetben azonban előfordulhat, hogy például a számítások eredményeként, nem a helyes eredményt kapjuk. Ilyenkor használhatjuk a hibakeresésre szolgáló eszközöket, melyeket a menüben a **Run** alatt találunk. Ezek közül a leggyakrabban használtak:

Trace Into lépegetés

Ezzel az eszközzel lépegetni tudunk soronként az alkalmazásunkban. Egyszerűbben az **F7** funkcióbillentyűvel indíthatjuk el, illetve léphetünk tovább a következő sorra. Ha alprogram hívásához érünk, beleugrik az alprogramba és ott is soronként lépeget tovább.

Step Over lépegetés

Hasonló az előző eszközhöz annyi különbséggel, hogy ha alprogram hívásához érünk, nem ugrik bele az alprogramba, hanem azt egy blokként (egy lépésben) elvégzi. Egyszerűbben **F8** funkcióbillentyűvel érhetjük el.

Run to Cursor

Ha ráállunk a kurzorral valamelyik sorra a forráskódban és ezzel (vagy egyszerűbben az **F4** funkcióbillentyűvel) indítjuk el, a program hagyományos módon elindul és fut mindaddig, amíg ahhoz a sorhoz nem ér, ahol a kurzorunk áll. Itt leáll a program, és innen lépegethetünk tovább a fent említett eszközökkel.

Breakpoints (Add Breakpoint – Source Breakpoint…)

A breakpoint (megszakítás pontja) segítségével a *Delphi*-nek megadhatjuk, hogy a programunk melyik pontján álljon meg.

Gyakorlatban: ráállunk valamelyik sorra a forráskódban, kiválasztjuk a menüből a **Run – Add Breakpoint – Source Breakpoint…** menüpontot, majd „Ok” (vagy rákattintunk a sor elején a

kék körre –). Ekkor a kijelölt sor háttere átszíneződik, és a sor előtt egy piros kör jelenik meg (). Ez jelenti azt, hogy a program ebben a sorban le fog állni. A programot utána elindítjuk a **Run – Run** (vagy F9) segítségével. Ha a program a futása során breakpoint-hoz ér, leáll. Innen lépegethetünk tovább egyesével az első két eszköz segítségével (F7, F8), vagy futtathatjuk tovább a programot a **Run – Run** (vagy F9) segítségével. Egy programban több breakpoint-ot is elhelyezhetünk.

A breakpoint-ot a forráskódban a sor elején található piros körre () kattintva szüntethetjük meg.

Watch (Add Watch…)

A program lépegetése közben ennek az eszköznek a segítségével megfigyelhetjük az egyes változók értékét.

A változók értékeit a **Watch List** ablakban követhetjük nyomon (ez az ablak automatikusan megjelenik a program indításakor, de ha mégsem jelenne meg a View – Debug Windows – Watches menüvel hívhatjuk elő).

Új változót vagy kifejezést a **Run – Add Watch…** (**CTRL**+**F5**) menüpont segítségével adhatunk a megfigyelt változók közé (a Watch List-be).

Gyakorlatban ezt úgy használhatjuk, hogy kijelölünk a programban Breakpoint-ot, ahonnan a változókat figyelni szeretnénk, vagy odaállunk a kurzorral és elindítjuk a programot a „Run to Cursor” segítségével. Majd az „Add Watch…” (vagy CTRL+F5) segítségével beállítjuk a figyelni kívánt változókat és elkezdünk lépegetni a „Trace Into” ill. a „Step Over” segítségével. Közben figyelhetjük a kiválasztott változók értékeit.

Evalute / Modify

Ennek az eszköznek a segítségével nem csak megfigyelhetjük, de **meg is változtathatjuk** a kifejezések, változók vagy tulajdonságok értékeit. Ez egy nagyon hasznos eszköz, ha arra vagyunk kíváncsiak, hogyan viselkedne a program, ha például az „i” változóban nem 7, hanem 1500 lenne. Ezt az eszközt egyszerűbben a **CTRL**+**F7** funkcióbillentyűvel hívhatjuk elő.

Program Reset

Előfordulhat, hogy a programunk lefagy, vagy csak egyszerűen olyan helyzetbe kerülünk, hogy a programunk futását le szeretnénk állítani, majd elölről futatni. Ebben az esetben hívhatjuk meg a **Run – Program Reset** menüpontot (vagy **CTRL**+**F2**).

10 Nagyobb projektek készítése

Ebben a fejezetben nagyobb projektek készítésének alapelveiről lesz néhány szó. Az itt felsorolt módszerek csak javaslatok, nem szükséges ezek szerint írni az alkalmazásunkat, de ezek betartásával sokkal áttekinthetőbb, olvashatóbb és érthetőbb lesz a projektünk.

Komponensek megnevezése

Ha komolyabb alkalmazást készítünk, nem jó ötlet a komponenseknek meghagyni azokat a nevüket, melyeket a *Delphi* automatikusan rendel hozzájuk. Kisebb alkalmazás készítésénél ez lényegtelen, viszont ilyent csak ritkán készítünk. A legjobb a komponenseket megnevezni valamilyen áttekinthető sablon alapján. Nyomógombot például **btnXXX**-nek nevezhetünk el, ahol **XXX** a nyomógomb funkcióját írja le, például: **btnKilepes**, **btnSzamitasok**, stb. Az ablakunkat (form-ot) legjobb **frmXXX**-nek elnevezni (vagy talán még jobb, ha **wndXXX**-nek nevezzük el), a beviteli mezőt megnevezhetjük **edtXXX**-nek, a képet **imgXXX**-nek. A lényeg, hogy a program könnyen áttekinthető, könnyen olvasható legyen mások számára is és főleg saját magunknak is, ha majd bizonyos (hosszabb) idő elteltével újra át szeretnénk nézni.

Forráskód külalakja

Az alábbi javaslatok betartásával olvasható és áttekinthető forráskódot tudunk majd írni:

- **Nagy és kisbetűk** – a Delphi (Pascal) nem case-sensitive programozási nyelv. Ennek ellenére jó, ha a nagy és kisbetűk használatában rendet tartunk és követünk valamilyen logikát. Például a „btnKilepesClick” sokkal áttekinthetőbb, mint a „btnkilepesclick” vagy a „BTNKILEPESCLICK”).
- **Megjegyzések** – hasznos megjegyzések gyakori használatával az alkalmazásunkban sok időt és problémát spórolhatunk meg magunknak a jövőben. Megjegyzést a forráskódba tehetünk {kapcsos zárójelek} közé vagy két törtvonal // segítségével a sor elején.
- **Bekezdések, üres sorok, szóközök** – ne spóroljunk az üres sorokkal és a bekezdésekkel (beljebb írásokkal), szóközökkel a programunkban. Ezek megfelelő használatával programunk sokkal áttekinthetőbb lesz.

11 Standard üzenetablakok

Az alkalmazásunkban nagyon sokszor előfordulhat, hogy a felhasználót értesíteni szeretnénk például a számítások állapotáról, figyelmeztetni a hibákra vagy a rosszul megadott bemeneti értékekre, megkérdezni tőle, hogy biztos ki akar-e lépni, akarja-e menteni a dokumentumot, stb. Az ilyen esetekre a *Delphi* egy elegáns megoldással rendelkezik: a **standard üzenetablakokkal**. Ezek használata nagyon egyszerű, mégis a beállítások és megjelenítések széles választékával rendelkezik.

11.1 ShowMessage

Ha csak egy egyszerű szöveget szeretnénk kiíratni üzenetablakban a felhasználónak, akkor használhatjuk a **ShowMessage** eljárást. Ez a legegyszerűbb standard üzenetablak. Szintaxisa:

```
procedure ShowMessage(const Msg:string);
```

Például:

```
ShowMessage('Ez egy rövid kis információ.');
```

Az üzenetablak felirata (nálunk „Project1”) ugyanaz, mint az alkalmazás futtatható (exe) állományának a neve.

11.2 MessageDlg

Az előző eljárásnál többet tud a **MessageDlg** függvény. Ezzel a függvénnyel az üzenetablakunk külalakját jelentős mértékben formálhatjuk. Szintaxisa:

```
function MessageDlg(const Msg:string;
      DlgType: TMsgDlgType;
      Buttons: TMsgDlgButtons;
      HelpCtx: Longint): Word;
```

A paraméterek leírása:

- **Msg**: a szöveg, amit meg szeretnénk jeleníteni
- **DlgType**: az üzenetablak célját jelzi. Lehetséges értékek:
  - **mtWarning** – figyelmeztetést jelző sárga-fekete ikon
  - **mtError** – hibát jelző piros „stoptábla”
  - **mtInformation** – információt jelző kék „i” betű
  - **mtConfirmation** – kérdést jelző kék kérdőjel
  - **mtCustom** – az üzenetablakon nem lesz kép
- **Buttons**: indikálja, hogy melyik gombok jelenjenek meg az üzenetablakon. Lehetséges értékek:
  - **mbYes**, **mbNo**, **mbOK**, **mbCancel**, **mbAbort**, **mbRetry**, **mbIgnore**, **mbAll**, **mbNoToAll**, **mbYesToAll**, **mbHelp**

Figyelem: Itt egy halmazt kell megadnunk, ezért a kiválasztott nyomógombokat szögletes zárójelek között kell felsorolnunk, például: [mbAbort, mbRetry, mbIgnore]. Ez alól egyedüli kivétel, ha valamelyik előre definiált konstanst használjuk (például az mbOKCancel ugyanazt jelenti, mint az [mbOk, mbCancel]).

- **HelpCtx**: a súgó azon témájának „Context ID”-je, amely megjelenjen, ha megnyomjuk az F1 billentyűt. Ha ezt nem akarjuk használni (nincs saját súgó fájlunk), adjunk meg 0-t.

A **MessageDlg** visszaadja annak a nyomógombnak az értékét, amellyel a felhasználó bezárta az üzenetablakot. Lehetséges értékek: **mrNone**, **mrAbort**, **mrYes**, **mrOk**, **mrRetry**, **mrNo**, **mrCancel**, **mrIgnore**, **mrAll**.

Például:

```
if MessageDlg('Elmenteni a fájlt?',
      mtConfirmation, [mbYes, mbNo, mbCancel],
      0) = mrYes then mentsd_el(fajlnev);
```

11.3 MessageDlgPos

Az előző függvény egyik hátránya, hogy az üzenetablak mindig a képernyő közepén jelenik meg. Ez néha nem megfelelő. Szerencsére a *Delhi*-ben létezik az előző függvénynek egy kibővített változata, a

MessageDlgPos, melynek ugyanolyan paraméterei vannak, mint a MessageDlg függvénynek, plusz még további két paramétere mellyel megadhatjuk az üzenetablak helyét a képernyőn. Szintaxisa:

```
function MessageDlg(const Msg:string;
      DlgType: TMsgDlgType;
      Buttons: TMsgDlgButtons;
      HelpCtx: Longint;
      X, Y: Integer): Word;
```


12 Információk bevitele

Az információkat a programunkba komponensek segítségével vihetjük be. Ebben a fejezetben veszünk néhány példát ezek használatára és megismerkedünk az egyes komponensek további (egyéni) tulajdonságaival, eseményeivel és metódusaival (metódus = olyan függvények és eljárások, melyek csak valamely komponensekre, pontosabban valamely osztály objektumaira vonatkoznak).

Igaz, hogy ennek a fejezetnek a címe „információk bevitele”, de sok komponens ugyanúgy használható bevitelre, mint adatok megjelenítésére (kimenetre). Ezért ne olvassuk ezt a fejezetet úgy, hogy itt csak azok a komponensek szerepelnek, mellyel adatokat vihetünk be a programunkba. Az itt tárgyalt komponensek szinte mindegyike ugyanúgy kimenetre is szolgálhat, mint bemenetre.

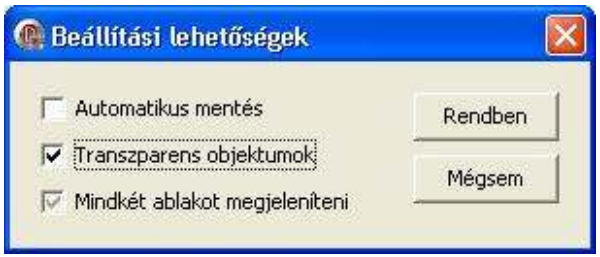
Kezdjük a legegyszerűbb beviteli komponensektől, melyekkel logikai értéket (igen-nem, igaz-hamis, 0-1) vihetünk be az alkalmazásunkba. Logikai értékek bevitelére használhatjuk a már említett üzenetablakokat is, azonban ez nagyon sokszor nem megfelelő. Ennél sokkal megfelelőbb a **CheckBox** (jelölőnégyzet) használata.

12.1 Jelölőnégyzet használata – CheckBox

Előnye, hogy állandóan látható ki van-e jelölve, egyetlen kattintással kijelölhető, áttekinthető és a felhasználónak nem ugrálnak elő állandóan ablakok (mint ahogy az lenne az üzenetablakok használatánál ez helyett).

A **CheckBox** fontosabb tulajdonságai:

- **AllowGrayed** – ha ennek a tulajdonságnak az értéke igaz (true), akkor a jelölőnégyzetnek három lehetséges értéke lehet: Checked (kipipálva), Unchecked (nincs kipipálva), Grayed (szürke). Egyébként csak két értéke lehet.
- **Caption** – felirat, amely a jelölőnégyzet mellett szerepeljen.
- **Checked** – megadja hogy ki van-e pipálva a jelölőnégyzet (true) vagy nincs kipipálva (false).
- **State** – hasonló az előzőhöz, de ennek értéke háromféle lehet: cbChecked, cbUnchecked, cbGrayed.



Nagyon egyszerűen tudjuk a jelölőnégyzetek értékeit kiolvasni. Például, meg szeretnénk tudni, hogy a felhasználó kijelölte-e az első jelölőnégyzetet (és ha igen, akkor el szeretnénk végezni valamilyen műveletet). Ezt a következőképpen tehetjük meg:

```
if CheckBox1.Checked = true then …
```

Vagy használhatjuk ugyanezt „rövidített” változatban:

```
if CheckBox1.Checked then …
```

Az adatok felhasználótól való beolvasásán kívül természetesen nagyon jól használható a CheckBox logikai értékek megjelenítésére is.

12.2 Választógomb – RádioButton

Ez a komponens általában csoportokban fordul elő, mivel itt a csoporton belül mindig csak egyet lehet kiválasztani. Az alkalmazás indításakor megoldható, hogy a csoporton belül egy gomb se legyen kiválasztva, de ez csak ritkán szokott előfordulni. Ha már ki van választva egy, kiválaszthatunk egy másikat, de nem szüntethetjük meg a kiválasztást, tehát egyet mindenképpen ki kell választanunk. Az egyik legfontosabb tulajdonsága:

- **Checked** – értéke (true/false) megadja, hogy a gomb ki van-e választva.

A RadioButton komponensek szinte mindig valamilyen logikai csoportot összekapcsoló komponensen vannak rajta (GroupBox, Panel komponenseken), de lehet közvetlenül a form-on (ablakunkon) is. Több RadioButton komponens használata helyett azonban jobb használni inkább egy RadioGroup komponens.

12.3 Választógomb csoport – RadioGroup

A RadioGroup komponens legfontosabb tulajdonságai:

- **Items** – értéke TStrings típus lehet. Ennek segítségével adhatjuk meg, milyen választógombok szerepeljenek a komponensünkön (tehát miből lehessen választani). Az egyes lehetőségek neveit külön-külön sorban adjuk meg. A „karikákat” a RadioGroup komponens kirakja automatikusan mindegyik sor elé.

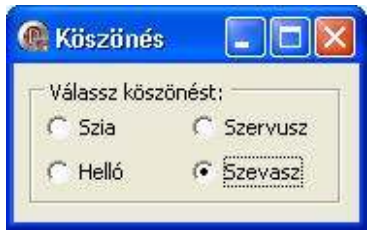
- **Columns** – segítségével megadhatjuk, hogy a választási lehetőségek (választógombok) hány oszlopban legyenek megjelenítve.
- **ItemIndex** – ennek a tulajdonságnak az értéke tartalmazza, hogy melyik választógomb (lehetőség) van kiválasztva. Ha értéke -1 akkor egyik sincs kiválasztva, ha 0 akkor az első, ha 1 akkor a második, stb. választógomb van kijelölve (tehát a számozás 0-tól kezdődik).

Azt, hogy melyik nyomógomb van kiválasztva, az **ItemIndex** tulajdonság tesztelésével vizsgálhatjuk:

```
if RadioGroup1.ItemIndex = 0 then …
```

Ez a tesztelés azonban nagyon sok programozónak nem a legmegfelelőbb, mivel meg kell jegyezni, melyik lehetőséghez melyik szám tartozik. Ezen könnyíthetünk, ha például a számok helyett konstansokat definiálunk (const SZIA = 0; HELLO = 1; …) és ezek segítségével teszteljük a feltételt:

```
if RadioGroup1.ItemIndex = SZIA then …
```



12.4 Beolvasás „üzenetablak” segítségével

Egysoros szöveg beolvasásához használhatunk egy újabb „üzenetablakot”, az **InputBox**-ot:

```
function InputBox(const ACaption, APrompt,
                  ADefault: string): string;
```

Az egyes paraméterek leírása:

- **ACaption**: a dialógusablak felirata,
- **APrompt**: a dialógusablakban megjelenő szöveg,
- **ADefault**: a beviteli mezőben megjelenő kezdeti szöveg.

Például:

```
nev := InputBox('Név megadása',
                'Kérlek add meg a neved:', '');
```

Természetesen ennél az eszköznél sokkal jobban felhasználható egysoros szöveg bevitelére az Edit komponens.

12.5 Egysoros szöveg beviteli doboz – Edit

Az Edit komponensnek sok specifikus tulajdonsága van, melyek segítségével az egysoros bevitelt korlátozhatjuk, vagy formátozhatjuk (például megadhatjuk egyetlen tulajdonság beállításával, hogy a bevitt szöveg helyett csillagocskák vagy más jelek jelenjenek meg – igy használhatjuk jelszó bevitelére is).

Az Edit komponens legfontosabb tulajdonságai:

- **Text** – ez a tulajdonság tartalmazza a beviteli mezőben megjelenő szöveget. Segítségével kiolvashatjuk, vagy beállíthatjuk az Edit komponensben levő szöveget.
- **MaxLength** – az Edit-be megadható szöveg maximális hossza. Segítségével beállíthatjuk milyen hosszú szöveget adhat meg a felhasználó.
- **Modified** – annak megállapítására szolgáló tulajdonság, hogy a beviteli mezőben történt-e változás.
- **AutoSelect** – segítségével beállíthatjuk, hogy a beviteli mezőbe lépéskor (kattintáskor) ki legyen-e jelölve az egész szöveg. Ezt az szerint adjuk meg, hogy a felhasználó a mező értékét előreláthatóan új értékkel fogja helyettesíteni, vagy csak az ott levő értéket fogja módosítani.
- **ReadOnly** – meghatározza, hogy a felhasználó megváltoztathatja-e a beviteli mezőben levő értéket.
- **PasswordChar** – ennek a tulajdonságnak az értéke egy karakter lehet, amely meg fog jelenni a bevitelnél a bevitt karakterek helyett. Jelszó bevitelénél használhatjuk.

Az Edit komponens több metódussal is rendelkezik, melyek közül az egyik:

- **CopyToClipboard** – vágólapra másolja a beviteli mezőben levő szöveget.

Az Edit komponens használata nagyon egyszerű elég elhelyezni a komponenst valahova a form-on. Ajánlott a komponens fölé elhelyezni mindjárt egy Label-t is, amely segítségével megadjuk mit kérünk a felhasználótól. Utána már csak pl. egy nyomógomb OnClick eseményében az Edit1.Text tulajdonság értékét ellenőrizzük (ha szükséges) és felhasználjuk (pl. valahova máshova berakjuk, stb.).

Sokkal érdekesebb a helyzet, ha a felhasználótól csak valamilyen számot szeretnénk beolvasni. Erre több megoldásunk is lehet:

Megengedjük, hogy a felhasználó beadhasson szöveget is, majd azt számmá alakítjuk. Ennek a hátránya, hogy a felhasználó beadhat betűket is, nem csak számokat. Ezek átalakításánál számmá természetesen a programban hiba következik be, melyet vizsgálnunk kell (pl. az átalakításhoz a Val függvényt használjuk, mely harmadik paraméterében visszaadja, történt-e hiba az átalakításnál). Például:

```
val(Edit1.Text, k, hiba);
if hiba<>0 then
  begin
    Edit1.SetFocus;
    MessageDlg('Csak számot adhatsz meg!',
               mtError, [mbOk], 0)
  end;
```

A SetFocus metódus a példában csupán arra szolgál, hogy az Edit1-et teszi aktívvá, tehát ahhoz, hogy új adatot adhasson meg a felhasználó, nem kell először rákattintania, mindjárt oda írhatja a számokat.

Másik megoldás, hogy **a bemenetnél kiszűrjük azokat a karaktereket, melyek nem számok.** Ennél azonban nagyon figyelmesen kell eljárnunk, ugyanis nem elég ha az OnKeyPress eseményben kiszűrjük a nem megfelelő karaktereket, mivel a felhasználó a vágólapról való bemásolással továbbra is tud szöveget beilleszteni (tehát figyelnünk kell pl. az OnChange eseményt is). Példa a billentyűzetről való kiszűrésre az OnKeyPress eseményben:

```
if not (Key in ['0'.. '9 ', #8]) then
  begin
    Key := #0;
    MessageBeep($FFFFFFFF);
  end;
```

A MessageBeep egy Windows API függvény, ezért a súgó *Delphi*-hez tartozó részében nem található meg, csak a „Microsoft Platform SDK" részben (ha ez is be van telepítve). A függvény a Windows-ban beállított, eseményekhez definiált hangok lejátszására szolgál. Paraméterének lehetséges értékei: $FFFFFFFF, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, MB_OK.

12.6 Többsoros szöveg beviteli doboz – Memo

A Memo komponens segítségével az Edit-hez hasonló, de többsoros szöveget olvashatunk be.

A Memo legfontosabb tulajdonságai:

- **Alignment** – a sorok igazításának beállítására használható tulajdonság. Segítségével a sorokat igazíthatjuk balra, jobbra vagy középre.
- **ScrollBars** – tulajdonsággal megadhatjuk, hogy a komponensen a vízszintes, a függőleges vagy mindkettő görgetősáv jelenjen-e meg, vagy ne legyen egyik görgetősáv se a komponensen.
- **WordWrap** – igaz (true) értéke az automatikus sortördelést jelenti (ha ez a tulajdonság igaz, nem lehet „bekapcsolva" a vízszintes görgetősáv, mivel ez a kettő tulajdonság kölcsönösen kizárja egymást).
- **WantTabs**, **WantEnter** – a Tab és Enter billentyűknek minden komponenseknél más funkciójuk van. A tabulátor megnyomására általában a következő komponens lesz aktív a form-on. Ha szeretnénk, hogy a felhasználó a Memo komponensben használni tudja a Tab és Enter billentyűket, tehát hogy tudjon tabulátorokat (bekezdéseket) és új sorokat létrehozni a szövegben, ezeknek a tulajdonságoknak az értékeit igazra (true) kell állítanunk. Ha ezt nem tesszük meg, a felhasználó akkor is tud létrehozni bekezdéseket és új sorokat a Ctrl+Tab és Ctrl+Enter billentyűzetkombinációk segítségével.
- **Text** – hasonló tulajdonság, mint az Edit komponensnél. A Memo komponensben levő szöveget tartalmazza.
- **Lines** – segítségével a Memo-ba beírt szöveg egyes soraival tudunk dolgozni. Ez a tulajdonság TString típusú,

melynek sok hasznos tulajdonsága és metódusa van. Például a Lines (TString) **LoadFromFile** és **SaveToFile** metódusaival be tudunk olvasni a merevlemezről ill. el tudunk menteni a merevlemezre szöveget.

A Lines tulajdonság használatát mutatja be a következő példa is:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('c:\autoexec.bat');
  ShowMessage('A 6. sor: ' + Memo1.Lines[5]);
end;
```

12.7 Görgetősáv - ScrollBar

Gyakori probléma a számok bevitele a programba. Számok bevitelére használhatjuk a már említett Edit vagy Memo komponenseket

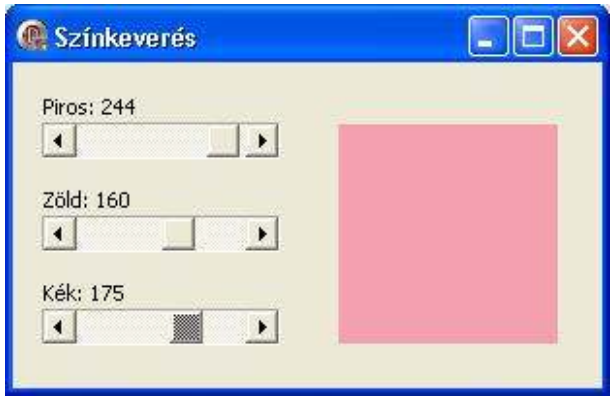
is. Most azt mutatjuk be, hogyan vihetünk be számokat a **ScrollBar** komponens segítségével.

A ScrollBar legfontosabb tulajdonságai:

- **Kind** – meghatározza, hogy a komponens vízszintesen (sbHorizontal) vagy függőlegesen (sbVertical) helyezkedjen el. Az új ScrollBar kezdeti beállítása mindig vízszintes.
- **Min**, **Max** – meghatározza a határértékeket.
- **Position** – meghatározza a csúszka aktuális pozícióját (aktuális értéket).
- **SmallChange** – az eltolás mértéke, ha a görgetősáv szélein levő nyilakra kattintunk, vagy a billentyűzeten levő nyilak segítségével állítjuk be.
- **LargeChange** – az eltolás mértéke, ha a görgetősáv sávjában kattintunk valahova, vagy a PageUp, PageDown billentyűket nyomjuk meg.

Ha meg szeretnénk határozni azt az értéket, amelyet a felhasználó beállított a görgetősávon, azt legegyszerűbben az OnChange eseményben vizsgálhatjuk.

Ha igazán szép görgetősávot akarunk létrehozni, akkor a görgetősáv mellé (vagy elé) tegyünk egy címkét (Label) is, amely folyamatosan a csúszka aktuális pozíciójának értékét mutatja.



Ebben a példában az **RGB** Windows API funkcióját használtuk a három színből a végső szín meghatározására. Enne a függvénynek a szintaxisa:

```
function RGB(red, green, blue: byte): cardinal;
```

ahol red, green, blue az alapszínek (piros, zöld, kék) mennyiségét jelölik, mindegyik értéke 0-tól 255-ig lehet. Ezeket az értékeket (0, 255) megadtuk mindegyik görgetősáv Min (0) és Max (255) tulajdonságában. Az egyes görgetősávok OnChange eseményeinek programkódjai durván a következők:

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Label1.Caption := 'Piros: ' +
                    IntToStr(ScrollBar1.Position);
  Label4.Color := RGB(ScrollBar1.Position,
                      ScrollBar2.Position,
                      ScrollBar3.Position);
end;
```


Ebben a programkódban használtuk az **IntToStr** ill. **StrToInt** függvényeket. Ezek egész számot alakítanak át szöveggé ill. fordítva. Szintaxisuk:

```
function IntToStr(Value: integer): string;
function StrToInt(const S: string): integer;
```

Ha az utóbbi függvényben az S paraméter nem számot tartalmaz (hanem betűket is), akkor az átalakítás során az EConvertError kivétel következik be. A kivételekről még lesz szó a későbbiekben, ezért itt most csak a használatát ismerjük meg:

```
try
  ertek := StrToInt(szöveg);
except
  on EConvertError do …
end;
```

12.8 Szám bevitele – SpinEdit segítségével

A SpinEdit komponens szintén számok bevitelére szolgál. A számot megadhatjuk billentyűzet segítségével és egér segítségével is a komponens szélén levő fel-le nyilakra kattintva.

Legfontosabb tulajdonságai:

- **Value** – meghatározza a beadott (kiválasztott) értéket.
- **MinValue** – meghatározza a minimum értéket, amit a felhasználó megadhat a SpinEditbe.
- **MaxValue** – segítségével megadhatjuk a maximum értéket, amit a felhasználó megadhat a SpinEditbe.

- **Increment** – megadja, hogy a jobb szélén levő nyilakra kattintva mennyivel növekedjen ill. csökkenjen a SpinEdit aktuális értéke.

12.9 Listadoboz – ListBox

A klasszikus listadoboz (ListBox) az egyik leggyakrabban használt kimeneti komponens.

Legfontosabb tulajdonságai:

- **Columns** – oszlopok száma, melyekben az adatok meg lesznek jelenítve.
- **Items** – a legfontosabb tulajdonság, a lista egyes elemeit tartalmazza. Ez is TString típusú, hasonlóan a Memo komponens Lines tulajdonságához, és mint olyannak, rengeteg hasznos metódusa van.
- **ItemIndex** – az éppen kiválasztott elem sorszáma. A számozás 0-tól kezdődik. Ha nincs kiválasztva egyik eleme sem a listának, akkor az ItemIndex értéke -1.
- **MultiSelect** – egyszerre több érték (elem) kiválasztását engedélyezi (true) ill. tiltja (false). Több elem kiválasztásánál azt, hogy melyik elemek vannak kiválasztva a ListBox **Selected** tulajdonságával vizsgálhatjuk, amely egy 0 indextől kezdődő tömb (pl. a Selected[0] igaz, ha az első elem van kiválasztva, a Selected[1] igaz, ha a második elem van kiválasztva, stb.).
- **SelCount** – kiválasztott elemek darabszámát tartalmazza (ha a MultiSelect értéke igaz).

- **Sorted** – megadja, hogy a lista elemei legyenek-e rendezve ábécé sorrendben. Ha értéke igaz (true), új elem hozzáadásánál a listához automatikusan rendezve kerül a listadobozba.

Nézzük meg egy kicsit részletesebben a ListBox legfontosabb tulajdonságát, az **Items** tulajdonságot. Ez egy TString típusú tulajdonság, melynek sok hasznos metódusa van. Ezek közül a leggyakrabban használt metódusok:

- **Add** – a lista végére új elemet rak be.
- **Clear** – a ListBox összes elemét törli.
- **Delete** – kitöröl egy kiválasztott elemet a listában.
- **Equals** – teszteli, hogy két lista tartalma egyenlő-e. False értéket ad vissza, ha a két lista különbözik a hosszában (elemek számában), más elemeket tartalmaznak, vagy ha más sorrendben tartalmazzák az elemeket.
- **Insert** – új elemet szúr be a listába a megadott helyre.
- **LoadFromFile** – beolvassa a lista elemeit egy szöveges állományból. A sikertelen beolvasást a kivételek segítségével kezelhetjük, melyekről később lesz szó.
- **Move** – egy helyével megadott elemet a listába egy másik (új) helyre helyez át.
- **SaveToFile** – elmenti a lista elemeit egy szöveges állományba. A lista minden eleme egy új sorban lesz a fájlban. A sikertelen mentést a kivételek segítségével kezelhetjük, melyről bővebben későbbi fejezetekben lesz szó.

Nézzünk meg egy példát ezeknek a metódusoknak a használatára, hogy jobban megérthessük őket: Pelda03

A form-ra helyezzünk el egy ListBox-ot, melynek Items tulajdonságába adjunk meg néhány nevet. Rakjunk a form-ra még pár nyomógombot is (Rendezd, Adj hozzá, Töröld, Töröd mind, Olvasd be, Mentsd el, Kilépés).

Most az egyes nyomógombok OnClick eseményeit fogjuk kezelni:

Adj hozzá nyomógomb – egy InputBox segítségével beolvasunk egy nevet, melyet a lista végéhez adunk:

```
procedure TForm1.Button2Click(Sender: TObject);
var s:string;
```



```
begin
  s := InputBox('Adj hozzá',
                'Kérlek add meg a nevet:', '');
  if s<>'' then
    ListBox1.Items.Add(s);
end;
```

Rendezd nyomógomb:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Sorted := true;
end;
```

Töröld nyomógomb – törli a lista kijelölt elemét:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  ListBox1.Items.Delete(ListBox1.ItemIndex);
end;
```

Töröld mind nyomógomb – törli a lista összes elemét:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  ListBox1.Clear;
end;
```

Megjegyzés: A lista törlését itt a ListBox1.Clear metódussal végeztük el. Ez ugyanúgy kitörli a lista elemét, mint a ListBox1.Items.Clear metódus, de az elemek törlésen kívül további „tisztító” műveleteket is elvégez. Gyakorlatilag a két metódus közti különbség a ComboBox komponensnél látható: a ComboBox.Clear

kitörli a teljes listát, a ComboBox.Items.Clear kitörli szintén a listát, de az utolsó kiválasztott érték a beviteli mezőben marad!

Mentsd el nyomógomb:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
  ListBox1.Items.SaveToFile('nevsor.txt');
end;
```

Olvasd be nyomógomb:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  ListBox1.Items.LoadFromFile('nevsor.txt');
end;
```

Láthatjuk, hogy az utóbbi két metódus saját maga megnyitja a fájlt, beolvassa / menti az adatokat, majd bezárja a fájlt.

Kilépés nyomógomb:

```
procedure TForm1.Button7Click(Sender: TObject);
begin
  Form1.Close;
end;
```

Végül még megemlítünk néhány példát a többi metódus használatára is:

Medve Elemér beszúrása a 3. helyre a listában:

```
ListBox1.Items.Insert(2, 'Medve Elemér');
```

A lista első elemének áthelyezése a 3. helyre:

```
ListBox1.Items.Move(0, 2);
```


Ezekből a példákból is jól látható, hogy a ListBox komponens felhasználására nagyon sok lehetőség van. Azonban a ListBox komponens használatának is lehet hátránya: az egyik hátránya lehet, hogy az alkalmazás ablakán állandóan ott van és sok helyet foglal el. Másik hátránya: ha a ListBox-ot bemeneti komponensként használjuk, a felhasználó csak a listában szereplő értékek közül választhat. Természetesen van amikor ez nekünk így jó, de előfordulhat, hogy a felhasználónak több szabadságot szeretnénk adni a választásnál (például saját érték beírását). Ezekre adhat megoldást a ComboBox komponens.

12.10 Kombinált lista – ComboBox

Ennek a komponensnek a formája a képernyőn nagyon hasonlít az Edit komponenséhez, ugyanis a felhasználó sok esetben írhat bele saját szöveget is. Hasonlít azonban a ListBox komponenshez is, mivel a jobb szélén levő nyílra kattintva (vagy Alt + lefelé nyíl, vagy Alt + felfelé nyíl) megjelenik (legördül) egy lista, amelyből a felhasználó választhat.

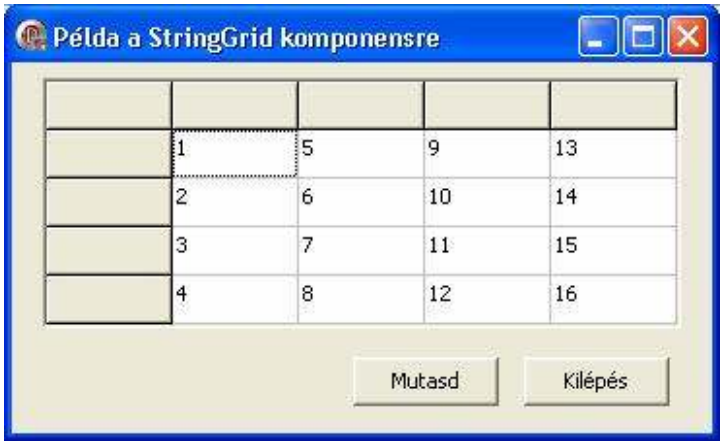
Mivel a ComboBox tulajdonságai, metódusai és használata sok mindenben megegyezik (vagy nagyon hasonlít) a ListBox-al, ezért nem vesszük át mindet még egyszer, helyette inkább kiegészítjük őket továbbiakkal:

- **Style** – ez a tulajdonság nem csak a ComboBox külalakját adja meg, de komponens viselkedését és a felhasználói bemenetek lehetőségét is. Értéke lehet:
  - **csDropDown**: tipikus ComboBox, amely megjeleníti a listát, de közvetlen szöveg bevitelt is lehetővé tesz.

- **csDropDownList**: szövegbevitelt nem tesz lehetővé. Valamelyik betű (billentyű) megnyomásakor az első olyan elemre ugrik a listában, amely ezzel a betűvel kezdődik.
- **csSimple**: a közvetlen szövegbevitelt is lehetővé teszi, a lista közvetlenül a beviteli mező alatt van megjelenítve (állandóan). A megjelenített lista méretét a komponens Height tulajdonsága határozza meg.
- **csOwnerDrawFixed**: kép megjelenítését teszi lehetővé a listában. A lista összes elemének a magassága azonos, melyet az ItemHeight tulajdonság határoz meg.
- **csOwnerDrawVariable**: hasonló az előzőhöz, de az egyes elemeknek a listában különböző magasságuk (méretük) lehet.

12.11 StringGrid komponens

Ez nem olyan gyakran használt komponens, mint a lista. Segítségével szöveges adatokat jeleníthetünk meg táblázatban.



Helyezzünk el a form-on egy StringGrid komponenst és két nyomógombot. Az első nyomógomb OnClick eseményébe írjuk be az alábbi programrészt:

```
procedure TForm1.Button1Click(Sender: TObject);
var i,j,k:integer;
begin
  k := 0;
  with StringGrid1 do
    for i:=1 to ColCount-1 do
      for j:=1 to RowCount-1 do
        begin
```

```
            k := k + 1;
            Cells[i,j] := IntToStr(k);
            end;
    end;
```

A forráskódban StringGrid komponens több tulajdonságával is megismerkedhettünk:

- **ColCount** – oszlopok számát határozza meg (fix oszlopokkal együtt).
- **RowCount** – hasonlóan az előzőhöz, csak ez a sorok számát határozza meg.
- **Cells** – az egész táblázat szövegeinek mátrixa.

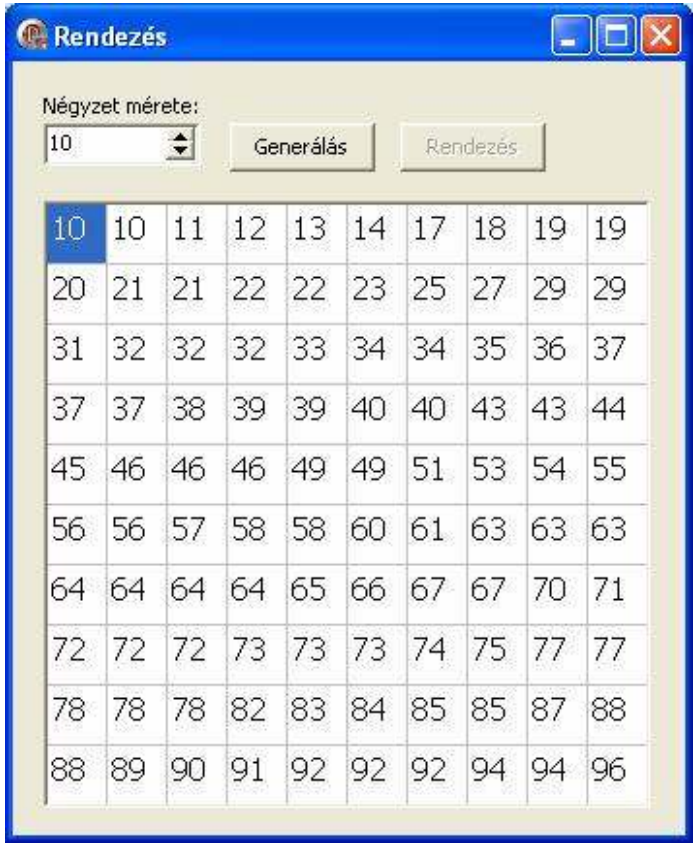
A StringGrid komponens további tulajdonságai, melyek a példában nem szerepelnek:

- **FixedCols** – a rögzített (fix) oszlopok száma.
- **FixedRows** – a rögzített (fix) sorok száma.
- **FixedColor** – a rögzített oszlopok és sorok háttérszíne.
- **GridLineWidth** – az egyes cellák közti vonal vastagsága.

Végül még néhány érdekes metódusa a StringGrid komponensnek:

- **MouseToCell**: az X, Y koordinátákhoz meghatározza a táblázat sorát és oszlopát.
- **CellRect**: a megadott cella képernyő-koordinátáit adja meg pixelekben.

A következő feladat szemlélteti a StringGrid komponens használatát: Készítsünk egy alkalmazást, melyben egy SpinEdit komponens segítségével beállíthatjuk a StringGrid komponens méretét 3x3-tól 10x10-ig. A programunk továbbá tartalmazzon két nyomógombot. Az első nyomógomb generáljon véletlenszerű számokat a StringGrid-be, a második nyomógomb pedig rendezze ezeket a számokat növekvő sorrendbe. Pelda04



Miután a szükséges komponenseket elhelyeztük a form-on, állítsuk be a komponensek alábbi tulajdonságait az Objektum Inspector-ban (vagy írjuk be a a Form OnCreate eseményébe):

```
Label1.Caption := 'A négyzet mérete';
StringGrid1.DefaultColWidth := 30;
StringGrid1.DefaultRowHeight := 30;
StringGrid1.FixedCols := 0;
StringGrid1.FixedRows := 0;
StringGrid1.ScrollBars := ssNone;
SpinEdit1.EditorEnabled := False;
SpinEdit1.MaxValue := 10;
SpinEdit1.MinValue := 3;
SpinEdit1.Value := 5;
Button1.Caption := 'Generálás';
Button2.Caption := 'Rendezés';
Button2.Enabled := False;
```

Majd írjuk meg az egyes komponensek eseményeihez tartozó programrészeket:

```
procedure TForm1.SpinEdit1Change(Sender: TObject);
var i,j:integer;
begin
 // toroljuk a cellak tartalmat
 for i:=0 to 9 do
   for j:=0 to 9 do StringGrid1.Cells[i,j]:='';
 // a rendezes gombot nem elerhetove tesszuk
 Button2.Enabled := false;
 // beallitjuk a sorok es oszlopok szamat
 StringGrid1.ColCount := SpinEdit1.Value;
 StringGrid1.RowCount := SpinEdit1.Value;
 // beallitjuk a StringGrid szelesseget es magassagat
 // minden cella 31 szeles(magas) a vonalal egyutt
 // az egyik szelen + 3 a StringGrig keretenek
 // szelessege(magassaga)
 StringGrid1.Width := 31 * SpinEdit1.Value + 3;
 StringGrid1.Height := 31 * SpinEdit1.Value + 3;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i,j:integer;
begin
 // a cellakba 10-99 kozotti veletlen szamokat
 // generalunk az oszlopok(sorok) 0-tol
 // vannak szamozva !!
 for i:=0 to StringGrid1.ColCount-1 do
  for j:=0 to StringGrid1.RowCount-1 do
    StringGrid1.Cells[i,j] :=
                        IntToStr(random(90)+10);
 // a rendezes gombot elerhetove tesszuk
 Button2.Enabled := true;
end;

procedure TForm1.Button2Click(Sender: TObject);
var i,j,ei,ej:integer;
    s:string;
    csere:boolean;
begin
 // a StringGrid1-el dolgozunk. Az alabbi sor
 // kiadasaval nem kell mindig megadnunk hogy
 // pl. StringGrid1.Cells[i,j], helyette eleg
 // a Cells[i,j] a with parancson belul.
 with StringGrid1 do
  repeat
  ei := 0; // elozo cella sorindexe
  ej := 0; // elozo cella oszlopindexe
  csere := false; // azt jelzi, volt-e csere
                  // (false=nem volt)
  for j:=0 to RowCount-1 do
    for i:=0 to ColCount-1 do
      begin
      // osszehasonlitjuk az aktualis cellat
      // az elozovel
```

```
      if StrToInt(Cells[i,j])<StrToInt(Cells[ei,ej])
      then
        begin
        s := Cells[i,j];
        Cells[i,j] := Cells[ei,ej];
        Cells[ei,ej] := s;
        csere := true;
        end;
      // beallitjuk az elozo cellat az
      // aktualis cellara
      ei := i;
      ej := j;
      end;
  until not csere; // addig megyunk vegig az egesz
                   // StringGrid-en, amig igaz nem
                   // lesz, hogy csere=false;
 // a rendezes gombot nem elerhetove tesszuk,
 // mivel mar rendezve vannak a szamok
 Button2.Enabled := false;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 // a program inditasakor beallitjuk a
 // veletlenszam generatort
 randomize;
end;
```

A StringGrid komponens nem csak adatok megjelenítésére, de adatok bevitelére is használható. Ahhoz, hogy a program futása közben ebbe a komponensbe a felhasználó tudjon beírni közvetlenül is adatokat, át kell állítanunk a **StringGrid.Options** tulajdonságának **goEditing** altulajdonságát true-ra.

12.12 Időzítő – Timer

Gyakran szükségünk lehet bizonyos időnként (intervallumonként) megszakítani a program normális futását, elvégezni valamilyen rövid műveletet, majd visszatérni a program normális futásához – ezt a Timer komponens segítségével tehetjük meg. Időzítővel tudjuk megoldani például mozgó szöveg (folyamatosan körbe futó szöveg) kiírását is.

A Timer komponens nem sok tulajdonsággal rendelkezik, pontosabban egy specifikus tulajdonsága van:

- **Interval** – meghatározza azt az időintervallumot (milliszekundumokban), ami eltelte után újra és újra bekövetkezik a OnTimer eseménye.

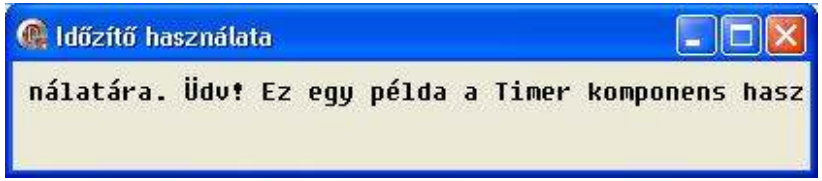
Az **OnTimer** eseménybe írhatjuk azt a kódot, amelyet periodikusan végre akarunk hajtani. Például a már említett körbe futó szöveg így oldható meg a segítségével: Pelda05

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := RightStr(Label1.Caption,
                    Length(Label1.Caption)-1) +
                    LeftStr(Label1.Caption, 1);
end;
```

Mivel a programunk használ két függvényt: RightStr és LeftStr, amelyek az StrUtils unitban találhatók, ki kell egészítenünk programunk uses részét ezzel a unittal:

```
uses
      …, StrUtils;
```

A RightStr függvény az első paraméterként megadott szöveg jobb, a LeftStr a szöveg bal részéből ad vissza a második paraméterben megadott mennyiségű karaktert.



Megjegyzés: A Timer a Windows időzítőjét használja, amely intervalluma a Windows 98-ban 55 milliszekundum, a Windows NT-ben 10 milliszekundum. Ebből következik, hogy ennék kisebb intervallumot hiába adunk meg a Timer komponensben, a művelet nem fog ennél rövidebb időközönként végrehajtódni. Továbbá a Windows belső órája nem pontos. Ha például a komponensünk Interval tulajdonságát 1000 ms-ra állítjuk be, az nem jelenti azt, hogy pontosan 1 másodpercenként fog bekövetkezni az OnTimer esemény. A Windows órája és a kerekítés végett ebben az esetben 989 ms-onként következne be az esemény. Továbbá ha az alkalmazásunk hosszabb ideig (példánkban 1 mp-nél tovább) foglalt, akkor sem következik be az esemény, és miután felszabadul, nem fog bekövetkezni több esemény egymás után hirtelen (nem halmozódik fel), hanem csak egy, majd a következő esemény csak a megadott intervallum eltelte után lesz.

12.13 Gauge, ProgressBar komponensek

Egy hosszabb folyamat állapotát jelezhetjük ezeknek a komponenseknek (és a Timer komponens) segítségével.

Nézzük először a **Gauge** komponens fontos tulajdonságait:

- **MinValue** – minimális értéke a sávnak (default: 0).
- **MaxValue** – maximális értéke a sávnak (default: 100).
- **Progress** – aktuális értéke a sávnak.

Például: ha a MinValue = 0 és MaxValue = 200, akkor a Progress = 20 érték 10%-nak felel meg, amely a komponensen is megjelenik.

További tulajdonságai a Gauge komponensnek:

- **ForeColor** – a kitöltés színe.
- **Kind** – a komponens külalakját határozza meg. Lehetséges értékek: gkHorizontalBar (vízszintes sáv), gkVerticalBar (függőleges sáv), gkNeedle („analóg sebességmérő óra”), gpPie („kalács” – kör formájú alak, melyben a szelet

nagyobbodik), gkText (csak a százalékot jeleníti meg, nem szemlélteti semmilyen grafikus elemmel).

Most nézzünk egy példát, amely segítségével a Gauge használatát szemléltetjük. A példánkban 10 másodperc alatt ér a folyamat a végére. Pelda06

A form-ra helyezzünk el egy Gauge komponenst és egy Timer komponens. A Timer komponens Interval tulajdonságát állítsuk be 100 milliszekundumra (tehát másodpercenként 10-szer fog bekövetkezni az OnTimer eseménye). Az OnTimer eseménybe a következő programrész szerepel:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Gauge1.Progress := Gauge1.Progress + 1;
end;
```

A **ProgressBar** komponens hasonló a Gauge-hoz, csak más a külalakja és mások a tulajdonságainak a nevei: a MinValue, MaxValue és Progress helyett **Min**, **Max** és **Position** tulajdonságai vannak.

13 További komponensek

Ebben a fejezetben főleg olyan további komponenseket sorolunk fel, melyek grafikailag szebbé, érdekesebbé tehetik alkalmazásunkat.

13.1 Kép használata – Image

Kép megjelenítését teszi lehetővé. Ezen kívül jól használható például rajzprogram készítésére is, mivel tartalmaz egy vászont (**Canvas** objektum), melyre bármit kirajzolhatunk. Az image komponens leggyakrabban használt tulajdonságai:

- **Picture** – megjelenítendő kép.
- **Stretch** – ha értéke igaz (true), akkor az egész képet megjeleníti a komponensben. Tehát ha nagyobb a kép, akkor lekicsinyíti a komponens méretére, ha kisebb, akkor felnagyítja.
- **Proportional** – ha értéke igaz (true), akkor betartja a szélesség és magasság arányát, tehát nem torzul a kép.
- **Transparent** – bitmap (BMP) kép esetében, ha a transparent tulajdonság értéke igaz (true), akkor a háttérszínt átlátszóvá teszi (csak akkor működik, ha a stretch tulajdonság hamis - false). Háttérszínnek a *Delphi* a bitmap bal alsó sarkában levő pont színét veszi.

Próbáljuk ki az Image és a Timer komponensek használatát a gyakorlatban is. Készítsünk egy egyszerű képernyővédőt, melyben egy

léggömb fog úszni balról jobbra. Ha kimegy a képernyő jobb szélén, akkor véletlenszerű magasságban beúszik a képernyő bal széléről. Ez mindaddig menjen, amíg nem nyomunk le egy billentyűt vagy nem mozgatjuk meg az egeret. Pelda07

Mielőtt belekezdenénk a programunk készítésébe, rajzoljuk meg Paint-ban (vagy más rajzoló programban) a léggömböt, melynek mérete legyen 200 x 200 pixel. Ezt a rajzot BMP fájlformátumban mentsük el.

Ha kész a rajzunk, nekiláthatunk az alkalmazásunk elkészítésének. Az alkalmazásunk elkészítése, mint azt már eddig is megfigyelhettük három fő lépésből fog állni:

1. komponensek kiválasztása és elhelyezése a form-on,
2. komponensek tulajdonságainak (Properties) beállítása az Objektum felügyelőben,
3. az eseményekhez (Events) tartozó eljárások programkódjának megírása.

Kezdjük tehát az elsőnél, a komponensek kiválasztásánál. A **Form**-unkra tegyünk egy képet (**Image**) és egy időzítőt (**Timer**). A kép lesz maga a léggömb, az időzítő pedig ezt a képet fogja mozgatni (minden 10. milliszekundumban egy képponttal jobbra teszi).

Állítsuk be a komponensek tulajdonságait az Objektum felügyelőben:

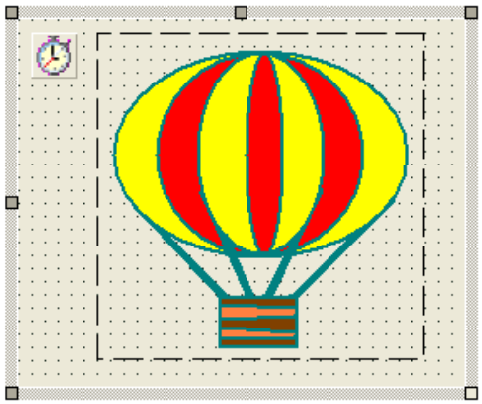
- **Form1.WindowState** := **wsMaximized**;
  Ezzel az ablakunk kezdetben maximalizált állapotban lesz.
- **Form1.BorderStyle** := **bsNone**;
  Az ablakunknak nem lesz látható kerete, tehát az induláskor

az egész képernyőt betakarja majd (mivel maximalizált és nincs keretje).

Megjegyzés: ugyanezzel a tulajdonsággal tudjuk beállítani azt is, hogy az ablakunk ne legyen futási időben átméretezhető (bsSingle). Továbbá egy hasonló tulajdonsággal, a BorderIcons-al megadhatjuk, hogy az ablakunk keretén melyik gombok legyenek elérhetők (minimalizálás, maximalizálás, bezárás). Igaz, ebben a programban ezekre most nincs szükségünk, de a jövőben még jól jöhet ezek ismerete más programok készítésénél.

- **Image1.Picture**
  Ennél a tulajdonságnál adjuk meg az elmentett képünket (BMP), amely a léggömböt ábrázolja.
- **Image1.Width** := **200**;
  Image1.Height := **200**;
  Meghatározzuk a képünk méretét.
- **Image1.Transparent** := **true**;
  Megadjuk, hogy képünk háttere átlátszó legyen.
- **Timer1.Interval** := **10**;
  Megadjuk, hogy az időzítőnél 10 milliszekundumonként következzen be az OnTimer esemény.

Ezzel megadtuk a fontosabb tulajdonságokat. Alkalmazásunk tervezési fázisban most valahogy így néz ki:



Ezek után nekiláthatunk az események kezelésének, tehát a programkód megírásának.

A **Form1 – OnCreate** eseményében beállítjuk véletlenszerűen a léggömb helyzetét, az ablakunk hátterének színét és az egér kurzorát (ez utóbbit nincs-re állítjuk):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  randomize;
  Image1.Top := Random(Screen.Height-200);
  Image1.Left := Random(Screen.Width-200);
  Form1.Color := clBlack;
  Form1.Cursor := crNone;
end;
```

Itt azért a **Screen** objektumot használtuk és nem a Form1-et, mert ennek az eljárásnak a meghívásakor az alkalmazásunk még nincs maximalizálva, így nem kapnánk meg az egész képernyő méretét. A Screen (képernyő) objektum segítségével meghatározható a képernyő felbontása úgy, ahogy azt a fenti programrészben is tettük.

Most beállítjuk, hogy a léggömb mozogjon, tehát növeljük a **Timer1 – OnTimer** eseményében a kép Left tulajdonságát 1-gyel. Ha a kép kiment a képernyő jobb oldalán, véletlenszerű magasságban átrakjuk a képernyő bal oldalára negatív pozícióba (-Image1.Width), ahonnan be fog jönni a képernyőre.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Image1.Left := Image1.Left + 1;
  if Image1.Left > Form1.Width then
    begin
    Image1.Left := -Image1.Width;
    Image1.Top := Random(Form1.Height-200);
    end;
end;
```

Most beállítjuk, hogy bármelyik billentyű megnyomására a program befejeződjön. Ezt a **Form1 – OnKeyDown** eseményében tehetjük meg:

```
procedure TForm1.FormKeyDown(Sender: TObject;
          var Key: Word; Shift: TShiftState);
begin
  Form1.Close;
end;
```

Próbáljuk meg lefuttatni az alkalmazásunkat. Kilépni egyelőre csak valamelyik gomb megnyomásával tudunk, egérmozgatással nem. Továbbá láthatjuk, hogy a képernyőn a léggömb mozgatása villogással jár. Ezt kiküszöbölhetjük, ha a **Form1 – OnCreate** eseményének kezelésében beállítjuk a Form **DoubleBuffered** tulajdonságát igazra (true). Ennek a tulajdonságnak igazra való állítása azt eredményezi, hogy a form-unk nem közvetlenül a képernyőn lesz átrajzolva, hanem egy bitmap segítségével a memóriában. Igaz, ezzel a memóriából több helyet lefoglal a programunk, viszont a villogás így megszűnik. Egészítsük ki tehát a TForm1.FormCreate eljárását a következő sorral:

```
Form1.DoubleBuffered := true;
```

Most már csak az hiányzik, hogy a programunkból ki lehessen lépni az egér megmozdításával is.

Ahhoz, hogy a képernyővédőnk ne lépjen ki az egér kisebb mozgatására (változására) szükségünk lesz két változóra (**egerpoz1**, **egerpoz2** neveket adunk nekik), melyek segítségével ki fogjuk számolni, hogy mennyi képpontot mozdult el az egér a program inditása óta. Amint ez több lesz, mint 10 pixel, leállítjuk a programot. Az egér koordinátáinak tárolására használt mindkét változónk (egerpoz1, egerpoz2) típusa TPoint (pont) lesz, amely valójában egy record típus, amely megtalálható (definiálva van) a Delphi-ben a következő képpen:

```
type TPoint = record
              X,Y:integer;
              end;
```

Programunkat tehát egészítsük ki ennek a két globális változónak a deklarálásával:

```
var egerpoz1, egerpoz2: TPoint;
```

Az **egerpoz1** változó értékét a program futásának kezdetén beállítjuk az egér kezdeti pozíciójára. Ezt a **GetCursorPos** függvény segítségével kérhetjük le. Egészítsük tehát ki a **Form1 – OnCreate** eseményét a következő sorral:

```
GetCursorPos(egerpoz1);
```

Ezek után az időzítő (Timer1) eljárásának minden egyes lefutásakor lekérjük az egér pozícióját az **egerpoz2** változóba, majd Pithagorasz tételének segítségével kiszámítjuk az egér kezdeti (egerpoz1) és jelenlegi (egerpoz2) pozíciója közti elmozdulás nagyságát. Amennyiben ez több mint 10 képpont, bezárjuk a formot

(leállítjuk a képernyővédőt). Tehát a **TForm1.Timer1Timer** eljárást egészítsük ki az **elmozdulas** változó deklarálásával és az elmozdulás kiszámítására (ill. szükség esetén a program leállítására) szolgáló programrésszel:

```
…
var elmozdulas:integer;
…
GetCursorPos(egerpoz2);
elmozdulas :=
       round(sqrt(sqr(egerpoz1.x-egerpoz2.x)
                  +sqr(egerpoz1.y-egerpoz2.y)));
if elmozdulas>10 then Form1.Close;
```

Futtassuk le az alkalmazásunkat. Próbáljunk meg kilépni az egér mozgatásával. Működik? Képernyővédőnk megpróbálhatjuk magunk továbbfejleszteni például úgy, hogy két léggömbünk legyen, a léggömb mérete változzon, esetleg ne csak vízszintesen haladjon a léggömb, hanem közben emelkedjen, vagy süllyedjen is.

13.2 Választóvonal – Bevel

Ez az egyik legegyszerűbb, mégis elegáns komponens az alkalmazásunk szebbé tételére. Segítségével a komponenseket vizuálisan szétoszthatjuk csoportokra, elválaszthatjuk őket egymástól. A Bevel komponenssel tehetünk az alkalmazásunkba egyszerű vonalat vagy keretet (amely lehet benyomódva vagy kiemelve is). Fontos megemlíteni, hogy a Bevel komponens nem lesz a ráhelyezett komponensek tulajdonosa (mint pl. a Panel komponensnél), csak

vizuálisan jelenik meg a programban (tehát nem csoportosíthatunk vele pl. RadioButton-okat sem logikai csoportokba)!

A komponensnek két fontos tulajdonsága van:

- **Shape** – meghatározza, hogyan nézzen ki a Bevel komponens. Lehetséges értékei:
  - **bsBottomLine** – a komponens alján egy vízszintes vonal jelenik meg.
  - **bsBox** – doboz (keret), amely belseje attól függően, hogy a komponens Style tulajdonsága mire van állítva vagy beljebb lesz vagy kijjebb lesz az ablakon (form-on).
  - **bsFrame** – keret körbe. A keret belseje egy szintben van az ablakkal (form-mal).
  - **bsLeftLine** – függőleges vonal a komponens bal oldalán.
  - **bsRightLine** – függőleges vonal a komponens jobb oldalán.
  - **bsSpacer** – üres (nem lesz látható semmi).
  - **bsTopLine** – vízszintes vonal a komponens tetején.
- **Style** – meghatározza, hogy az alakzat beljebb lesz (bsLower) vagy kijjebb lesz (bsRaised) az ablakon. Természetesen ez csak azoknál az alakzatoknál (Shape tulajdonság) használható, melyeknél van értelme.

Ennek a komponensnek nincs egyetlen eseménye sem.

13.3 Alakzat – Shape

Egyszerű geometriai alakzatok, melyekkel a programunkat szépíthetjük. Hasonlóan az előző komponenshez, nem lesz a ráhelyezett komponens tulajdonosa, csak vizuális célt szolgál. Fontosabb tulajdonságai:

- **Shape** – az alakzat formája (stCircle, stEllipse, stRectangle, stRoundRect, stRoundSquare, stSquare).
- **Brush** – kitöltés színe és stílusa.
- **Pen** – körvonal színe, vastagsága és stílusa.

13.4 Grafikus nyomógomb – BitBtn

Ha a hagyományos nyomógombtól (Button) egy kicsit eltérőt szeretnénk kialakítani, használhatjuk a BitBtn komponenst. Ez nagyon hasonló a Button komponenshez, de BitBtn nyomógombon képet is el lehet helyezni, így bármilyen egyedi külalakú gombot létrehozhatunk. Itt most csak a Button-nál nem található, további fontos tulajdonságait fogjuk felsorolni a BitBtn komponensnek:

- **Glyph** – a gombon megjelenítendő kép (bitmap). Ha azt szeretnénk, hogy a saját képünk hasonló legyen az előre definiáltakhoz, akkor 18 x 18 képpontnyi méretű képet használjunk.
- **Kind** – néhány előre definiált kép közül választhatunk, lehetséges értékei: bkCustom, bkYes, bkNo, bkOK, bkCancel, bkAbort, bkRetry, bkIgnore, bkAll, bkClose, bkHelp. Ha a bkCustom-ot választjuk, saját képet adhatunk meg (a Glyph tulajdonságban).

- **Layout** – meghatározza, hogy a kép hol jelenjen meg a nyomógombon (bal oldalon, jobb oldalon, fent, lent).
- **Margin** – meghatározza a kép és a gomb bal széle közti távolságot pixelekben. Ha értéke -1, akkor a képet a szöveggel együtt középre igazítja.
- **Spacing** – meghatározza a kép és a felirat közti távolságot pixelekben.
- **NumGlyphs** – Megadja, hogy a bitmap hány képet tartalmaz. A bitmap-nek teljesítenie kell két feltételt: minden képnek ugyanolyan méretűnek kell lennie és a képeknek sorban, egymás után kell elhelyezkedniük. A BitBtn komponens ezek után egy ikont jelenít meg ezek közül a képek közül a BitBtn állapotától függően:
  - Up – a nem lenyomott gombnál jelenik meg, és a többi állapotnál akkor, ha nincs más kép,
  - Disabled – akkor jelenik meg, ha a gombot nem lehet kiválasztani,
  - Clicked – ha a gomb éppen meg lett nyomva (rá lett klikkelve egérrel),
  - Down – ha a gomb tartósan lenyomott állapotban van (ez az állapot a BitBtn komponensnél nem következik be, ennek az állapotnak csak a SpeedButton gombnál van jelentősége, melyet a következő részben írunk le).

Azzal, hogy hogyan hozhatunk létre bitmap-ot több képből, megismerkedünk az ImageList komponenst tárgyaló fejezetben.

13.5 Eszköztár gomb – SpeedButton

Az eddig tárgyalt gombok, a sima Button, a BitBnt gombok nem használhatóak eszköztár gombnak. Gondoljuk végig például Word-ben a szöveg igazítását:

A szöveg igazításánál a gombok közül egy állandóan lenyomott állapotban van. Ezt a többi gomb nem tudja megvalósítani. A kölcsönös kizárás lényege ilyen esetben, hogy az egy csoportba tartozó gombok közül, mindig csak egy lehet lenyomva, s a másik lenyomásakor az előző felenged. Külön megadható az is, hogy legalább egynek benyomva kell-e lennie, vagy egy csoportban mindegyik lehet-e felengedett állapotban.

A SpeedButton komponens nagyon hasonló a BitBtn komponenshez, melyet az előző fejezetben tárgyaltunk. Hasonlóan a BitBtn komponenshez ez is tartalmazhat több képet, mindegyik állapotához egy-egy képet. Itt jelentősége lesz a negyedik – down (tartósan lenyomva) állapotnak is.

Legfontosabb tulajdonságai:

- **Glyph** – a gombon elhelyezkedő kép. Négy különböző képet helyezhetünk el a gomb állapotának megfelelően.

- **GroupIndex** – ennek a tulajdonságnak a segítségével csoportosíthatóak a gombok. Az egy csoportba tartozó gomboknak azonos GroupIndex-el kell rendelkezniük. Amennyiben a GroupIndex = 0 akkor a gomb BitBtn gombként fog viselkedni, tehát nem fog „benyomva" maradni. Ha GroupIndex-nek 0-nál nagyobb számot adunk, akkor létrejön a csoportosítás.
- **Down** – ez a tulajdonság adja meg, hogy a gomb lenyomott állapotban van-e (true esetén lenyomott állapotban van). Csak 0-tól különböző GroupIndex esetén használható.
- **AllowAllUp** – ez a tulajdonság határozza meg, hogy az egy csoportba tartozó gombok közül lehet-e mind egyszerre felengedve (true), vagy az egyik gombnak mindenképen muszáj benyomva maradnia (false).

A SpeedButton-okat szinte mindig egy Panel komponensen helyezzük el, eszköztárat alakítva ki belőlük.

13.6 Kép lista – ImageList

Ahhoz, hogy megismerkedjünk az eszköztár egy másik módszerének kialakításával (ToolBar komponens segítségével), előbb szükséges néhány szót szólnunk az ImageList komponensről. Ennek a komponensnek a célja, több ugyanolyan méretű kép tárolása. Ezek a tárolt képek (bitmapek, ikonok…) indexek segítségével érhetők el. A komponenst tervezési fázisban egy kis négyzet jelképezi, amely az alkalmazás futása alatt nem látható (hasonlóan pl. a Timer komponenshez).

Az ImageList komponenst tehát hatékonyan ki tudjuk használni több kép vagy ikon tárolására. Az összes kép az ImageList-ben úgy van reprezentálva, mint egy darab széles bitmap. Ezt felhasználhatjuk a BitBtn, SpeedButton komponenseknél, de hasonlóan felhasználható további komponenseknél is (pl. ToolBar).

Mindegyik ListBox-ban tárolt kép index segítségével érhető el, melynek értéke 0-tól N-1-ig lehet (N darab kép esetében). Például: az ImageList1.GetBitmap(0, Image1.Picture.Bitmap); metódus az ImageList1-ben tárolt 0 indexű képet állítja be az Image1 Bitmap-jának.

Az alkalmazás tervezési fázisában a képek megadásához az Image List Editor eszközt használhatjuk, melyet a komponensre kattintva jobb egérgombbal (vagy dupla kattintással) hívhatunk elő.

Az ImageList fontosabb tulajdonságai:

- **Width** – a komponensben tárolandó képek szélessége.
- **Height** – a komponensben tárolandó képek magassága.

13.7 Eszköztár – ToolBar

A ToolBar komponens segítségével szintén előállíthatunk eszköztárt. Az előbbi fejezetekben leírt eszköztár kialakításával (panel és SpeedButton komponensekből) szemben a ToolBar komponens valamivel több lehetőséggel rendelkezik, továbbá más a tervezési fázisa is. Szorosan együttműködik az ImageList komponenssel, melyet az előző részben ismerhettünk meg. Az eszköztás kialakítását ToolBar komponens segítségével egy példán ismertetjük:

1. A form-on elhelyezünk egy ToolBar és egy ImageList komponenst.

2. Jobb egérgombbal rákattintunk az ImageList-re és kiválasztjuk az Image List Editor-t.
3. Az Editor-ban az Add gomb megnyomásával hozzáadjuk az ImageList-hez a szükséges képeket (bitmapot – BMP és ikont – ICO választhatunk). A képek hozzáadása után az OK gomb megnyomásával bezárjuk az Editor-t.
4. Az egér bal gombjával rákattintunk a ToolBar komponensre, majd az Object Inspector-ban beállítjuk az **Images** tulajdonságot az ImageList komponensünkre.
5. Jobb egérgombbal rákattintunk a ToolBar komponensre, majd kiválasztjuk a „New Button” menüpontot. Ezt a lépést megismételjük annyiszor, ahány gombot akarunk elhelyezni. Ne felejtsünk el néha berakni „New Separator”-t is, hogy a gombok áttekinthetően legyenek elrendezve.
6. Az előző lépés ismétlésénél a gombokhoz automatikusan hozzá lettek rendelve az ImageList-bel levő ikonok. Ha ez a kezdeti beállítás nekünk nem felel meg, nem probléma megváltoztatni őket a ToolBar nyomógombjainak (tehát a **ToolButton** objektumoknak) az **ImageIndex** tulajdonságainak átállításával.

Láthatjuk, hogy ennek a komponensnek a segítségével nagyon egyszerűen és rugalmasan kialakíthatunk eszköztárakat. Megemlítjük még a ToolBar komponens fontosabb tulajdonságait:

- **DisabledImages** – segítségével meghatározhatjuk, hogy a nyomógombokon milyen képek jelenjenek meg, ha a nyomógomb nem elérhető.

- **HotImages** – segítségével meghatározhatjuk, hogy a nyomógombokon milyen képek jelenjenek meg, ha a nyomógomb aktív (ki van választva).

A ToolBars-on elhelyezkedő nyomógombok, választóvonalak (ToolButton objektumok) fontosabb tulajdonságai:

- **Style** – meghatározza a nyomógombok stílusát és viselkedését is. Lehetséges értékek:
  - **tbsButton**: nyomógomb,
  - **tbsDivider**: látható függőleges választóvonal,
  - **tbsDropDown**: legördülő választék - menü (pl. *Word* betűszínének kiválasztása az eszköztárban),
  - **tbsCheck**: nyomógomb két lehetséges (lenyomott ill. felengedett) állapottal,
  - **tbsSeparator**: üres hely (választósáv).
- **Grouped** – meghatározza, hogy a gombok olyan logikai csoportokba vannak-e osztva, melyekben mindig csak egy gomb lehet lenyomva (pl. *Word* sorigazitásai). A gombok logikai csoportokba való osztását az határozza meg, hogyan vannak elválasztva (tbsDivider, tbsSeparatos stílusú ToolButton-okkal). Ennek a tulajdonságnak csak a tbsCheck stílusú nyomógomboknál (ToolButton-oknál) van értelme.
- **MenuItem** – segítségével a gombokhoz a főmenü egyes menüpontjait lehet hozzárendelni.

13.8 Állapotsáv – StatusBar

Az alkalmazásunk ablakának alján az állapotsáv (StatusBar) segítségével tudunk kiíratni a felhasználónak különféle információkat. Például egy grafikus editorban kiírathatjuk ide az egér koordinátáit, a kijelölt rész koordinátáit és méretét, a vonalvastagságot, az aktuális betűtípust, stb. Ha StatusBar komponenst rakunk az alkalmazásunkba, az automatikusan az ablak aljához „tapad”, mivel az Align tulajdonsága alapértelmezésben erre van állítva. Legfontosabb tulajdonsága:

- **Panels** – tervezési fázisban egy editor segítségével megadhatjuk hány részre legyen szétosztva, pontosabban hány részből álljon az állapotsávunk. Az egyes részeket indexek segítségével érhetjük el. Minden egyes rész egy új, TStatusPanel típusú objektum. A StatusPanel fontosabb tulajdonságai:
  - **Width –** meghatározza a szélességét, azonban az utolsó StatusPanel szélessége mindig az alkalmazásunk ablakának szélességétől függ, mivel az utolsó a maradék részt tölti ki.
  - **Text** – a StatusPanelon megjelenítendő szöveget tartalmazza.
  - **Alignment** – a szöveg igazítását határozza meg a StatusPanelen belül.

Az alkalmazás futási idejében ha meg szeretnénk jelentetni valamit a StatusBar első StatusPanel-ján (ez a 0. indexű), például a betűtípust, azt a következő módon tehetjük meg:

```
StatusBar1.Panels[0].Text := 'Times New Roman';
```


13.9 Könyvjelzők – TabControl, PageControl

A *Delphi*-ben könyvjelzőkkel kétféle képen dolgozhatunk. Vagy TabControl vagy PageControl segítségével. Első látásra nem látunk a két komponens között különbséget, mégis mindkét komponens működése eltérő.

A **TabControl** csak a könyvjelzők definiálására szolgál. A felső részében megjeleníti a könyvjelzőket, de saját maga nem tartalmaz semmilyen lapokat: ha elhelyezünk egy komponenst valamelyik „lapon” (bár fizikailag nincs egyetlen lapja sem), mindegyik „lapon” látható lesz. Ebből adódik, hogy a lapok közötti átváltást nekünk kell beprogramoznunk (pl. átváltáskor nekünk kell a rajta levő komponenseket láthatóvá ill. láthatatlanná tenni).

A **PageControl** az előzővel ellentétben már tartalmaz lapokat is. Mindegyik lapja tartalmazhat saját komponenseket. Ha valamelyik

lapra elhelyezünk egy komponenst, az a többi lapon nem lesz látható, tehát fizikailag is csak az adott lapon lesz rajta.

Ezekből a különbségekből adódik a két komponenssel való eltérő munka és a két komponens eltérő kialakítása is a tervezési fázisban.

TabControl

A TabControl-nál a könyvjelzőket (füleket) a **Tabs** tulajdosnág segítségével adhatjuk meg, amely a már ismert TString típusú. Így a tervezési időben használhatjuk a String List Editor-t. Továbbá kihasználhatjuk az össze metódust, amelyet a TString típusnál megismertünk. A **TabIndex** tulajdonság meghatározza az aktuális könyvjelzőt.

A TabControl egyik fontosabb eseménye az **OnChanging**, amely akkor következik be, ha a felhasználó át szeretne váltani másik fülre (könyvjelzőre). Ebben az eljárásban megakadályozhatjuk az átváltást is, ha például nem megfelelő értékeket adott meg – erre az AllowChange paraméter szolgál.

PageControl

Vegyük észre, hogy a PageControl-nál az egyes lapok fizikailag új komponensek (TabSheet). Igaz, hogy ez egy kicsit bonyolítja a tervezését, viszont itt mindegyik lapra külön-külön komponenseket rakhatunk.

A PageControl-nál nincs editorunk, amelyben be tudnánk állítani az egyes könyvjelzőket (füleket). A tervezési fázisban úgy tehetünk bele

új lapot, hogy a PageControl komponensre jobb egérkattintással rákattintunk és kiválasztjuk a New Page menüpontot. Minden létrehozott lappal úgy tudunk dolgozni, mint egy külön komponensel.

A kiválasztott lapot az **ActivePage** tulajdonság határozza meg. Ez egy TTabSheet típusú tulajdonság, tehát egyenesen az adott lapot használja, nem az indexét vagy más „mutatót”. A következő vagy előző lapra való átmenésre a programban elég meghívni a PageControl **SelectNextPage** metódusát.

13.10 Formázható szövegdoboz – RichEdit

A többi szövegdoboztól a legfőbb eltérés, hogy itt a szöveg formázható. A Memo-hoz hasonló tulajdonságokkal és metódusokkal rendelkezik. További előnye, hogy beolvassa, vagy elmenti RTF állományba a formázott szöveget. Pl. *Wordpad*-del előállíthatunk egy RTF állományt, s azt beolvastathatjuk *Delphi*-ben.

Tulajdonságai hasonlóak a Memo komponens tulajdonságaihoz, ezért itt csak néhány további fontosabb tulajdonságát említjük meg:

- **Lines** – a Memo-hoz hasonlóan épül fel, TStrings a típusa. A TStrings típusnak van olyan metódusa, mely fájlból olvassa be a szöveget, ill. oda ki tudja menteni. Itt a RichEdit esetében van egy automatikus konverzió, hogy ne az RTF fájl sima szöveges változatát lássuk, hanem a megformázott szöveget. Így nekünk itt is csak a beolvasással vagy a fájlba írással kell törődnünk.

- **PlainText** – igaz (true) érték esetén a szöveget sima TXT állományba menti el, hamis (false) értéknél a mentés RFT fájlba történik.

Példa egy RTF állomány beolvasására:

```
RichEdit1.Lines.LoadFromFile('c:\delphi.rtf');
```

13.11 XPManifest komponens

Ha azt szeretnénk, hogy az alkalmazásunknak, melyet létrehozunk, elegáns kinézete legyen, mint más modern Windows XP alatti alkalmazásoknak, használhatjuk az XPManifest komponenst.

Ennek a komponensnek a használata nagyon egyszerű, elég elhelyezni bárhova az alkalmazásunkban (futási időben nem látható). A különbség a program futása alatt mindjárt látható lesz az egyes komponensek külalakján:

Ha el szeretnénk távolítani az XPManifest komponenst az alkalmazásunkból, nem elég kiszedni az alkalmazás ablakából, a teljes eltávolításhoz ki kell törölnünk a programunk uses részéből is az **XPMan** unitot.

14 Menük létrehozása

Az alkalmazásunkban kétfajta menüt hozhatunk létre: főmenüt és lokális (popup) menüt. Nézzük ezeket sorban.

14.1 Főmenü – MainMenu

A főmenü az alkalmazásunk ablakának legtetején helyezkedik el. Mielőtt belekezdenénk a menük létrehozásába a *Delphi*-ben, nézzük meg milyen követelményeknek kell megfelelnie a főmenünek. Természetesen ezek csak javaslatok, nem kötelező őket betartani, de ajánlott.

A főmenü menüpontjai lehetnek:

- **parancsok** – azok a menüpontok, melyek valamilyen parancsot hajtanak végre, cselekményt indítanak el.
- **beállítások** – olyan menüpontok, amelyek segítségével a program valamilyen beállításának ki vagy bekapcsolása lehetséges. Ezeknél a menüpontoknál bekapcsolt állapotban egy „pipa” (vagy pont) van a menüpont bal oldalán.
- **dialógusok** – menüpontok, melyek hatására egy új ablak (dialógusablak) jelenik meg. Az ilyen menüpontoknál a nevük (feliratuk) után három pont van. Ezt a három pontot a név után mi írjuk be. A három pont kirakása a menüpontban nem kötelező, ez nélkül is működik, de ajánlott ezt az elvet betartanunk.

- **almenüt megnyitó menüpontok** – olyan menüpont, mely egy mélyebb szinten levő almenüt nyit meg. Az ilyen menüpont a jobb szélén egy kis háromszöggel van megjelölve.

Menüpontot, mely valamilyen parancsot végrehajt, tehetünk a főmenü sávjába is (amely mindig látható az ablak tetején). Ez azonban nem ajánlatos, mivel a felhasználó általában ezekre klikkelve egy menü megnyílását várja el alatta (melyből aztán választhat), nem azonnal egy parancs lefutását. Így ez nagyon zavaró lehet.

Másik dolog, mely a felhasználót zavarhatja, ha a menüből egy almenü nyílik meg, abból egy újabb almenü, stb. Legjobb, ha a menü legfelső szintjére klikkelve megnyílik egy olyan választék, melyből már nem nyílik meg további, alacsonyabb szintű almenü, csak kivételes esetekben. Almenük helyett inkább használjunk a menüben vízszintes választóvonalakat.

A felhasználó számára másik, nagyon zavaró eset lehet, ha a menüben megváltoznak a menüpontok nevei. Néha ez jól jöhet, pl. ha rákattintunk a „Táblázat megjelenítése” menüpontra, akkor az megváltozhat „Táblázat eltüntetése” menüpontra, de nagyon sok esetben megzavarhatjuk vele a felhasználót (főleg ha a megváltozott név nincs logikai összefüggésben az előzővel, pl. „Táblázat megjelenítése” után ha megjelenne „Lista megjelenítése” ugyanabban a menüpontban).

További zavaró eset lehet, ha a menüben eltűnnek és megjelennek menüpontok. A felhasználók többsége csak a menüpont helyzetét jegyzi meg, nem a pontos nevüket. A menüpontok eltüntetése (visible) helyett használjuk inkább a menüpontok engedélyezésének

tiltását (enabled). Így a menüpont a helyén marad, csak „szürke” lesz, nem lehet rákattintani.

A menüpontokat próbáljuk valamilyen logikailag összefüggő csoportokban elrendezni. Használjunk a csoportok között vízszintes választóvonalakat, de azért vigyázzunk, hogy ezt se vigyük túlzásba. Egy összefüggő csoportban jó, ha nincs több 5-6 menüpontnál.

Fontos, hogy betartsuk a menü standard struktúráját, melyek minden alkalmazásban hasonlóak, és melyekhez a felhasználók már hozzászoktak. A menüsávot Fájl, Szerkesztés, Nézet… menüpontokkal kezdjük, a végén legyenek a Beállítások, Eszközök, Ablak, Súgó menüpontok. Az almenüknél is próbáljuk meg betartani a standard elrendezést, pl. a File menüpont alatt legyen az Új, Megnyitás, Mentés, Mentés másként, …, Nyomtató beállítása, Nyomtatás, Kilépés menüpontok. Hasonlóan a Szerkesztés alatt legyen a Visszavonás, Kivágás, Másolás, stb.

Tartsuk be a megszokott billentyűkombinációkat is, pl. a Ctrl+C a másolás, Ctrl+V a beillesztés legyen, stb. Nem nagyon örülnének a felhasználók, ha pl. a Crtl+C megnyomásakor befejeződne a program. A másik fajta billentyűzetkombinációk, melyekre szintén próbáljunk meg odafigyelni: az Alt+betű típusúak, melyeknél, ha lehet, az adott betűre kezdődő menüpont nyíljon meg.

Ezzel összefügg a menü nyelve is. Ha magyar programot készítünk, használjunk benne a menük neveinek (feliratainak) is magyar szavakat. Ha külföldön is szeretnénk a programunkat terjeszteni, készítsünk külön egy angol változatot. A billentyűkombinációkat azonban nem ajánlatos lefordítani! Pl. a Ctrl+C maradjon Ctrl+C, ne változtassuk meg pl. Crtl+M-re (mint másolás). A megváltoztatásukkal több kárt érnénk el, mint hasznot.

Ez után a rövid bevezető után nézzük, hogyan készíthetünk a *Delphi*-ben menüt. Helyezzünk el az ablakunkon bárhova egy **MainMenu** komponenst. A komponens az alkalmazásunkban egy kis négyzettel lesz jelezve, mely természetesen futási időben nem látható. Ha erre a kis négyzetre duplán rákattintunk, megnyílik a Menu Designer, amely segítségével könnyen kialakíthatjuk a főmenünket.

Klikkeljünk az új menüpont helyére, majd adjuk meg a menüpont feliratát (**Caption** tulajdonság). Észrevehetjük, hogy minden egyes menüpontnak vannak külön tulajdonságaik. Csak rajtunk múlik, hogy itt beállítjuk-e a **Name** tulajdonságot is (pl. mnuFajl, mnuSzerkesztes), vagy hagyjuk azt, amit a *Delphi* automatikusan hozzárendelt. A menünk a Menu Designer-ben hasonlóan néz ki, mint ahogy ki fog nézni az alkalmazásunkban, azzal a különbséggel, hogy itt láthatjuk azokat a menüpontokat is, melyeknek a **Visible** tulajdonságuk hamis (false).

Minden menüpontnak egyetlen fontos eseménye van, az **OnClick** esemény. Ebben adhatjuk meg azokat a parancsokat, melyeket végre akarunk hajtani, ha a felhasználó rákattint a menüpontra.

Ha valamelyik menüpontból egy új almenüt szeretnénk megnyitni, kattintsunk rá a tervezési időben jobb egérgombbal és válasszuk ki a „Create Submenu”-t.

Ha a menüpontokat el szeretnénk választani egymástól egy vízszintes vonallal, hozzunk létre oda egy új menüpontot és adjunk meg a **Caption** tulajdonságnak egy kötőjelet (-). A menüben ez a menüpont egy vízszintes választóvonalként fog megjelenni.

Ha szeretnénk, hogy a menüpontot az Alt+betű billentyűzetkombinációval is el lehessen érni, a **Caption** tulajdonságban a betű elé tegyünk egy „and” (&) jelet. Például: &File, &Szerkesztés, stb.

A menüpontok fontosabb tulajdonságaik:

- **Checked** – meghatározza, hogy a menüpont ki legyen-e jelölve, pontosabban hogy mellette (a bal oldalán) legyen-e pipa (jelölőpont).
- **Enabled** – meghatározza, hogy a menüpont engedélyezve van-e, vagy szürke és nem lehet rákattintani.
- **GroupIndex** – a menüpontok logikai csoportokba való osztását lehet vele megoldani (az ugyanabba a csoportba tartozó menüpontoknak a GroupIndex-e egyforma, 0-nál nagyobb szám).
- **RadioItem** – segítségével meghatározható, hogy az egy csoportba tartozó menüpontok közül egyszerre csak egy lehet-e kiválasztva (kipipálva). Ha néhány menüpontnak ugyanazt a GroupIndex-et állítjuk be (0-nál nagyobb) és a RadioItem értékét igazra (true) állítjuk, akkor a menüpontok közül egyszerre mindig csak egy lehet kiválasztva. A menüpontra kattintva nekünk kell beállítani a programban a Checked tulajdonságot true-ra, nem jelölődik be automatikusan a menüpontra klikkelve.
- **ShortCut** – a menüpont billentyűzetkombinációját határozza meg. Tervezési időben a billentyűzetkombinációt az Object Inspector-ban egyszerűen kiválaszthatjuk, futási időben a következő példa mutatja, hogyan állíthatjuk be például a Ctrl+C kombinációt:


```
mnuItmMasolas.ShortCut :=
            ShortCut(Word('C'), [ssCtrl]);
```

A rövidítés (billentyűzetkombináció) a menüpont mellé (jobb oldalára) automatikusan kiíródik.

- **Visible** – meghatározza, hogy látható-e a menüpont.

Már szó volt arról, hogy a menüpontokat nem jó gyakran változatni, eltüntetni és megjeleníteni. Mégis előfordulhat, hogy a menüpontokat vagy az egész menüt meg szeretnénk változtatni (például váltás a nyelvi verziók között, vagy ha van egy kezdő és egy haladó felhasználónak készített menünk). Ebben az esetben létrehozunk két menüt (két MainMenu komponenst teszünk a form-ra) és a Form **Menu** tulajdonságában beállítjuk azt, amelyiket éppen használni szeretnénk.

A menüpontok sok metódussal is rendelkeznek. Egyik közülük pl. az **Add** metódus, melynek segítségével futási időben is adhatunk új menüpontot a menü végére. Például:

```
procedure TForm1.Button1Click(Sender: TObject);
var mnItmUj: TMenuItem;
begin
  mnItmUj := TMenuItem.Create(Self);
  mnItmUj.Caption := 'Új menüpont';
  mnItmFile.Add(mnItmUj);
end;
```

Ezzel kapcsolatban felsoroljuk, hogy milyen lehetőségeink vannak, ha futási időben szeretnénk új menüpontokat berakni a menübe:

- Berakhatjuk az új menüpontokat a fent említett módon (hasonlóan az **Add**-hoz létezik **Insert** metódus is, amely segítségével beszúrhatunk menüt máshova is, nem csak a

végére). A problémánk itt az új menüpont **OnClick** eseményéhez tartozó eljárás megadásánál lehet.

- Létrehozunk több menüt (több MainMenu komponenst) és a futási időben ezeket cserélgetjük (a Form Menu tulajdonságának segítségével).
- Létrehozunk egy „nagy” menüt, melybe mindent belerakunk és a menüpontok **Visible** tulajdonságainak segítségével állítgatjuk, hogy melyek legyenek láthatók.

14.2 Lokális (popup) menü – PopupMenu

Manapság már szinte nem létezik olyan alkalmazás, amely ne tartalmazna lokális (popup) menüt. Ezek a menük általában a jobb egérgombbal kattintáskor jelennek meg. Az popup menük varázsa abban rejlik, hogy pontosan azokat a menüpontokat tartalmazza, amelyre az adott pillanatban szükségünk lehet (az aktuális komponenshez vonatkozik).

A *Delphi*-ben popup menüt a **PopupMenu** komponens segítségével hozhatunk létre. Az ilyen menü létrehozása nagyon hasonlít a MainMenu létrehozásához, ezért itt ezt nem részletezzük.

Amivel foglalkozni fogunk, az az, hogy hogyan lehet bebiztosítani, hogy mindig a megfelelő popup menü jelenjen meg. A megjelenítendő popup menüt a form-on levő komponensekhez tudjuk külön-külön beállítani, mégpedig a komponensek **PopupMenu** tulajdonságával (egy alkalmazásban természetesen több PopupMenu-nk is lehet).

A PopupMenu a MainMenu-hoz képest egy új tulajdonsággal, egy új metódussal és egy új eseménnyel rendelkezik. Az új tulajdonsága az **AutoPopup**. Ha ennek értéke igaz (true), akkor a menü automatikusan megjelenik a komponensre kattintáskor a jobb egérgombbal, ahogy azt megszoktuk más programokban. Ha a tulajdonság értéke hamis (false), akkor a menü nem jelenik meg automatikusan, hanem azt a programkódban a **Popup** metódus segítségével jeleníthetjük meg. A PopupMenu komponens új eseménye az **OnPopup**. Ez az esemény pontosan az előtt következik be, mielőtt megjelenne a popup menü. Itt tehát még letesztelhetünk valamilyen beállításokat, majd azok szerint beállíthatjuk a menüpontokat.

15 Objektum orientált programozás

Ez a programozási stílus különálló objektumokat használ, melyek tartalmazzák (magukban zárják) az adataikat és a programkódjaikat is. Ezek az objektumok az alkalmazás építőelemei. Az objektumok használata lehetővé teszi az egyszerűbb beavatkozást a programkódba. Továbbá mivel az adatokat és a programkódot is együtt tartalmazza az objektum, ezért a hibák eltávolítása és az objektumok tulajdonságainak változtatása minimális hatással van a többi objektumra.

Egy objektumnak négy jellemző tulajdonsága van:

- **Adat és kód kombinációja**: Az objektum egyik alkotóelem az adat (vagy adatszerkezet), a másik a kód. A kettőnek elválaszthatatlan egészén értjük az objektumot.
- **Öröklés**: Lehetőségünk van új objektumok létrehozására létező objektumokból. Az új objektum a létező objektum összes mezőjét (adat) és metódusát (kód) örökli, de rendelkezhet további adatmezőkkel és metódusokkal is.
- **Polimorfizmus**: Az utód örökölt metódusait a régi helyett új utasítás sorozattal láthatjuk el. Tehát ugyanolyan nevű függvényt vagy eljárást deklarálhatunk, amilyen az ősben szerepel (azért, hogy felülírjuk a régit).
- **Zártság**: A polimorfizmus megengedi ugyanazt a metódust „kicserélni” egy új metódusra. Ezután a zártság (tehát hogy mennyire zárt az osztály) két szálon futhat tovább. Az egyik szál – az öröklésre vonatkoztatva – a statikus, a másik a virtuális.

A **statikus metódusok** az öröklésкor csupán kicserélik az elől metódusát újra, nincs hatással az objektum más részeire – így nem változik meg teljesen annak tulajdonsága (statikus kötődés). Gondoljunk itt pl. az objektum más részében elhelyezkedő, esetleg őt meghívó más metódusokra, akik nem az újat, hanem a régit fogják meghívni, a statikus megoldás következménye képen.

A **virtuális metódusok** segítségével lehet megoldani az öröklés folyamaton keresztül a sokoldalúságot. Ez azt jelenti, hogy nem csak a régi metódust cseréli ki az újra, hanem az egész objektumot „átnézve” a régi metódusra mutató összes hivatkozást átírja az új metódusra mutatóvá (dinamikus kötődés). Ezáltal megváltozik az egész objektum tulajdonsága, és az öröklés folyamatra nézve sokoldalúvá válik.

Az objektum orientált programozás két legfontosabb szakkifejezése az osztály és az objektum.

Az **osztály** egy adattípus, melyet úgy képzelhetünk el, mint bizonyos objektumok sablonját (pl. autók), és amely meghatározza a konkrét objektumok viselkedését. Az osztály tartalmazhat valamilyen adatokat (adatmezők) és metódusokat (eljárások, függvények). Az osztálynak jellemeznie kéne a viselkedését és a tulajdonságait több hasonló objektumnak (pl. különféle autótípusoknak).

Az **objektum** az osztály egy konkrét előfordulása (pl. az autó egy konkrét, fizikailag létező példánya). Az objektum a program futásakor memóriát foglal le.

Az objektum és osztály közötti összefüggést úgy képzelhetjük el, mint a változó és az adattípus közti összefüggést.

Ahhoz, hogy mindent jobban megértsünk, létrehozunk egy autó osztályt (TAuto), melynek következő mezői lesznek:

- **típusa** – az autó típusa – szöveg típusú;
- **gyártási éve** – egész szám típusú;
- **benzin** – a tartályban levő benzin mennyisége – egész szám típusú;
- **kapacitás** – a tartály űrtartalma – egész szám típusú.

Az osztálynak a következő metódusai lesznek:

- **információk kiírása** – ez az eljárás kiírja az összes adatmezőt;
- **tankolj** – ez a függvény a megadott mennyiségű benzinnel feltölti a tartályt. Ha a megadott mennyiség nem fér a tartályba, feltölti a maximumra, ami még belefér és hamis (false) értéket ad vissza.

Hozzunk létre a *Delhi*-ben egy új alkalmazást, melynek ablakán kezdetben egyetlen gomb legyen. Pelda08

Írjuk be a következő kódot a modulunk ***interface*** részébe:

```
type
 TAuto = class
   Tipus: String;
   GyartasiEv, Benzin, Kapacitas: Integer;
   procedure InfoKiir;
   function Tankolj(Mennyit: Integer): Boolean;
 end;
```

Ahhoz, hogy ezzel az osztállyal tudjunk dolgozni, meg kell adnunk a metódusok (procedure, function) programkódját is. Ezt az ***implementation*** részben adjuk meg. Továbbá ahhoz, hogy a kompilátor tudja melyik osztályhoz tartoznak ezek a metódusok, a metódus elé ponttal elválasztva megadjuk az osztály nevét:


```
procedure TAuto.InfoKiir;
begin
  ShowMessage(Format('%s, %d: %d (%d).',
    [Tipus, GyartasiEv, Benzin, Kapacitas]));
end;

function TAuto.Tankolj(Mennyit: Integer):
                                    Boolean;
begin
  Result := (Benzin + Mennyit) <= Kapacitas;
  Benzin := Min(Kapacitas, (Benzin + Mennyit));
end;
```

A **Result** változót használhatjuk a *Delhi*-ben a függvény értékének visszaadására (a *Delphi*-ben nem szokták használni a klasszikus „pascalos” felírást: funkcio_neve := visszaadási_érték).

A **Min** függvény (argumentumokban megadott számok közül a kisebbet adja vissza) a Math unitban található, ezért használatához ki kell egészítenünk a modulunk uses részét ezzel a unittal.

Most deklarálunk egy TAuto osztály típusú változót és megmutatjuk, hogyan hívhatjuk meg a metódusait, hogyan dolgozhatunk a változóval (pontosabban objektummal). Az alábbi kódot az *implementation* részben bármilyen eljárásba vagy függvénybe beletehetjük. Mi például a nyomógomb OnClick eseményének kezelését megvalósító eljárásba tesszük bele:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  EnAutom: TAuto;
begin                          // (A)
  EnAutom.Tipus := 'Skoda';  // (B)
  EnAutom.GyartasiEv := 1950;
  EnAutom.Benzin := 0;
  EnAutom.Kapacitas := 5;
  EnAutom.InfoKiir;
  if not EnAutom.Tankolj(2) then
    ShowMessage('Ne vidd túlzásba a
                              tankolást!');
```

```
  EnAutom.InfoKiir;
end;
```

Ha most elmentjük és lefuttatjuk az alkalmazásunkat, első pillanatban minden működik mindaddig, amíg nem nyomjuk meg a nyomógombot. Ekkor hiba (kivétel) következik be. Hogy miért? A hiba elmagyarázása egy kicsit bonyolult és összefügg az objektum orientált modell alapgondolatával – megértéséhez kell valamit mondanunk az objektumok létrehozásáról. Az alábbi néhány sor kulcsfontosságú az objektum orientált programozás megértéséhez!

Az objektum orientált modell alapgondolata abban rejlik, hogy az osztály típusú változó (most nem az objektumról beszélünk, csak a változóról), amely a fenti példában az EnAutom, nem tartalmazza az objektum „értékét". Nem tartalmazza sem az auto objektumot sem az auto mezőit. Csupán egy hivatkozást (mutatót) tartalmaz a memóriának arra a helyére, ahol az objektum fizikailag megtalálható. Amikor a változót úgy hozzuk létre, ahogy azt a fenti példában tettük (a var szócska segítségével), akkor nem hozzuk létre az objektum fizikai reprezentációját (nem hozzuk létre azt a helyet a memóriában, ahová az objektumot eltehetjük), hanem csak egy hivatkozást az objektumra (a memóriában csak azt a helyet hozzuk létre, ahová a hivatkozást tehetjük el – tehát ahol csak egy memóriacímet tárolhatunk)! Magát az objektumot nekünk kell létrehoznunk az osztály **Create** metódusának a segítségével, melyet konstruktor-nak neveznek (ez az eljárás szolgál a memória lefoglalására és az objektum inicializálására).

A megoldás tehát az, hogy az (A) és (B) betűvel jelölt sorok közé az előző eljárásban beszúrjuk a konstruktor meghívását:

```
…
begin                          // (A)
  EnAutom := TAuto.Create;
```



```
  EnAutom.Tipus := 'Skoda';  // (B)
…
```

Honnan lett az osztályunknak Create konstruktora? Ez valójában a TObject konstruktora, amelytől az összes többi osztály (tehát ez is) örökli.

Miután az objektumot létrehoztuk és használtuk, meg is kell szüntetnünk a végén. Ezt a **Free** metódus segítségével tehetjük meg:

```
…
  EnAutom.InfoKiir;
  EnAutom.Free;
end;
```

15.1 Konstruktor

A **Create** metódust a memória lefoglalása végett hívtuk meg. Gyakran azonban az objektumot inicializálni is kell. Általában ezzel a céllal teszünk az osztályunkba konstruktort. Használhatjuk a Create metódus megváltoztatott verzióját, vagy definiálhatunk egy teljesen új konstruktort is. Nem ajánlatos azonban a konstruktort másként elnevezni, mint Create.

A konstruktor egy specifikus eljárás, mivel a *Delphi* maga foglalja le annak az objektumnak a memóriát, melyen a konstruktort meghívjuk. A konstruktor tehát megoldja helyettünk a memória lefoglalással kapcsolatos problémákat. A konstruktort a **constructor** kulcsszó segítségével deklarálhatjuk. Tegyünk tehát konstruktort a TAuto osztályba:

```
type
 TAuto = class
   Tipus: String;
   GyartasiEv, Benzin, Kapacitas: Integer;
```

```
    constructor Create(TTipus: String;
     GGyartasiEv, BBenzin, KKapacitas: Integer);
    procedure InfoKiir;
    function Tankolj(Mennyit: Integer): Boolean;
  end;
```

Meg kell adnunk a konstruktorhoz tartozó programkódot is. Ezt bárhova beírhatjuk a modulunk *implementation* részébe akár két eljárás közé is, de ajánlott első eljárásként feltüntetni rögtön az *implementation* után. Ha helyesen akarunk eljárni, akkor a konstruktorban mindig először meg kell hívnunk az ősének a konstruktorát (**inherited Create;**) és utána feltüntetnünk a saját utasításainkat. A TObject-től közvetlenül származó osztályban az ős konstruktorának hívása nem szükséges, de ennek ellenére jó és formálisan is helyes, ha ott van.

```
constructor TAuto.Create(TTipus: String;
    GGyartasiEv, BBenzin, KKapacitas: Integer);
begin
  inherited Create;
  Tipus := TTipus;
  GyartasiEv := GGyartasiEv;
  Benzin := BBenzin;
  Kapacitas := KKapacitas;
end;
```

A Button1Click eljárásunk, melyben az EnAutom objektummal dolgozunk, ezek után így módosul:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  EnAutom: TAuto;
begin
  EnAutom := TAuto.Create('Skoda', 1950, 0, 5);
  EnAutom.InfoKiir;
  if not EnAutom.Tankolj(2) then
    ShowMessage('Ne vidd túlzásba a
                                tankolást!');
  EnAutom.InfoKiir;
  EnAutom.Free;
end;
```


15.2 Destruktor, free metódus

A destruktor egyszerűen fogalmazva a konstruktor ellentettje. A destruktor neve általában mindig **Destroy**. A destruktor feladata az objektum megszüntetése, a lefoglalt memória felszabadítása.

Ez előző eljárásban már találkoztunk egy metódussal, amely a destruktort hívja meg (EnAutom.Free). Ez a **Free** metódus leelenőrzi, hogy az adott objektum létezik-e (nem NIL-re mutat-e a mutató), majd meghívja az objektum destruktorát.

15.3 Hozzáférés az adatokhoz

Az osztály elméletileg bármennyi adatot és metódust tartalmazhat. A helyesen megalkotott osztály adatainak rejtve kéne maradnia az osztályon belül. Az a jó, ha ezekhez az adatokhoz csak az osztály metódusainak a segítségével lehet hozzáférni. Nem ajánlatos hogy bárki tudjon manipulálni az osztály adataival. Egyik alapelve az objektum orientált programozásnak, hogy „mindenki a saját adataiért a felelős”.

Az optimális hozzáférés tehát a következő: „kívülről” nem lehet az adatokhoz közvetlenül hozzáférni, de rendelkezésre állnak azok a metódusok, melyek ezt a hozzáférést (olvasást, írást) bebiztosítják. Így be van biztosítva az autorizált hozzáférés az adatokhoz.

Az Objekt Pascal-ban ezt a következő hozzáférési kulcsszavak használatával érhetjük el:

- **public** – ebben a részben elhelyezett adatmezők és metódusok az objektumon kívülről is, bárhonnan elérhetők, ahol maga az objektum típus is „látható”.
- **private** – az adatmezők és metódusok elérése csak az objektum-osztály „belsejére” korlátozott. Az itt felsorolt adatmezők és metódusok kívülről nem érhetők el.
- **protected** – kizárólag az objektumon „belülről”, azaz csak magában az objektum típusban ill. annak leszármazottaiban is, azok metódusaiban is elérhetők.
- **published** – az itt szereplő adatmezők és metódusok nem csak a program futásakor, de az alkalmazás létrehozásakor is elérhetők. Az alkalmazás szemszögéből hasonlóan bárhonnan látható, mint a public részben szereplő adatmezők és metódusok.

Ahhoz, hogy az objektumokat megfelelően tudjuk használni és programunk is áttekinthetőbb legyen, **az osztályok definícióját és metódusainak implementációját ajánlott mindig külön unit-ban tárolni**. Így biztosítva van az adatokhoz való megfelelő hozzáférést is, ugyanis abban a unitban, ahol az osztályt definiáljuk bármelyik adatmező és metódus elérhető az objektumon kívülről is (még a private típusú is), de a többi unitban már csak azok, amelyek engedélyezve vannak a fenti négy kulcsszó segítségével úgy, ahogy azt leírtuk.

Új unitot a **File – New – Unit - Delphi for Win32** menüpont segítségével hozhatunk létre. Ne felejstük el a programunk (főablakhoz tartozó modulunk) **uses** részébe beírni ennek az új unitnak a nevét.

Hozzunk tehát létre egy új unitot, adjunk neki a mi alkalmazásunkban **AutoUnit** nevet és mentsük el ezen a néven. Ebbe

rakjuk át a TAuto osztály definícióját és az osztály metódusainak implementációját. Az első (főablakhoz tartozó) unit uses részét egészítsük ki az AutoUnit modullal:

```
uses Windows, Messages, …, AutoUnit;
```

Az AutoUnit-ot pedig egészítsük ki a

```
uses Dialogs, SysUtils, Math;
```

sorral. Ez utóbbira azért van szükség, mert a Dialogs unitban található a ShowMessage eljárás, a SysUtils-ban a Format függvény és a Math unitban a Min függvény, amelyeket használunk az osztályunk metódusaiban.

Mostantól a saját osztályaink definícióit és a hozzájuk tartozó metódusok implementációit mindig az AutoUnit modulba írjuk majd. A másik modulban marad a TForm1 osztály definíciója és a form-on található komponensek eseményeihez tartozó eljárások (a mi esetünkben egyelőre csak a Button1 Onclick eseményét kezelő eljárás).

Megmutatjuk, hogyan lehet bebiztosítani az autorizált hozzáférést a mi TAuto osztályunkon:

```
type
 TAuto = class
  protected
   Tipus: String;
   GyartasiEv, Benzin, Kapacitas: Integer;
  public
   constructor Create(TTipus: String;
    GGyartasiEv, BBenzin, KKapacitas: Integer);
   procedure InfoKiir;
   function Tankolj(Mennyit: Integer): Boolean;
 end;
```

15.4 Öröklés

Az öröklés az objektum orientált programozás tulajdonsága, melyet kihasználhatunk, ha egy új osztályt szeretnénk létrehozni egy létező (kész) osztály mintája alapján, de az új osztályt további adatmezőkkel és metódusokkal is el szeretnénk látni.

Példaként hozzunk létre egy teherautó osztályt az auto osztály segítségével (a TAuto osztályból való örökléssel). A teherautó osztálynak lesz egy plusz mezője: a **teherbírása**, és természetesen lesz saját konstruktora (**Create**), melynek paraméterében megadjuk a teherbírást is. Az **InfoKiir** metóduson is változtatunk (létrehozunk újat), mivel a teherautó teherbírását is ki szeretnénk írni.

```
Type
 TTeherauto = class(TAuto)
  private
   Teherbiras: Integer;
  public
   constructor Create(TTipus: String;
              GGyartasiEv, BBenzin, KKapacitas,
              TTeherbiras: Integer);
   procedure InfoKiir;
  end;
```

A megváltozattott eljárások programkódjai:

```
constructor TTeherauto.Create(TTipus: String;
            GGyartasiEv, BBenzin, KKapacitas,
            TTeherbiras: Integer);
begin
  inherited Create(TTipus,
           GGyartasiEv, BBenzin, KKapacitas);
  Teherbiras := TTeherbiras;
end;

procedure TTeherauto.InfoKiir;
begin
  ShowMessage(
```



```
    Format('%s, %d: %d (%d). Teherbiras = %d',
    [Tipus, GyartasiEv, Benzin, Kapacitas,
                              Teherbiras]));
end;
```

Ezek után így dolgozhatunk az új osztállyal:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  EnTeherautom: TTeherauto;
begin
  EnTeherautom := TTeherauto.Create('Avia',
                             1980, 20, 200, 10);
  EnTeherautom.InfoKiir;
  if not EnTeherautom.Tankolj(2) then
    ShowMessage('Ne vidd túlzásba a
                                    tankolást!');
  EnTeherautom.InfoKiir;
  EnTeherautom.Free;
end;
```

Megjegyzés: az utód osztály helyett bármikor használhatjuk az ősének az osztályát. Fordított eset nem lehetséges. Például:

```
EnAutom := EnTeherautom;  // lehet
EnTeherautom := EnAutom;  // NEM lehet, hiba!!!
```

15.5 Polimorfizmus, virtuális és absztrakt metódusok

A Pascal függvények és eljárások általában *statikus* kötődésen alapulnak. Ez azt jelenti, hogy a metódus hívása már a fordítónál (és linkernél) „meg van oldva”. Az összes metódus a fenti példákban ***statikus kötődésen*** alapul. Az objektum orientált programozási nyelv azonban más, ***dinamikus kötődést*** is lehetővé tesz.

Az ilyen hozzáférés előnye a **polimorfizmus** néven ismert. Tegyük fel, hogy a mi két osztályunk (TAuto, TTeherauto) a kötődést

dinamikusan definiálja. Ekkor pl. az InfoKiir metódust használhatjuk egy általános változónál (mint pl. az EnAutom), amely a program futása során a két osztály közül bármelyiknek az objektumára hivatkozhat:

```
var
 EnAuto: TAuto;

begin
   …
   EnAutom := TAuto.Create('Skoda',1950,0,5);
   EnAutom.InfoKiir;
   …
   …
   EnAutom := TTeherauto.Create('Avia',1980,
                                    20,200,10);
   EnAutom.InfoKiir;
   …
   …
end;
```

Az, hogy a két ugyanolyan nevű metódus (InfoKiir) közül melyik osztály metódusa lesz meghívva (az, amelyik a teherbírást is kiírja, vagy az amelyik nem), mindig a program futása alatt dől el a konkrét helyzettől függően (pontosabban attól függően, hogy az EnAutom éppen melyik osztály objektumára hivatkozik). ***Statikus*** kötődés esetén mindig a TAuto metódusa lett volna meghívva (teherbírás kiírása nélküli), mivel az EnAutom TAuto típusú.

A ***dinamikus*** kötődést a **virtual** és **override** kulcsszavak használatával definiálhatunk:

```
type
 TAuto = class
   …
   procedure InfoKiir; virtual;
   …
end;

Type
 TTeherauto = class(TAuto)
```



```
   …
   procedure InfoKiir; override;
   …
 end;
```

Az **abstract** kulcsszó segítségével olyan metódusokat deklarálhatunk, melyek csak az utódokban lesznek definiálva. Gyakorlatilag ebből következik, hogy az osztályban nem kell leírni (definiálni) az absztrakt metódus programkódját (testét).

```
type
 TAuto = class
   …
   procedure EvValtoztatasa; virtual; abstract;
   …
 end;
```

16 Az osztályok hierarchiája, VCL

Az összes komponens (TEdit, TLabel, TButton, TCheckBox, ...) a **vizuális komponenskönyvtárban** (Visual Component Library, VCL) van összegyűjtve.

A delphi szíve az **osztályok hierarchiája**. Minden osztály a rendszerben a **TObject** típusú osztály utódja, tehát az egész hierarchiának egyetlen gyökere van. Ezzel meg van engedve a TObject osztály használata bármely más osztály helyett.

A komponensek használatakor valójában az osztályok hierarchiájának a „leveleiből” hozunk létre konkrét objektumokat. Az Object Inspector és az Elempaletta lehetőséget adnak a VCL komponenseinek elhelyezésére a formunkon, majd a komponensek tulajdonságainak változtatására a nélkül, hogy programkódot kellene írnunk.

Megjegyzés: Az eseményekre reagáló metódusok általában tartalmaznak egy TObject típusú Sender paramétert. A fent említett tények miatt ez a paraméter a VCL bármelyik osztályának eleme lehet, mivel minden osztály a TObject osztályból van levezetve. A TObject osztály egy absztrakt osztály, melynek metódusai az osztályok alap viselkedését határozzák meg, amilyen például az objektum létrehozása, megszüntetése, stb.

Hogy jobban megértsük az osztály és objektum közti különbséget, vegyünk rá még egy példát:

```
var
  Valami: TValami;   // TValami – osztály

begin
  Valami := TValami.Create;
       // Valami – újonnan létrehozott objektum
  ... munka az objektummal ...
       // itt dolgozhatunk a Valami-vel
  Valami.Free;
       // objektum megszüntetése
end;
```

Ezzel a módszerrel komponenseket is adhatunk a programunkhoz a program futása alatt (mivel valójában a komponensek is osztályok). Például a form bármelyik metódusában egy új nyomógombot a következő módon hozhatunk létre:

```
var
  btnUj: TButton;
begin
  btnUj := TButton.Create(self);
  btnUj.Parent := self;     // self = form1-ünk
  btnUj.Left := 100;
  btnUj.Top := 200;
  btnUj.Visible := true;
end;
```


A VCL (és a *Delphi*) nagy előnye, hogy a komponensek használatához nem kell ismernünk az osztályok részletes hierarchiáját, elég, ha ismerjük az egyes komponenseket (a hierarchiában a „fa leveit”).

17 Billentyűzet, egér

Ebben a fejezetben a két legfontosabb bemeneti eszközre – a billentyűzetre és az egérre fogunk összpontosítani.

17.1 Az egér

Az egér szempontjából az egyik leggyakrabban használt esemény az **OnClick**, amely szinte minden komponensnél megtalálható. Ez az esemény pontosabban akkor következik be, ha a felhasználó felengedi az egérgombot az adott komponens fölött. Az OnClick esemény azonban máskor is bekövetkezik:

- ha a felhasználó a listában a billentyűzet (nyilak) segítségével kiválasztja az elemet,
- ha a felhasználó a szóközt nyomja meg és az aktív komponens a CheckBox vagy a Button,
- ha a felhasználó megnyomja az Enter billentyűt és az aktív komponens a Button, vagy az aktív ablaknak van „default button”-ja (a nyomógomb Default tulajdonságával lehet beállítani)
- a felhasználó megnyomja az Esc billentyűt és az aktív ablaknak van „cancel button”-ja (a nyomógomb Cancel tulajdonságával állítható be)

Az ablak (form) esetében az OnClick akkor következik be, ha a felhasználó az ablak üres részére klikkel, ahol nincs egyetlen

komponens sem, vagy nem elérhető komponensre (melynek Enabled tulajdonsága false).

Ehhez hasonló az **OnDblClick** esemény, amely duplakattintásnál következik be.

Továbbá rendelkezésünkre áll még az **OnMouseDown**, **OnMouseUp** és az **OnMouseMove** események. Ezek akkor következnek be, amikor a felhasználó lenyomja az egérgombot, felengedi az egérgombot, illetve mozgatja az egérkurzort. A három esemény eljárásainak paraméterei hasonlóak (csak az OnMouseMove érthető okokból nem tartalmazza a lenyomott egérgombot megadó paraméterét). Nézzük milyen paraméterei vannak az eljárásoknak (szemléltetjük az OnMouseDown eljáráson):

```
Procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
```

A `Sender` paraméter azt a komponenst jelöli, amelyen bekövetkezett az esemény.

A `Button` paraméter megadja, melyik egérgomb volt lenyomva (melyik egérgomb lenyomásával következett be az esemény). Lehetséges értékei: mbLeft (bal egérgomb), mbRight (jobb egérgomb), mbMiddle (középső egérgomb).

A `Shift` paraméter megadja némely gombok állapotát az esemény bekövetkezésekor. Továbbá az egér saját gombjainak az állapotát is jelzi. Ez a paraméter egy halmaz, mivel egyszerre több értéket is tartalmazhat. Lehetséges értékei: ssShift (Shift billentyű le volt nyomva), ssAlt (Alt billentyű), ssCtrl (Ctrl billentyű), ssLeft (bal egérgomb), ssRight (jobb egérgomb), ssMiddle (középső egérgomb), ssDouble (duplakattintás következett be).

Az `X`, `Y` paraméterek az egérkurzor koordinátáit adják meg azon a komponensen belül, amelyen az esemény bekövetkezett. A koordináták képpontokban (pixelekben) vannak megadva a komponens bal felső sarkához viszonyítva.

Rendelkezésünkre állnak még az **OnMouseWheel**, **OnMouseWheelDown** és az **OnMouseWheelUp** események, melyek segítségével az egér görgetőgombjával dolgozhatunk. Az első esemény akkor következik be, ha a görgetőt bármelyik irányba görgetjük, a második ha lefelé, a harmadik ha felfelé görgetjük. Az események eljárásaiban a `WheelDelta` paraméterből megtudhatjuk, mennyivel görgettük a görgőt. A `Shift` paraméter hasonló mint az előző eseményeknél, a `MousePos` paraméter pedig megadja az egér kurzorának a koordinátáit.

Ha ezek az események és paramétereik nem elég nekünk valamilyen művelet elvégzéséhez, akkor használhatjuk a TMouse osztályt is, mely tartalmaz további információkat is az egérről.

Ha ki szeretnénk használnia TMouse osztályt, elég használnunk a **Mouse** globális változót. Ennek a változónak van néhány tulajdonsága, melyekből minden fontosat megtudhatunk az egérről.

Ezek közül a legfontosabb a **MousePresent**, melynek értéke igaz (true), ha van egér a rendszerben (be van telepítve). A többi tulajdonságból megemlítjük még a **CursorPos**-t (egér koordinátái a képernyőhöz viszonyítva), **WheelPresent**-et (van-e görgetője az egérnek) és a **WheelScrollLines**-t (sorok száma, amennyivel a szöveg elmozduljon a görgetésnél).

17.2 Billentyűzet

A billentyűzettel való munka során leggyakrabban az **OnKeyDown**, **OnKeyUp** és **OnKeyPress** eseményeket használjuk.

Az **OnKeyPress** a billentyű megnyomásakor következik be. Az esemény kezelésében rendelkezésünkre áll a `Key` paraméter, amely Char típusú és a lenyomott billentyű ASCII kódját tartalmazza. Azok a billentyűk, melyeknek nincs ASCII kódjuk (pl. Shift, Ctrl, F1, …) nem generálnak OnKeyPress eseményt. Tehát pl. a Shift+A megnyomásakor egyetlen OnKeyPress esemény következik be.

Ha azokat a billentyűket szeretnénk figyelni, melyeknek nincs ASCII kódjuk, akkor az **OnKeyDown** ill. **OnKeyUp** eseményeket kell használnunk. Az első akkor következik be, ha a felhasználó lenyom egy billentyűt, a második amikor felengedi. Mindkét esemény kezelésének eljárásában van `Key` paraméter, amely a lenyomott billentyű kódját tartalmazza. Ez itt egy virtuális billentyűkód, pl. VK_Control (Ctrl), VK_Back (Backspace), stb. Továbbá használhatjuk az eljárás `Shift` paraméterét is, amely megadja, hogy a Shift, Ctrl, Alt gombok közül melyik választógomb volt lenyomva az esemény bekövetkezésekor.

17.3 Példaprogramok az egér és a billentyűzet használatára

Az egér melyik nyomógombjával volt kattintva? Pelda09

Az alábbi példa bemutatja az OnMouseDown esemény használatát. Minden egérkattintásnál a form-ra (ablakra) kiírja, melyik egérgombbal történt a kattintás.

```
Procedure TForm1.FormMouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer);
begin
  case Button of
    mbLeft: ShowMessage('Bal egérgomb.');
    mbRight: ShowMessage('Jobb egérgomb.');
    mbMiddle: ShowMessage('Középső egérgomb.');
  end;
end;
```

Meg volt nyomva a Shift a dupla egérkattintásnál? Pelda10

Az alábbi példában az OnMouseDown esemény segítségével megállapítjuk, hogy meg volt-e nyomva a Shift billentyű, amikor az egérrel duplán kattintottunk a form-on.

```
Procedure TForm1.FormMouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer);
begin
  if (ssShift in Shift) and (ssDouble in Shift) then
    ShowMessage('Shift + dupla kattintás');
end;
```

Az egérkurzor koordinátáinak kiírása. Pelda11

Az OnMouseDown esemény segítségével kiírjuk az ablak azon pontjának koordinátáit, ahová az egérrel kattintottunk.

```
Procedure TForm1.FormMouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState;
```



```
      X, Y: Integer);
begin
  ShowMessage('Koordináták: X=' + IntToStr(X) +
              ', Y=' + IntToStr(Y));
end;
```

Koordináták kiírása a képernyőhöz viszonyítva. Pelda12

Az előző példában a koordinátákat az ablakhoz viszonyítva írtuk ki. Ha az egész képernyőhöz viszonyítva szeretnénk megtudni a koordinátákat, akkor erre a **ClientToScreen** metódust használhatjuk.

```
Procedure TForm1.FormMouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer);
var
  Pont: TPoint;
begin
  Pont := ClientToScreen(Point(X,Y));
  ShowMessage('Koordináták: X=' + IntToStr(Pont.X) +
              ', Y=' + IntToStr(Pont.Y));
end;
```

Van-e egér a rendszerben? Van-e görgetőgombja? Pelda13

Ahhoz, hogy megtudjuk van-e egér telepítve az operációs rendszerben, a globális **Mouse** változót fogjuk használni. Ha van egér, akkor hasonlóan megállapítjuk van-e görgetőgombja.

```
Procedure TForm1.FormCreate(Sender: TObject);
begin
  if not Mouse.MousePresent then
    begin
```

```
      MessageDlg('Hiba: Nincs egér. Az alkalmazás
        leáll.', mtError, [mbOk], 0);
      Application.Terminate;
    end
  else
    if Mouse.WheelPresent then
      MessageDlg('Info: az egérnek van görgetője.',
        mtInformation, [mbOk], 0);
end;
```

Billentyűzetről bemenet kódolása. Pelda14

Az alábbi példa szemlélteti a billentyűzettel való munkát. Az alkalmazás egy beviteli szövegdobozt tartalmaz, ahová a felhasználó megadhat valamilyen szöveget. A szöveg azonban nem jelenik meg úgy ahogy azt a felhasználó megadja, hanem „elkódolt" formában íródik a szövegdobozba. A nyomógomb megnyomásával a szöveg dekódolódik olvaható formába. A kódolás a mi példánkban úgy fog történni, hogy eggyel nagyobb ASCII kódú jelet írunk ki. A dekódolás ennek ellentettje lesz.


```
procedure TForm1.Edit1KeyPress(Sender: TObject;
            var Key: Char);
begin
  Key := Chr(Ord(Key)+1);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
```



```
  ret: String;
  i: Integer;
begin
  ret := Edit1.Text;
  for i := 1 to Length(ret) do
    ret[i] := Chr(Ord(ret[i])-1);
  Edit1.Text := ret;
end;
```

17.4 Drag & Drop – fájlok tartalmának megtekintése

A következő példa bemutatja, hogyan használhatjuk alkalmazásunkban a drag and drop műveletet. Pelda15

Az alkalmazásunk szöveges fájlok kiírását (megjelenítését) fogja lehetővé tenni a drag-and-drop művelet segítségével. A felhasználó a kiválasztott állományt meg tudja majd fogni a FileListBox komponensben és áthúzni a Memo komponensbe, ahol a fájl tartalma megjelenik. Az alkalmazás létrehozásához fogjuk használni a DirectoryListBox, FileListBox és Memo komponenseket. Kezelni fogjuk a Form1: **OnCreate**, Memo1: **OnDragOver**, **OnDragDrop** és a FileListBox: **OnEndDrag** eseményeit.

Figyelmeztetés: az **OnStartDrag** esemény minden egyes bal egérgomb lenyomáskor bekövetkezik. Ebben a pillanatban van ugyanis inicializálva a drag-and-drop művelet. A valódi indítása a műveletnek

azonban nem kell hogy azonnal bekövetkezzen, hanem bekövetkezhet például csak az egér elmozdításánál bizonyos számú képponttal. Az **OnEndDrag** esemény bekövetkezésekor tehát lehetséges, hogy a drag-and-drop művelet egyáltalán nem is volt elindítva (csak inicializálva volt a bal egérgomb megnyomásakor). Azt, hogy a művelet el volt-e indítva (pontosabban hogy fut-e), megtudhatjuk a **Mouse.IsDragging** globális objektum változójából, melynek ebben az esetben true értéke lesz.

Ahhoz, hogy a DirectoryListBox komponensben a mappa változtatásakor a FileListBox komponenst tartalma automatikusan megváltozzon, be kell állítanunk a **DirectoryListBox1.FileList** tulajdonságát. Ennek a tulajdonságnak hivatkozást kell tartalmaznia FileListBox komponensünkre (ezt beállíthatjuk az Object Inspector-ban is, mi a Form1 OnCreate eseményében állítjuk be).

Az egyes eseményekhez tartozó eljárásokat tartalmazó programrész:

```
…
procedure TForm1.FormCreate(Sender: TObject);
begin
 DirectoryListBox1.FileList := FileListBox1;
 FileListBox1.DragMode := dmAutomatic;
 Mouse.DragImmediate := false;
end;

procedure TForm1.FileListBox1EndDrag(Sender,
                 Target: TObject; X, Y: Integer);
begin
 if Mouse.IsDragging then
   if (Target <> nil) then
     ShowMessage('Az állomány sikeresen át
                       lett húzva a komponensbe.')
   else
     ShowMessage('Az állományt nem sikerült
                                       áthúzni!');
```



```
end;

procedure TForm1.Memo1DragOver(Sender,
           Source: TObject; X, Y: Integer;
           State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TFileListBox;
end;

procedure TForm1.Memo1DragDrop(Sender,
           Source: TObject; X, Y: Integer);
begin
  if Source is TFileListBox then
    Memo1.Lines.LoadFromFile(FileListBox1.FileName);
end;
…
```

Az ablak OnCreate eseményében beállítottuk a **Mouse.DragImmediate** globális objektum tulajdonságát False-ra. Ezzel elértük, hogy a drag-and-drop művelet nem indul el rögtön az egérgomb lenyomása után, hanem csak akkor, ha az egérkurzor egy megadott távolságot tesz meg. Ez a távolság alapértelmezett beállításban 5 pixel, értékét a **Mouse.DragThreshold** tulajdonság segítségével változtathatjuk meg.

A drag-and-drop operáció kezdete (inicializálása) abban a pillanatban következik be, amikor a felhasználó lenyomja a bal egérgombot a FileListBox komponensben. Ahhoz, hogy nekünk ezt ne kelljen kezelni az OnMouseDown eseményben (a **BeginDrag** függvény segítségével), beállítjuk a FileListBox **DragMode** tulajdonságát dmAutomatic-ra már az ablak OnCreate eseményében.

Továbbá kezelve van a Memo komponens **OnDragOver** eseménye. Ez az esemény akkor következik be, amikor a komponens fölé húzunk valamilyen objektumot. Az esemény kezelésében az

Accept paraméter értékét állítjuk be True-ra, ha a húzás forrása egy TFileListBox típusú objektum. Ezzel bebiztosítjuk, hogy a Memo komponenst a belehúzott objektumot tudja fogadni (és hajlandó legyen fogadni). Ez látszódik az egérkurzor alakján is.

A másik kezelt esemény a Memo komponens **OnDragDrop** eseménye. Ez akkor következik be, amikor az objektumot elengedjük a komponens fölött. Az esemény kezelésében megint meggyőződünk róla, hogy a belehúzott objektum forrása egy TFileListBox típusú objektum-e. Ha igen, a megadott állományt beolvassuk a **Lines** tulajdonságba.

Az utolsó eseményt csak bemutatás végett kezeljük a programunkba. Ez az esemény a FileListBox **OnEndDrag** eseménye. Ez az esemény akkor következik be, amikor a drag-and-drop művelet befejeződik (akár sikeresen – az objektum fogadva volt, akár sikertelenül – az objektum nem volt fogadva). Itt fontos a **Target** paraméter, melynek értéke sikeres művelet esetén a célkomponenst tartalmazza, sikertelen művelet esetén pedig az értéke nil.

18 Grafika, rajzolás, szöveg kiírása

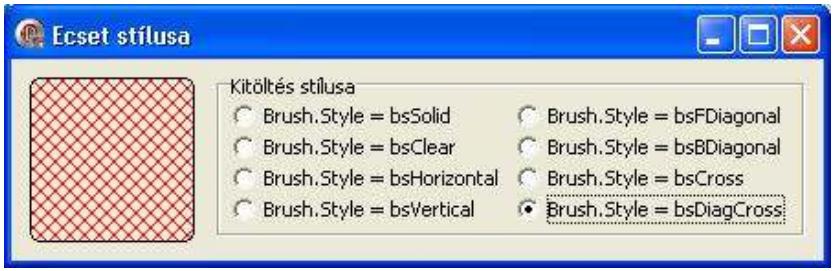
A Delphi-ben van egy alapobjektum a rajzolásra – a vászon (TCanvas osztály). Képzeljük el az ablakunkat (form) úgy, mint egy üres területet, amelyen vászon van. Erre a vászonra (**Canvas**) rajzolhatunk, hasonlóan, mint ahogy a festőművészek is rajzolnak a vászonra.

Canvas objektuma sok grafikus komponensnek van. Vászon van a Form-on, de ugyanúgy megtalálható további komponensekben is, mint pl. az Image-ben és a TBitmap osztályban. Ne feledjük, hogy a vászon nem egy különálló komponens, hanem csak némely komponensek része. A következő felsorolásból megismerhetjük a vászon legfontosabb tulajdonságait:

- **Brush** – ecset. A Brush tulajdonság beállításával változik az alakzatok kitöltésének színe és mintája. Az ecsetnek vannak további tulajdonságai is: **Bitmap** (az ecset mintáját definiáló bitkép), **Color** (szín) és **Style** (stílus).
- **Font** – betű. A Font tulajdonságnak is vannak altulajdonságai: **Color**, **Charset** (karakterkészlet), **Name** (betűtípus neve), **Size** (méret), **Style** (stílus), stb.
- **Pen** – toll. A vászon tollának típusát adja meg. Altulajdonságai: **Color** (szín), **Style** (stílus), **Width** (vonalvastagság), **Mode** (toll rajzolási módja).
- **PenPos** – toll aktuális pozíciója. Ezt a tulajdonságot írni és olvasni is lehet.
- **Pixels** – a pixelek színe. A tulajdonság értékét olvasni és írni is lehet, így rajzolhatunk a vászonra pontonként.

18.1 Ecset stílusa

Az első program ebben a fejezetben bemutatja, hogyan lehet a vászonra kirajzolni egyszerű geometriai alakzatot megadott kitöltési stílussal. Pelda16



Az egyes események kezelésének programkódja:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 RadioGroup1.Columns := 2;
 RadioGroup1.ItemIndex := 0;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Style :=
               TBrushStyle(RadioGroup1.ItemIndex);
  Canvas.Brush.Color := clRed;
  Canvas.RoundRect(10,10,100,100,10,10);
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  Repaint;
end;
```


Maga a kirajzolás az OnPaint eseményben történik. Ez az esemény mindig akkor következik be, ha szükséges átrajzolni az ablakot (pl. ha az ablak el volt takarva másik ablakkal, vagy alkalmazás indításakor, stb.).

Miután a felhasználó rákattint valamelyik választógombra (RadioGroup), kikényszerítjük az ablak átrajzolását a **Repaint** metódus segítségével.

Az alkalmazásban beállítjuk a **Canvas.Brush.Color** és a **Canvas.Brush.Style** tulajdonságok segítségével az ecsetet. Az ecset stílusának beállításánál az ItemIndex aktuális értékét átalakítjuk (áttipizáljuk) TbrushStyle típusra, így rögtön hozzárendelhetjuk a **Brush.Style** tulajdonsághoz.

A négyzet kirajzolását a **RoundRect** (lekerekített sarkú téglalap) metódus segítségével biztosítjuk be. Megpróbálhatunk más alakzatokat is kirajzolni, pl. a **Rectangle** (téglalap), **Pie** (körszelet), **Polygon**, **Polyline**, **Chord**, stb. segítségével.

18.2 Bitmap beolvasása állományból

Megmutatjuk, hogyan olvashatunk be külső állományból egy bitképet és jeleníthetjük meg az ecset segítségével. Az alkalmazás beolvas egy bitképet a tapeta.bmp állományból, majd hozzárendeli az ablak (form) ecsetéhez. Utánna ezzel az ecsettel (tehát a bitmap-pal) kitöltjük az egész ablakot. Most nem lesz szükségünk semmilyen

komponensre. Minden programkódot a form OnPaint eseményének kezelésébe írunk. Pelda17

```
procedure TForm1.FormPaint(Sender: TObject);
var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromFile('tapeta.bmp');
  Canvas.Brush.Bitmap := bmp;
  Canvas.FillRect(Rect(0,0,Width,Height));
  Canvas.Brush.Bitmap := nil;
  bmp.Free;
end;
```

A programban használt **FillRect** metódus a megadott téglalapot kifesti az aktuális ecsettel.

18.3 Szöveg grafikus kiírása

A vászonra nem csak írhatunk, de rajzolhatunk is. A következő alkalmazás egyrészt szemlélteti a szöveg kiírását grafikus módban, másrészt megmutatja, hogyan dolgozhatunk a FontDialog komponenssel (erről bővebben a későbbi fejezetekben lesz szó). Miután a felhasználó ennek a dialógusablaknak a segítségével választ betűtípust, a kiválasztott betűtípussal kiírunk egy szöveget a form-ra. A FontDialog komponensen kívül szükségünk lesz még egy Button komponensre. Az alkalmazásban kezelni fogjuk a Button komponens OnClick eseményét és a Form OnPaint eseményét. Pelda18

Az események eljárásaihoz tartozó programkódok:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    Canvas.Font.Assign(FontDialog1.Font);
  Repaint;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.TextOut(20,50,'Teszt szöveg');
end;
```

A betűtípus kiválasztása után a vászon **Font.Assign** metódusát használjuk, amely az egyik betűtípus összes atribútumát átmásolja a másikba.

A betűtípus változtatásakor meghívjuk az ablak átrajzolására szolgáló **Repaint** metódust.

Az OnPaint eseményben kiírjuk a szöveget a **TextOut** metódus segítségével, melynek első két paramétere a szöveg koordinátáit jelentik.

18.4 Egyszerű grafikus editor

Ennek az alkalmazásnak a segítségével egyszerűen rajzolhatunk bitképeket, megváltoztathatjuk a vonal vastagságát és színét, majd a munkák eredményét elmenthetjük fájlba. Pelda19

Szükségünk lesz a következő komponensekre: 3 x Button, ColorDialog, SavePictureDialog, Image, Label és UpDown.

A következő eseményeit fogjuk kezelni az egyes komponenseknek: OnCreate (Form1), OnMouseDown (Image1), OnMouseMove (Image1), OnClick (UpDown1, Button1, Button2, Button3), OnResize (Form1).

Az egyes eseményekhez tartozó programkód:


```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.DoubleBuffered := true;
  Image1.Canvas.Brush.Color := clWhite;
  Image1.Cursor := crCross;
  UpDown1.Position := 1;
  UpDown1.Min := 1;
  UpDown1.Max := 20;
  SavePictureDialog1.DefaultExt := 'bmp';
end;

procedure TForm1.Image1MouseMove(Sender: TObject;
                  Shift: TShiftState; X, Y: Integer);
begin
  if ssLeft in Shift then
    Image1.Canvas.LineTo(X,Y);
end;

procedure TForm1.Image1MouseDown(Sender: TObject;
                  Button: TMouseButton;
                  Shift: TShiftState; X, Y: Integer);
begin
  Image1.Canvas.MoveTo(X,Y);
end;

procedure TForm1.UpDown1Click(Sender: TObject;
                  Button: TUDBtnType);
begin
  Image1.Canvas.Pen.Width := UpDown1.Position;
  Label1.Caption := 'Vonalvastagság: '
                        + IntToStr(UpDown1.Position);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
    Image1.Canvas.Pen.Color := ColorDialog1.Color;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Image1.Canvas.Brush.Color := clWhite;
  Image1.Canvas.FillRect(Rect(0,0,Image1.Width,
                                    Image1.Height));
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if SavePictureDialog1.Execute then
   try
    Image1.Picture.SaveToFile(
                        SavePictureDialog1.FileName);
   except
    ShowMessage('Hiba a kép mentésénél!');
   end;
end;

procedure TForm1.FormResize(Sender: TObject);
begin
  Image1.Picture.Bitmap.Width := Image1.Width;
  Image1.Picture.Bitmap.Height := Image1.Height;
end;
```

A toll vonalvastagságának beállításához az **UpDown** komponenst használjuk (a Win32 kategóriából).

A rajzolás a következő képpen megy végbe: az egérgomb megnyomásakor a toll pozícióját azokra a koordinátákra állítjuk, ahol a kattintás történt. Az egér mozgatásakor, ha a bal egérgomb lenyomva van, húzunk vonalat az egér koordinátáig a toll előző pozíciójától (ezzel a toll pozícióját is megváltoztatjuk az egér koordinátáira). Tehát a OnMouseDown és OnMouseMove események együttműködését használjuk. Itt fontos metódusok a **MoveTo** (toll pozícióját állítja be) és a **LineTo** (toll mozgatása és egyben vonal rajzolása).

A vonal színének kiválasztásához a ColorDialog komponenst használjuk. A szín kiválasztása után elég ennek a komponensnek a **Color** tulajdonságát hozzárendelnünk a toll színéhez.

A kép mentéséhez a SavePictureDialog komponenst (fájlnév és mentés helyének meghatározásához) és az **Image.Picture.SaveToFile** metódusát használjuk.

18.5 Színátmenet létrehozása

Ebben a részben egy színetmenetet fogunk kirajzolni az ablakunk vásznára. Ez a trükk jól jöhet a jövőben, ha például valamilyen grafikus editort készítünk. Pelda20

Az alkalmazásban az ablak OnCreate, OnPaint és OnResize eseményeihez írjuk meg a programkódot.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := true;
end;

procedure TForm1.FormPaint(Sender: TObject);
var
  deltaR, deltaG, deltaB: Double;
  szinTol, szinIg: TColor;
  i: Integer;
begin
  // kezdeti beállítások
```

```
  szinTol := clRed;
  szinIg := clYellow;
  // egyes színösszetevők növekménye
  deltaR := (GetRValue(szinIg)-GetRValue(szinTol))
                                                / Width;
  deltaG := (GetGValue(szinIg)-GetGValue(szinTol))
                                                / Width;
  deltaB := (GetBValue(szinIg)-GetBValue(szinTol))
                                                / Width;
  // rajzolas
  for i:=0 to Width do
    begin
      Canvas.Brush.Color := RGB (
        Round(deltaR*i+GetRValue(szinTol)),
        Round(deltaG*i+GetGValue(szinTol)),
        Round(deltaB*i+GetBValue(szinTol)));
      Canvas.FillRect(Rect(i,0,i+1,Height));
    end;
end;

procedure TForm1.FormResize(Sender: TObject);
begin
  Repaint;
end;
```

A rajzolás előtt meghatároztuk mennyi az egyes színösszetevőkben (R - piros, G - zöld, B - kék) a különbség a szomszédos pontok között (deltaR, deltaG, deltaB).

A szín változtatását és a kirajzolást egy ciklus segítségével oldottuk meg. Az aktuális szín összetevőit úgy határoztuk meg, hogy a kezdeti szín (szinTol) összetevőjéhez hozzáadtuk a növekmény (deltaX) és a sor elejétől számított képpontok (i) szorzatát. Az eredményt a Round függvénnyel kikerekítettük egész számra.

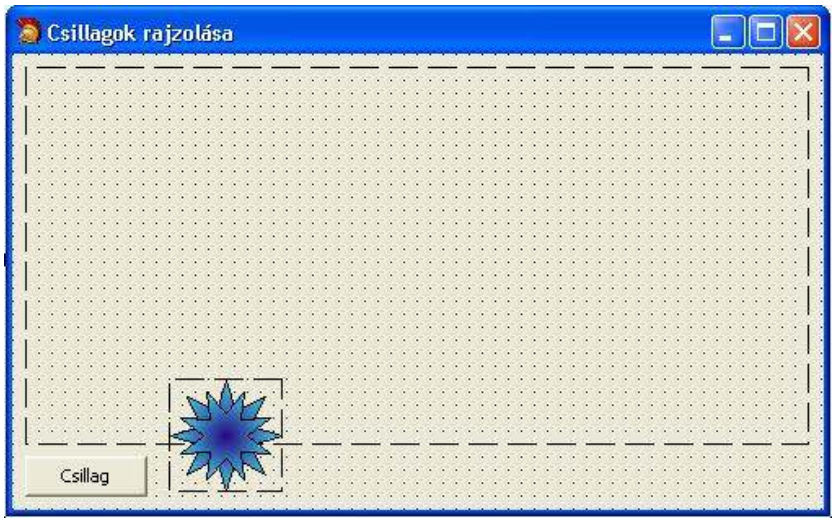
Végül az ablak átméretezésekor bekövetkező eseményhez (OnResize) beírtuk az ablak átrajzolására szolgáló metódust (Repaint).

Az ablak létrehozásánál beállítottuk a DoubleBuffered értékét true-ra azért, hogy az átméretezésnél az átrajzolás ne villogjon.

18.6 Kép kirajzolása megadott koordinátákra

Ebben a programban megismerkedünk azzal, hogyan rajzolhatunk ki egy képet egy Image komponens megadott részére. Ezzel a módszerrel egy kis módosítás után könnyel létrehozhatunk egy olyan rajzprogramot gyerekeknek, melyben pecsételgethetik a kiválasztott képet a megadott helyre. Pelda21

A programunkban használni fogunk két Image komponenst és egy Button komponenst.



A nagyobb **Image1** komponensre fogunk rajzolni, a másik, **Image2** komponens csupán a kép tárolására fog szolgálni. Ebbe, az Image2 komponensbe töltsük be a **Picture** tulajdonság segítségével az a képet, melyet akarunk majd az Image1-re kirajzolni. Továbbá állítsuk be az Image2 komponenst **Visible** tulajdonságát false-ra, hogy a program futásakor ne legyen látható, majd a **Transparent** tulajdonságát true-ra, hogy a képunk háttere átlátszó legyen.

A **Button1** nyomógomb megnyomásakor egy véletlenszerű helyre kirakjuk az Image1 komponensre az Image2-ben tárolt képet. Erre a **Canvas.Draw** metódusát használjuk. Mivel véletlenszerű helyre rajzoljuk ki, ezért a gomb újboli megnyomásakor mindig más helyre fog kirajzolódni a képünk.

Ne felejtsük még beállítani a **Form OnCreate** eseményében a véletlenszám generátor inicializálását (randomize). Itt beállítjuk az Image1 komponens háttérszínét is sötétkékre az RGB függvény segítségével.

Az alkalmazásunkhoz tartozó programkód tehát így néz ki:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  // Hattert kifestjuk sotetkekre
  Image1.Canvas.Brush.Color := RGB(0,0,50);
  Image1.Canvas.FillRect(
            Rect(0,0,Image1.Width,Image1.Height));
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  x,y: Integer;
begin
  // Kiszamitjuk, hova fogjuk kirajzolni a csillagot
```

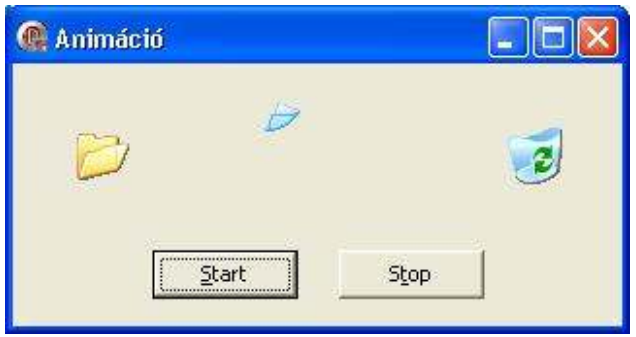


```
  x := Random(Image1.Width)
            - Image2.Picture.Graphic.Width div 2;
  y := Random(Image1.Height)
            - Image2.Picture.Graphic.Height div 2;
  // Kirajzoljuk a csillagot az X, Y koordinatakra
  Image1.Canvas.Draw(x,y,Image2.Picture.Graphic);
end;
```

18.7 Animáció megjelenítése

A Delphi-ben nem csak statikus grafikát, de animációt is megjeleníthetünk. Erre szolgál az **Animate** komponens (a Win32 kategóriából). Ennek segítségével nem csak AVI állományokat játszhatunk le, de lehetőségünk van néhány rendszeranimáció lejátszására is (pl. fájl törlésekor, áthelyezésekor megjelenő animációk).

Hozzunk létre egy új alkamazást, helyezzünk el rá egy Animate komponenst és két nyomógombot. A nyomógomvok segítségével elindíthatjuk ill. megállíthatjuk majd az animációt. Pelda22



Az egyes eseményekhez tartozó programkód:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Animate1.CommonAVI := aviRecycleFile;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Animate1.Active := true;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Animate1.Stop;
end;
```

A **CommonAVI** tulajdonság segítségével választhatunk a standard, előre definiált animációk közül. Az aviRecycleFile értéken kívül felvehet például aviFindFile, aviFindComputer, aviCopyFile, stb. értékeket.

Az Animate komponens segítségével AVI állományok is lejátszhatók. Ebben az esetben az álloányt az útvonallal együtt a **FileName** tulajdonságban kell megadni.

19 Hibák a program futásakor, kivételek kezelése

Amint sejtjük, hogy a program egy adott részében előfordulhat hiba a futása közben, ezt a hibát megfeleően kezelnünk kell még akkor is, ha a hiba csak nagyon ritkán, kis valószínűséggel fordul elő.

Létrehozunk egy eljárást, amely segítségével szemléltetni fogjuk az egyes hibakezelési lehetőségeket. Hozzunk létre egy ablakot (form), tegyünk rá egy nyomógombot (button), majd a nyomógomb OnClick eseményének kezelésébe írjuk be az alábbi programrészt:

```
procedure Form1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 0;
  b := 0;
  c := a div b;
  Button1.Caption := IntToStr(c);
end;
```

A program kiszámolja, majd beírja a nyomógomb feliratába két szám egész részű hányadosát (div).

Ebben a programban nincs kezelve semmilyen hiba. Mivel az a, b változókba 0-t tettünk a program elején, nyilvánvaló, hogy a hiba bekövetkezik (osztás nullával). Ez a hiba egy kivételt eredményez, melyet a div művelet generál. Ha a programot lefuttatjuk és megnyomjuk a nyomógombot, láthatjuk az üzenetet a kivételről. Még ha mi nem is kezeltük a hibát, láthatjuk, hogy a kivétel kezelve van. Hogy miért van ez így, erről a későbbiekben lesz szó.

19.1 Hibák kezelése hagyományos módon

A hiba kezelése hagyományos módon általában feltételek segítségével történik, melyekben valamilyen változók, hibakódok, függvények és eljárások visszaadási értékeit figyeljük. Ennek a módszernek a hátrányai egyszerűen megfogalmazva a következők:

- a hibakódokat meg kell jegyeznünk,
- minden függvény az eredménytelenséget másképp reprezentálja – false értéked ad vissza, 0-t, -1-et, stb,
- az eljárásokban a hiba valamelyik paraméterben van megadva, esetleg valamilyen globális paraméterben.

Nézzük meg hogyan kezelnénk hagyományos módszerekkel a hibát az előző programunkban. Pelda23

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 0;
  b := 0;
  if b<>0 then
    begin
     c := a div b;
     Button1.Caption := IntToStr(c);
    end
  else
    begin
     ShowMessage('Nullával nem lehet osztani!');
     Button1.Caption := 'Hiba';
    end;
end;
```


19.2 Hibák kezelése kivételek segítségével

Most ugyanebben az eljárásban a hibát kivételek segítségével fogjuk kezelni. Nem baj, ha még nem ismerjük a kivételek használatának pontos szintaxisát, ebben a példában csak azt mutatjuk be, hogyan fog az eljárásunk kinézni. Pelda24

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 0;
  b := 0;

  try
    c := a div b;
    Button1.Caption := IntToStr(c);

  except
    on EdivByZero do
      begin
       ShowMessage('Nullával nem lehet osztani!');
       Button1.Caption := 'Hiba';
      end;
  end;

end;
```

Ez program kiírja a hibát, majd a nyomógomb feliratában megjelenítti a „Hiba” szót. Ha a program elején a b változó értékét megváltoztatjuk nem nulla számra, akkor a program az osztás eredményét megjeleníti a nyomógomb feliratában.

Ez az egyszerű példa bemutatja a munkát a kivételekkel. Az kivételek egész mechanizmusa négy kulcsszón alapszik:

- **try** – a védett kód elejét jelzi, tehát azt a programrészt, amelyben előreláthatóan bekövetkezhet a hiba,
- **except** – a védett kód végét jelenti, a kivételek kezelésére szolgáló parancsokat tartalmazza a következő formában: `on` *kivétel_típpusa* `do` *parancsok* `else` *parancsok*
- **finally** – annak a programrésznek az elejét jelzi, amely minden esetben végrehajtódik, akár bekövetkezett a kivétel, akár nem. Ezt a részt általában a lefoglalt memória felszabadítására, megnyitott fájlok bezárására használjuk.
- **raise** – kivétel előhívására szolgál. Még ha úgy is tűnik, hogy a kivételt értelmetlen dolog kézzileg előhívni, mégis néha hasznos lehet.

Mielőtt konkrét példákon megmutatnánk a kivételek használatát, elmondunk néhány dolgot a kivételekről és a program állapotáról. Ha bekövetkezik valamilyen kivétel, akkor kerestetik egy „kezelő eljárás”, amely a kivételt kezeli. Ha az adott programrészben nincs ilyen eljárás, akkor a kivétel „feljebb vivődik” mindaddig, amíg valaki nem foglalkozik vele. Extrém esetekben ezt maga a Delphi kezeli, ezért van az, hogy végül minden kivétel kezelve lesz. Fontos, hogy a kivétel kezelése után a program a „kivételt kezelő eljárás” után fog folytatódni, nem a kivételt okozó programkód után.

Nézzük meg most részletesebben a **finally** részt. Ezt a részt olyan tevékenységek elvégzésére használjuk, amelyet el akarunk végezni minden esetben, akár a kivétel bekövetkezik, akár nem. Ilyen pl. a memória felszabadítása.

Nézzünk most példát a kivételek kezelésére a finally blokk nélkül (a memória felszabadítása helyett most az ablak feliratját fogjuk megváltoztatni „Szia”-ra).

```
procedure Form1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 0;
  b := 0;

  try
    c := a div b;
    Button1.Caption := IntToStr(c);
    Form1.Caption := 'Szia';

  except
    on EdivByZero do
      begin
       ShowMessage('Nullával nem lehet osztani!');
       Button1.Caption := 'Hiba';
      end;
  end;

end;
```

Ha bekövetkezik a 0-val való osztás, soha nem lesz végrehajtva az ablak feliratának beállítása „Szia”-ra. A megoldás a **finally** blokk használata lehet:

```
procedure Form1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 0;
  b := 0;
```

```
  try
    c := a div b;
    Button1.Caption := IntToStr(c);

  finally
    Form1.Caption := 'Szia';
  end;

end;
```

Most már biztosak lehetünk benne, hogy az alblak feliratának megváltoztatása (memóriatisztogatás) minden esetben megtörténik. Sajnos azonban most nincs lekezelve a kivételünk, ami végett az egészet tettük. A megoldás: kombináljuk (egymásba ágyazzuk) a **finally** és az **except** blokkot. Pelda25

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 0;
  b := 0;

  try
    try
      c := a div b;
      Button1.Caption := IntToStr(c);

    except
      on EdivByZero do
        begin
         ShowMessage('Nullával nem lehet osztani!');
         Button1.Caption := 'Hiba';
        end;
    end;

  finally
```



```
    Form1.Caption := 'Szia';
  end;

end;
```

19.3 Except blokk szintaxisa

Az **except** rész több felhasználási lehetőséget is ad:

```
try
  {parancsok}

except
  on {kivétel_típusa} do
    {ennek a kivételnek a kezelése}
  on {kivétel_típusa} do
    {ennek a kivételnek a kezelése}
  …
  else
    {bármilyen más kivétel kezelése}
end;
```

Láthatjuk, hogy az else részben bármilyen más kivételt kezelhetünk, melyet előre nem vártunk. Az ismeretlen kivételek kezelésénél azonban legyünk maximálisan óvatosak. Általában legjobb az ismeretlen kivételeket nem kezelni, így a Delphi-re hagyni. Az sem jó ötlet, ha a kivételt kezeljük pl. egy MessageBox-al, majd újból előhívjuk, mivel ebben az esetben a felhasználó kétszer lesz figyelmeztetve: egyszer a saját MessageBox-unkkal, egyszer pedig a Delhi MessageBox-ával. Tehát a kivételt vagy kezeljük, vagy figyelmen kívül hagyjuk, így a standard kezelése következik be.

Ha a kivételt kezeljük, lehetőségünk van például egy új kivétel meghívására megadott hibaszöveggel:

```
raise EConvertError.Create('Nem lehet konvertálni!');
```


20 Műveletek fájlokkal

A fájlok támogatását a Delphiben három pontba lehet szétosztani:

- az Object Pascal-ból eredő fájltámogatásra. Ennek az alap kulcsszava a File.
- a vizuális komponenskönyvtár fájltámogatása, amelyben metódusok segítségével lehet adatokat beolvasni ill. elmenteni (pl. LoadFromFile, SaveToFile metódusok)
- fájltámogatás adatbázis formátumokhoz. Ez csak a Delphi Professional változatától érhető el, ezzel nem fogunk foglalkozni ebben a fejezetben.

20.1 Fájltámogatás az Object Pascal-ban

A fájlokkal való munkát az Object Pascalban egy példa segítségével említjük meg. Hozzunk létre egy ablakot (form), melyen helyezzünk el egy Button és egy Memo komponenst. A gomb megnyomásakor az aktuális könyvtárban található DATA.TXT fájl tartalmát beolvassa a program a Memo komponensbe. Pelda26

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fajl: TextFile;
  sor: String;
begin
  AssignFile(fajl,'data.txt');
  Reset(fajl);
  while not Eof(fajl) do
```

```
    begin
     ReadLn(fajl,sor);
     Memo1.Lines.Add(sor);
    end;
  CloseFile(fajl);
end;
```

Lehet, hogy a Pascal-ból megszoktuk a Text (szöveg fájl típusa), Assign (fájl hozzárendelése), Close (fájl bezárása) parancsokat. Ezek a Delphi-ben **TextFile**, **AssignFile** és **CloseFile** parancsokkal vannak helyettesítve. Ennek az oka az, hogy a Delphi-ben az eredeti parancsok máshol vannak használva (pl. a Text több komponens tulajdonsága, pl. Edit, Memo). Az eredeti parancsszavak is a Delphiben továbbra is megmaradtak, de a System modullal lehet csak őket használni. Pl. az Assign(F) helyett a System.Assign(F) parancsot használhatjuk.

Ha felhasználjuk az előző fejezetben szerzett ismereteinket, magunk is rájöhetünk, hogyan tudjuk a fájl megnyitásánál, írásánál, olvasásánál kelezkező hibákat kezelni.

Ha más, nem szöveges fájlal szeretnénk dolgozni, hanem valamilyen típusos állománnyal, akkor használhatjuk a `file of <tipus>` formát a deklaráláshoz, pl. `file of Integer`.

Fájlokkal kapcsolatos leggyakrabban használt parancsok:

- **AssignFile(fájl, fizikainév)** – a fájl változóhoz egy fizikai fájl hozzákapcsolása a merevlemezen,
- **Reset(fájl) –** fájl megnyitása olvasásra,
- **Rewrite(fájl)** – fájl megnyitása írásra,
- **Read(fájl, változó)** – egy adat olvasása fájlból,
- **Write(fájl, változó)** – egy adat írása fájlba,

- **ReadLn(fájl, szöveg)** – sor olvasása szöveges (txt) fájlból,
- **WriteLn(fájl, szöveg)** – sor írása szöveges (txt) fájlba,
- **Seek(fájl, pozíció)** – a mutató beállítása a megadott helyre a típusos fájlban. A pozíció értéke 0-tól számolódik (0-első adat elé állítja be a mutatót, 1-második adat elé, 2-harmadi adat elé, stb.),
- **CloseFile(fájl) –** állomány bezárása.

20.2 Fájltámogatás a Delphi-ben

Sokszor nem akarunk foglalkozni a „hosszadalmas” Object Pascalból eredő fájltámogatással, hanem helyette egy rövid, egyszerű megoldást szeretnénk használni. Erre is van lehetőségünk a Delphiben. A legismertebb metódusok a **LoadFromFile** és a **SaveToFile**, melyek adatokat beolvasnak (megjelenítenek) ill. elmentenek a fájlba egyetlen parancssor beírásával. Ezek a metódusok elérhetők pl. a TString, TPicture, TBitmap osztályokban, ahogy további osztályokban is.

Változtassuk meg az előző példánkat a LoadFromFile metódus használatával. Pelda27

```
Procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('data.txt');
end;
```

Láthatjuk, hogy ez így mennyivel egyszerűbb. Nem kell deklarálnunk változókat, megnyitni, bezárni az állományt, hozzárendelni a külső fájl a változónkhoz.

20.3 Hibák a fájlokkal való munka során

A Delphi bármilyen I/O hiba esetében **EInOutError** kivételt generál. A hiba pontosabb definíciója az **ErrorCode** lokális változóban szerepel, melynek értéke a következők lehetnek:

ErrorCode	*Jelentése*
2	File not found
3	Invalid file name
4	Too many open files
5	Access denied
100	Disk read error – ha pl. a fájl végéről akarunk olvasni (eof)
101	Disk write error – ha pl. teli lemezre akarunk írni
102	File not assigned – ha pl. nem volt meghívva az Assign
103	File not open – ha pl. olyan fájlból akarunk dolgozni, amely nem volt megnyitva Reset, Rewrite, Append segítségével
104	File not open for input – ha olyan fájlból akarunk olvasni, amely írásra volt megnyitva
105	File not open for output – ha olyan fájlba akarunk írni, amely olvasásra volt megnyitva
106	Invalid numeric format – ha pl. nem számot karunk beolvasni szöveges fájlból szám típusú változóba

A kivételek standard kezelése természetesen képes a kivételt kezelni, ha azt nem tesszük meg a programunkban.

A következő példa bemutatja a kivételek kezelését a fájl beolvasásakor a Memo komponensbe. Pelda28

```
Procedure TForm1.Button1Click(Sender: TObject);
var
  fajl: TextFile;
  sor: String;
begin
  AssignFile(fajl,'data.txt');

  try
    Reset(fajl);

    try
     while not Eof(fajl) do
       begin
        ReadLn(fajl,sor);
        Memo1.Lines.Add(sor);
       end;

    finally
     CloseFile(fajl);
    end;

  except
    on E:EInOutError do
      case E.ErrorCode of
        2: ShowMessage('Nincs meg a fájl!');
      103: ShowMessage('A fájl nem volt megnyitva!');
      else
        ShowMessage('Hiba: ' + E.Message);
      end;
  end;
end;
```

Ebben a példában kezeltük a hibákat a fájl megnyitásánál és a fájlból való olvasáskor is.

Képzeljük el, hogy egy programban több helyen is dolgozunk az állományokkal. Leírni mindenhova ugyanazt a programrészt a kivételek kezelésére unalmas és hosszadalmas lehet. Szerencsére ez fölösleges is, mivel használhatjuk a **TApplication** objektum **OnException** eseményét, melybe beírjuk a kivételeket kezelő programrészt „egyszer s mindenkorra". Ezt egy egyszerű példán szemléltetjük, melyben bármilyen nem kezelt kivétel esetében a program leáll. Pelda29

```
Procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := AppException;
end;

Procedure TForm1.AppException(Sender: TObject;
                              E: Exception);
begin
  Application.ShowException(E);    // hibaüzenet
  Application.Terminate;           // programleállás
end;
```

20.4 További fájlokkal kapcsolatos parancsok

Csak röviden megemlítjük a Delphi további metódusait, melyekkel a fájlok és a mappák összefüggnek:

- **FileExist(**név**)** – értéke true, ha a megadott nevű állomány létezik.
- **DeleteFile(**név**)** – kitörli a megadott nevű állományt és true értéket ad vissza, ha a törlés sikeres volt.

- **RenameFile(**réginév**,** újnév**) –** átnevezi a fájlt és true értéket ad vissza, ha az átnevezés sikeres volt.
- **ChangeFileExt(**név**,** kiterjesztés**) –** megváltoztatja a fájl kiterjesztését és visszaadja az fájl új nevét.
- **ExtractFileName(**teljes_név**)** – A teljes útvonallal együtti fájlnévből kiszedi csak a fájl nevét, melyet visszaad.
- **ExtractFileExt(**név**)** – Az adott fájl kiterjesztését adja vissza.

További fájlokkal és mappákkal kapcsolatos függvények találhatók a **SysUtils** modulban.

21 Standard dialógusablakok

Mielőtt belekezdenénk a standard dialógusablakok ismertetésébe, nézzünk meg még néhány komponenst, melyek az állományokkal, mappákkal való munkát segítik. Ezek a komponensek nem dolgoznak közvetlenül a fájlokkal, könyvtárokkal, hanem csak a neveiket jelenítik meg, választási lehetőséget adnak, stb.

Az alábbi komponensek igaz, egy kicsit régebbiek, de lehet őket jól használni. Ezek a Win 3.1 kategória alatt találhatók:

- **FileListBox** – ez egy speciális ListBox az aktuális könyvtárban található összes fájl kírására.
- **DirectoryListBox** – szintén egy speciális ListBox, amely az adott meghajtón levő könyvtárszerkezet megjelenítésére szolgál.
- **DriveComboBox** – egy speciális legördülő lista, amely számítógépről előrhető meghajtók listáját tartalmazza.
- **FilterComboBox** – a fájlok maszkjainak megjelenítésére szolgáló legördölő lista.

Sok alkalmazásnak hasonló igénye van: fájlokat megnyitnak meg, mentenek el, keresnek valamilyen kifejezést, nyomtatnak, színt (betű, háttér) választanak, stb. Ezért léteznek úgynevezett „common dialog”-usok, tehét előre definiált **standard dialógusablakok**.

Előnyei:

- ugyanaz az ablak jelenik meg a felhasználónak minden alkalmazásnál (pl. mentésnél, megnyitásnál, stb.), igy könnyen tudja kezelni,

- a programozónak is egyszerűbb ezeket használnia, mint sajátot készíteni.

Hátrányai:

- egy bizonyos határokon túl nem lehet őket változtatni, úgy kell használnunk őket, ahogy kinéznek, még ha a dialógusablak néhány funkcióját nem is használjuk vagy hiányzik valamilyen funkció,
- ezek standard Windows dialógusablakok, ezért ha pl. magyar nyelvű programunkat angol Windows alatt futtatjuk, keveredik a két nyelv – a programunk magyar marad, de a dialógusablakok angolul lesznek.

A Delphi-ben a következő dialógusablakokkal találkozhatunk:

Dialógusablak neve	*Modális?*	*Jelentése*
OpenDialog	igen	Az állomány kiválasztása megnyitáshoz.
SaveDialog	igen	Az állomány megadása mentéshez.
OpenPictureDialog	igen	Az állomány kiválasztása megnyitáshoz, tartalmazza a kiválasztott kép előnézetét is.
SavePictureDialog	igen	Az állomány megadása mentéshez, tartalmazza a kép előnézetét is.

FontDialog	igen	A betűtípus kiválasztására szolgáló dialógusablak.
ColorDialog	igen	Szín kiválasztására szolgáló dialógusablak.
PrintDialog	igen	A nyomtatandó dokumentum nyomtatóra való küldéséhez szolgáló dialógusablak.
PrinterSetupDialog	igen	A nyomtató (nyomtatás) beállítására szolgáló dialógusablak.
FindDialog	nem	Szövegben egy kifejezés keresésére szolgáló dialógusablak.
ReplaceDialog	nem	Szövegben egy kifejezés keresésére és kicserélésére szolgáló dialógusablak.

Megjegyzés: A modális ablak olyan ablak, amelyből nem lehet átkapcsolni az alkalmazás másik ablakába, csak a modális ablak bezárása után.

A dialógusablakok tervezési időben egy kis négyzettel vannak szemléltetve, amely futási időben nem látszódik. A dialógusablakot az **Execute** metódussal lehet megnyitni. Ez a metódus true értéket ad vissza, ha a felhasználó az OK gombbal zárta be, false értéket pedig ha a felhazsnáló a dialógusablakból a Cancel gombbal vagy a jobb felső sarokban levő X-szel lépett ki. A PrinterSetupDialog kivételével

mindegyik dialógusnak van **Options** tulajdonsága, amely több true/false altulajdonságból áll.

21.1 OpenDialog, SaveDialog

Ez a két dialógusablak annyira hasonlít egymásra, hogy egyszerre vesszük át őket.

Tulajdonságok:

- **DefaultExt** – kiterjesztés, amely automatikusan hozzá lesz téve a fájl nevéhez, ha a felhasználó nem ad meg.
- **FileName** – a kiválasztott állomány teljes nevét (útvonallal együtt) tartalmazza. Lehet megadni is mint bemeneti érték.
- **Files** – read-only, run-time tulajdonság, amely a kiválasztott állomány (állományok) teljes nevét (neveit) tartalmazza útvonallal együtt.
- **Filter** – az állományok szűrése oldható meg a segítségével (megadott maszk alapján, pl. *.txt, *.doc, stb.). A program tervezési fázisában ez a tulajdonság a Filter Editor segítségével adható meg.
- **FilterIndex** – melyik legyen a „default” maszk a dialógusablak megnyitásakor.
- **InitialDir** – kezdeti könyvtár (mappa).
- **Options** – beállítási lehetőségek (logikai értékek), melyekkel a dialógusablak kölalakját lehet módosítani:

- **ofOverwritePrompt** – megjelenik a figyelmeztetés, ha a felhasználó létező fájlt akar felülírni.
- **ofHideReadOnly** – nem jelenik meg a „megnyitás csak olvasásra” lehetőség a dialógusablakon.
- **ofShowHelp** – megjelenik a „Help” nyomógomb. Ha nincs súgónk hozzá, akkor ezt ajánlott letiltani.
- **ofAllowMultiSelect** – lehetőséget ad több állomány kiválasztására egyszerre. A kiválasztott fájlokat a TString típusú **Files** tulajdonságban kapjuk vissza.
- **ofEnableSizing** – lehetőséget ad a felhasználónak változattni a dialógusablak méretét.
- **ofOldStyleDialog** – „régi stílusú” dialógusablakot jelenít meg.

- **Title** – a dialógusablak felirata (Caption helyett).

Események:

A legfontosabb események az **OnShow** és az **OnClose**, melyek a dialógusablak megnyitásakor ill. bezárásakor következnek be. Hasznos esemény lehet még az **OnCanClose**, melyben a dialógusablak bezárását lehet megakadályozni. Továbbá használhatjuk még az **OnFolderChange**, **OnSelectionChange**, **OnTypeChange** eseményeket is, melyek akkor következnek be, ha a felhasználó megváltoztatja a mappát, fájlok kijelölését ill. a fájlok maszkját (szűrő).

Metódusok:

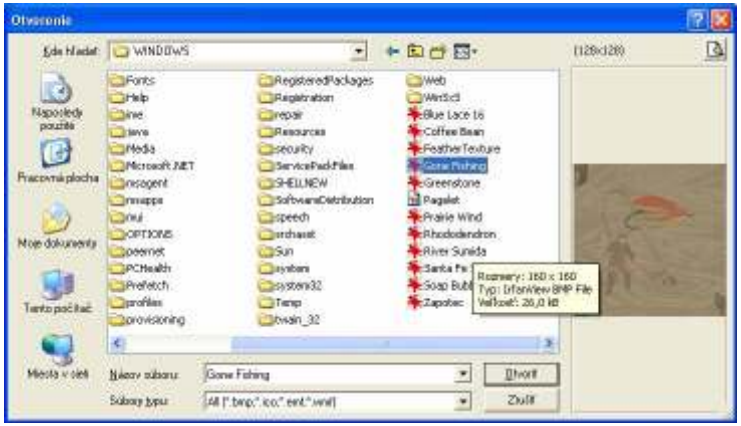
Legfontosabb metódus az **Execute**, mellyel a dialógusablakot megjelentetjük.

Példa: a kiválasztott állomány nevét a Form1 feliratába szeretnénk kiírni, majd megállapítani, hogy az állomány csak olvasásra lett-e megnyitva.

```
…
if OpenDialog1.Execute then
  begin
   Form1.Caption := OpenDialog1.Filename;
   if ofReadOnly in OpenDialog1.Options then
     ShowMessage('Csak olvasásra megnyitott.');
  end;
…
```

21.2 OpenPictureDialog, SavePictureDialog

Hasonló az előzőkhöz, de a dialógusablak része a kép előnézetét mutató felület is. Ezt az előnézetet azonban csak akkor láthatjuk, ha a képet felismeri a TPicture osztály, tehát ha a kép .bmp, .ico, .wmf, .emf típusú.



21.3 FontDialog

Ez a dialógusablak biztosan mindenki számára ismerős a szövegszerkesztő programokból.

Tulajdonságai:

- **Device** – meghatározza melyik berendezés számára van a betűtípus (fdScreen, fdPrinter, fdBoth)

- **Font –** bemeneti és kimeneti információk a betűtípusról (bemeneti lehet pl. az aktuális betűtípus).
- **MinFontSize, MaxFontSize** – meg lehet segítségükkel határozni milyen értékek között választhat a felhasználó betűméretet. Szükséges hozzá még engedélyezni a **fdLimitSize**-ot az **Options** tulajdonságban. Ha értéknek a 0-t hagyjuk, akkor a választás nem lesz korlátozva.
- **Options** – különféle beállítási lehetőségek:
    - **fdAnsiOnly –** csak szöveges betűtípusokat jelenít meg, tehát kiszűri pl. a Symbols, Wingdings, stb. betűtípusokat.
    - **fdApplyButton –** megjeleníti az „Alkalmaz” nyomógombot.
    - **fdEffects** – a betűstílusok beállításának lehetőségét jeleníti meg a dialógusablakban (pl. aláhúzott, stb.)
    - **fdTrueTypeOnly** – csak a True-Type betűtípusokat jeleníti meg.
    - **fdForceFontExist –** ajánlott értékét true-ra állítani, különben a felhasználó megadhat olyan nevű betűtípust is, amely nem létezik.

Események:

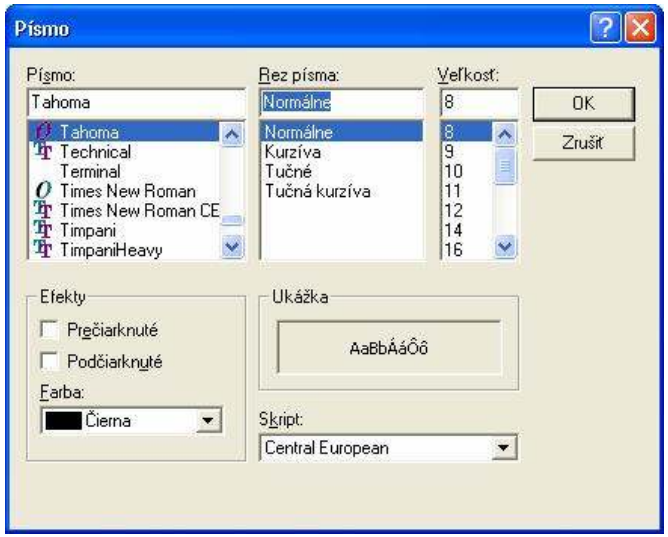
Az előző dialógusokhoz képest van egy új eseménye, az **OnApply**, amely akkor következik be, ha a felhasználó megnyomja az „Alkalmaz” nyomógombot.

Metódusok:

Legfontosabb metódusa az **Execute**.

Nézzünk egy konkrét példát a FontDialog használatára. A példánkban a Memo komponens betűtípusát szeretnénk beállítani.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;
```



21.4 ColorDialog

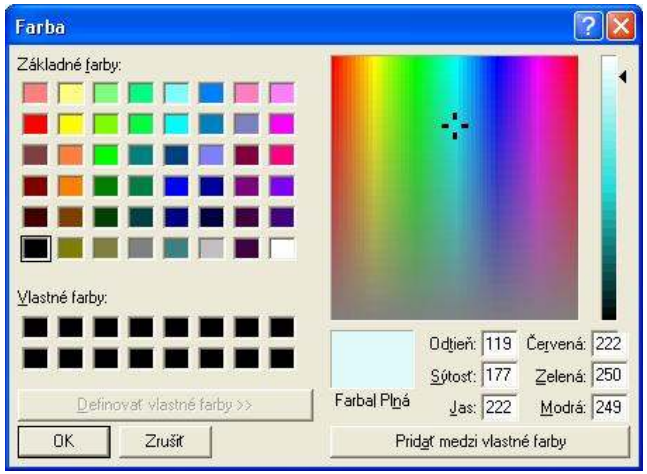
Legfontosabb tulajdonsága a **Color**, melyben megadható és melyből kiolvasható a konkrét szín. A **CustomColor** tulajdonság segítségével definiálhatunk 16 felhasználói színt. A definiáláshoz klasszikus String List Editor-t használhatunk, melyben a formátum a következő:

```
ColorX=AABBCC,
```

ahol X helyére A..P betűket írhatunk, az AA, BB, CC pedig a szín egyes összetevőit jelölik hexadecimális számrendszerbe.

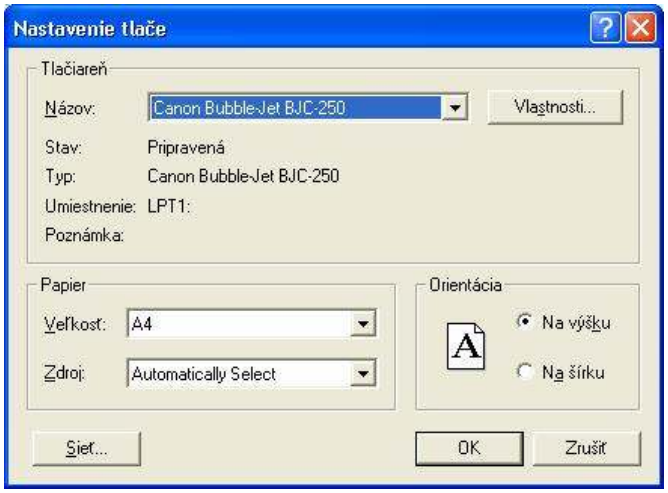
Az **Options** tulajdonság altulajdonságai:

- **cdFullOpen** – az egész dialógusablak megnyitását eredményezi (tehát a felhasználói színeket is).
- **cdPreventFullOpen** – megakadájozza (tiltja) a felhasználói színek részének megnyitását a dialógusablakban.

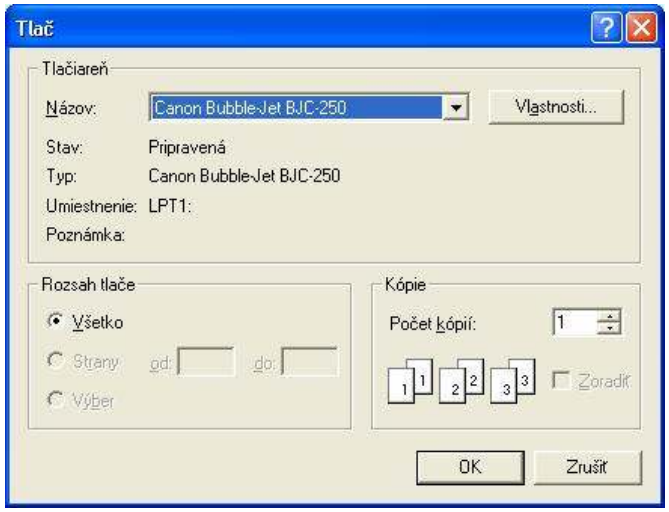


21.5 PrinterSetupDialog, PrintDialog

A **PrinterSetupDialog** a nyomtató beállításait megjelenítő dialógusablakot nyit meg. Ennek nincs semmi különösebb eseménye vagy tulajdonsága. Amit ezzel a dialógusablakkal kapcsolatban szükséges megtennünk, az csak annyi, hogy megnyitjuk az **Execute** metódussal. A dialógusablak formája szorosan összefügg a beinstallált nyomtató típusával.



A **PrintDialog** komponensnek már van néhány paramétere. Be lehet állítani kezdeti értékeket vagy ki lehet olvasni beállított értékeket, melyek megadhatják például: a másolatok számát (**Copies** tulajdonság), a leválogatás módját (**Collage** tulajdonság). Szintén be lehet határolni (korlátozni) a kinyomtatandó oldalak számát (**MinPage**, **MaxPage**).



21.6 FindDialog, ReplaceDialog

A **FineDialog** és **ReplaceDialog** ablakoknál szintén csak a dialógusablakokról van szó, amely kizárólag az adatok bevitelére szolgál, magát a keresést, cserét sajnos nekünk kell teljes mértékben beprogramoznunk.

A szöveget, amelyet keresnünk kell a **FindText** tulajdonságban kapjuk meg. A ReplaceDialog-nak van még egy **ReplaceText** tulajdonsága is.

Az **Options** tulajdonság lehetőségei:

- **frDown** – keresés iránya, true értéke azt jelenti, hogy az alapértelmezett = lefelé.
- **frMatchCase** – meg legyenek-e különböztetve a kis és nagybetűk.

- **frWholeWord** – csak egész szavak legyenek keresve.

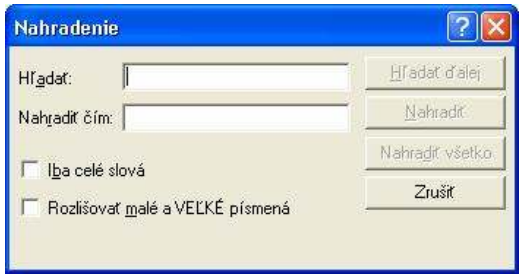
Ha ezek valamelyikét egyáltalán nem szeretnénk a felhasználónak megjeleníteni a dialógusablakban, használjuk a Hide-al kezdődő lehetőségeket (pl. **frHideMatchCase**, stb.). Van lehetőség arra is, hogy ezek a lehetőségek a felhasználónak megjelenjenek, de le legyenek tiltva. Ehhez a Disable szóval kezdődő lehetőségeket használhatjuk (pl. **DisableMatchCase**).

Események:

A FindDialog **OnFind** eseménnyel rendelkezik, amely akkor következik be, ha a felhasználó a „Find Next” nyomógombra kattintott. A ReplaceDialog még rendelkezik egy **OnReplace** eseménnyel is, amely a „Replace” nyomógomb megnyomásakor következik be.

Metódusok:

Ezek a dialógusablakok nem modálisak, tehát a képernyőn maradhatnak többszöri keresés / csere után is. Az **Execute** metóduson kívül rendelkezésre áll még az **CloseDialog** metódus is, amely bezárja a dialógusablakot.



22 Több ablak (form) használata

Az alkalmazások készítésénél egy idő után eljön az a pillanat, amikor már nem elég egy ablak az alkalmazás elkészítéséhez. Szükségünk lehet további ablakokra, amelyek segítségével például valamilyen adatokat viszünk be a programba, beállításokat állítunk be, stb.

Tehát az alkalmazásnak lesz egy fő ablaka (ez jelenik meg rögtön a program indításakor) és lehetnek további ablakai (pl. bevitelre, beállítások megadására, stb.). Ezek az ablakok két féle módon jeleníthetők meg:

- modális ablakként: az így megjelenített ablakból nem tudunk átlépni az alkalmazás másik ablakába.
- nem modális ablakként: az ilyen ablakból át lehet lépni az alkalmazás másik ablakába.

22.1 Alkalmazás két ablakkal (modális ablak)

Készítsünk egy programot! Az alkalmazás két ablakot fog tartalmazni: egy fő ablakot és egy segédablakot, amely segítségével adatot viszünk be a programba. A segédablak modálisan lesz megjelenítve, tehát nem lehet majd belőle átkapcsolni az alakalmazás másik (fő) ablakába. Pelda30

Alkalmazásunkban a következő komponenseket fogjuk használni: Label, 2 x Button (Form1-en) és Label, Edit, 2 x Button (Form2-n).

Az alkalmazás létrehozása után (**File – New – VCL Form Applications - Delphi for Win32**) az alkalmazásunknak egy ablaka (Form1) van. Mivel a mi alkalmazásunk két ablakot fog tartalmazni, a második ablakot külön be kell raknunk az alkalmazásba. Ehhez válasszuk a **File – New – Form - Delphi for Win32** menüpontot. Az alkalmazásba bekerül a Form2 ablak és a hozzá tartozó program modul is – Unit2.pas. Ahhoz, hogy az egyik modulból (unitból) tudjuk használni a másikat (tehát hogy az egyik modulból elérhetők legyenek a másikban levő komponensek), egy kicsit változtatnunk kell a forráskódokon (Unit1.pas, Unit2.pas) a következő képpen:

1. Az első modul (Unit1.pas) **uses** részét egészítsük ki a Unit2-vel:

```
uses
   Windows, Messages, SysUtils, Variants,
   Classes, Graphics, Controls, Forms,
   Dialogs, Unit2;
```

2. A második modulban keressük meg az **implementation** részt és ez alá írjuk be:

```
uses Unit1;
```

Ezzel bebiztosítottuk, hogy mindkét modulban fogunk tudni dolgozni a másik modulban definiált komponensekkel.

A második ablakot a Delphi automatikusan létrehozza a program indításakor. Mi azonban most megmutatjuk, hogyan tudjuk kikapcsolni az ablak automatikus létrehozását, majd hogyan tudjuk a program futása alatt mi létrehozni a második ablakot (Form2).

Ehhez tehát előbb kapcsoljuk ki a Form2 automatikus létrehozását a program indításakor a következő képpen: nyissuk meg a **Project – Options** menüpontot. Ha nincs kiválasztva, válasszuk ki itt a

Forms részt. Majd a Form2-t az **Auto-create forms** listából helyezzük át az **Available forms** listába. A form létrehozását a programban így akkor hajthatjuk végre, amikor szükségünk lesz rá. Ezt a nyomógomb megnyomásakor fogjuk megtenni:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Form2: TForm2;
begin
  Form2 := TForm2.Create(Self);
  try
    if Form2.ShowModal = mrOk then
      Label1.Caption := Form2.Edit1.Text;
  finally
    Form2.Free;
  end;
end;
```

Észrevehetjük, hogy az alkalmazás futása alatt ha megnyitjuk a második form-ot, az alkalmazásból át lehet kapcsolni másik

alkalmazásba, de ugyanennek az alkalmazásnak a főablakába nem tudunk visszamenni, amíg a modális ablakot nem zárjuk be.

Nézzük meg részlezesen először a Unit1.pas forráskódját:

```
unit Unit1;

…

procedure TForm1.Button1Click(Sender: TObject);
var
  Form2: TForm2;
begin
  Form2 := TForm2.Create(Self);
  try
    if Form2.ShowModal = mrOk then
      Label1.Caption := Form2.Edit1.Text;
  finally
    Form2.Free;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

A Unit2.pas forráskódja pedig így néz ki:

```
unit Unit2;

…
```



```
procedure TForm2.Button1Click(Sender: TObject);
begin
  ModalResult := mrOk;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  ModalResult := mrCancel;
end;

end.
```

A megnyitott modális ablakot a **ModalResult** tulajdonság beállításával zárjuk be. A beállított értéket megkapjuk a Form2.ShowModal függvény hívásánál visszatérési értékként.

22.2 Ablakok, melyekből át lehet kapcsolni másik ablakokba (nem modális ablak)

Készítsünk el az előző alkalmazáshoz hasonló alkalmazást annyi különbséggel, hogy a segédablak nem modálisan lesz megjelenítve, tehét a felhasználó átkapcsolhat a főablakba a segédablak bezárása nélkül. Pelda31

Az eltérés az előző példától a modális és nem modális ablakok közötti különbségekből adódik. Míg a modális ablakot csak az ablak bezárásával hagyhatjuk el, a nem modális ablaknál ez nem igaz. Az ablakot ezért nem hozhatjuk létre ugyanúgy mint az előző példában: elsősorban azért nem, mert a felhasználó belőle átléphet az alkalmazás fő ablakába és megpróbálhatja újra létrehozni a segédablakot annak ellenére, hogy az már létezik. Ezért mielőtt létrehoznánk a segédablakot tesztelnünk kell, hogy az már létezik-e, és ha igen, akkor csak aktiválnunk kell. A másik „problémánk“ az, hogy a **ShowModal** metódus

meghívásánál a program „várakozik“ addig, amíg a modális ablakot nem zárjuk be, míg a nem modális ablak létrehozásánál, a **Show** metódus meghívásánál a program fut tovább.

Ahhoz, hogy az ablak létezését bármikor tesztelhessük szükségünk lesz egy globális változóra (Form2). Ezt tehát a program elején kell deklarálnunk, nem valamelyik eljárásban.

Most megváltoztatjuk a második ablak megnyitására szolgáló eljárást:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Form2 = nil then
    Form2 := TForm2.Create(Self);
  Form2.Show;
end;
```

Megoldásra maradt még két problémánk: hogyan zárjuk be a második ablakot és hogyan biztosítsuk be a második ablakban megadott érték átadását. Mindkét tevékenységet a második ablakban (Form2) fogjuk végrehajtani. Ne felejsük el, hogy a **Close** metódus a második ablaknál (Form2) csak az ablak elrejtését eredményezi, nem a felszabadítását a memóriából. A Form2 továbbra is a memóriában van és bármikor előhívhatjuk. Ha azt akarjuk, hogy a Form2 a bezárása után a memóriában ne foglalja a helyet, fel kell szabadítanunk.

Nézzük most meg a két forráskódot, először a Unit1-et:

```
unit Unit1;
```



```
…

var
  Form1: TForm1;
  Form2: TForm2;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Form2 = nil then
    Form2 := TForm2.Create(Self);
  Form2.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

Majd nézzük meg a Unit2 modult is:

```
unit Unit2;

…

procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.Label1.Caption := Edit1.Text;
  Close;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  Close;
end;
```

```
procedure TForm2.FormClose(Sender: TObject;
                  var Action: TCloseAction);
begin
  Action := caFree;
end;

procedure TForm2.FormDestroy(Sender: TObject);
begin
  Form2 := nil;
end;

end.
```

A második form nyomógombjainak **OnClick** eseményeibe a **Close** metódust használtuk, mellyel bezárjuk az ablakot.

Az ablak bezárásánál bekövetkezik az **OnClose** esemény. Ennek az eseménynek a kezelésében beállítottuk az **Action** paramétert **caFree**-re, ami az ablak felszabadítását eredményezi a memóriából.

A felszabadításkor bekövetkezik az **OnDestroy** esemény, melynek kezelésében beállítottuk a Form2 mutató értékét nil-re.

A Form2 értékét az első unitban teszteljük, amikor megpróbáljuk létrehozni a második ablakot.

22.3 Könyvnyilvántartó program

Az alábbi példán keresztül szemléltetjük a két ablakkal és a fájlokkal való munkát. A következő egyszerű program könyvek nyilvántartására fog szolgálni. Minden könyvről meg szeretnénk jegyezni a szerzőt, könyv címét, oldalainak számát és a könyv árát. Az alkalmazásunk tartalmazzon négy nyomógombot: előző könyv

adatainak megtekintésére, következő könyv adatainak megtekintésére, új könyv hozzáadására és könyv törlésére. Az új könyv hozzáadása egy másik (modálisan megjelenített) form segítségével történjen. A program kilépéskor egy külső fájlba mentse az adatokat, melyeket az alkalmazás indításakor automatikusan olvasson be. Pelda32

Az új könyv hozzáadása a következő form segítségével fog történni:

Mivel minden egyes könyvről meg kell jegyeznünk több adatot is (szerző, cím, oldalak száma, ára), ezért definiálunk egy record típust, ennek a programban a **TKonyv** nevet adjuk. A könyveket egy ilyen típusú tömbben tároljuk és egy ilyen típusú állományba mentjük el, ill. olvassuk be.

A második formot most nem fogjuk mi létrehozni, hanem hagyjuk, hogy a Delphi automatikusan létrehozza a program indításakor. Tehát most nem fogjuk kikapcsolni a második form létrehozását a projekt beállításaiban.

Nézzük először is az első ablakhoz tartozó programkódot:

…

uses Windows, Messages, … , **Unit2**;

…

```
implementation
```



```
{$R *.dfm}

type
  { TKonyv tipus definialasa }
  TKonyv = record
             Szerzo, Cim: string[255];
             Oldalak, Ar: integer;
           end;

var
  { tomb, file deklaralasa }
  a: array[1..10000] of TKonyv;
  f: file of TKonyv;
  { n - konyvek szama a tombben }
  n: integer;
  { akt - aktualis, megjelenitett konyv }
  akt: integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
 n:=0;
 { adatok beolvasasa kulso fajlbol }
 AssignFile(f,'konyvek.dat');
 try
   Reset(f);
   try
     while not eof(f) do
       begin
         inc(n);
         Read(f,a[n]);
       end;
   finally
     CloseFile(f);
   end;
 except
   MessageDlg('Hiba az adatok megnyitasakor.'
              + chr(10) + chr(13) +
              'A file nem letezik?',
              mtWarning,[mbOK],0);
 end;
 { elso konyv megjelenitese, ha letezik }
 if n>0 then begin
             akt := 1;
             Label5.Caption := a[1].Szerzo;
             Label6.Caption := a[1].Cim;
```

```
            Label7.Caption :=
                          IntToStr(a[1].Oldalak);
            Label8.Caption := IntToStr(a[1].Ar);
           end
       else begin
            akt := 0;
            Label5.Caption := '';
            Label6.Caption := '';
            Label7.Caption := '';
            Label8.Caption := '';
           end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  { ugras az elozore }
  if akt>1 then
    begin
     dec(akt);
     Label5.Caption := a[akt].Szerzo;
     Label6.Caption := a[akt].Cim;
     Label7.Caption := IntToStr(a[akt].Oldalak);
     Label8.Caption := IntToStr(a[akt].Ar);
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  { ugras a kovetkezore }
  if akt<n then
    begin
     inc(akt);
     Label5.Caption := a[akt].Szerzo;
     Label6.Caption := a[akt].Cim;
     Label7.Caption := IntToStr(a[akt].Oldalak);
     Label8.Caption := IntToStr(a[akt].Ar);
    end;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  i: integer;
begin
  { uj konyv hozzaadasa Form2 megjelenitesevel }
  if Form2.ShowModal = mrOk then
```



```
    begin
    { a konyv helyenek megkeresese ugy, hogy
      a konyvek cimei ABC szerint legyenek rendezve }
    i := n;
    while (i>0) and (a[i].Cim>Form2.Edit2.Text) do
      begin
       a[i+1] := a[i];
       dec(i);
      end;
    a[i+1].Szerzo := Form2.Edit1.Text;
    a[i+1].Cim := Form2.Edit2.Text;
    a[i+1].Oldalak := StrToInt(Form2.Edit3.Text);
    a[i+1].Ar := StrToInt(Form2.Edit4.Text);
    inc(n);
    { a beirt konyv megjelenitese }
    akt := i;
    Button2.Click;
    end;
end;

procedure   TForm1.FormClose(Sender:   TObject;   var
Action: TCloseAction);
var
  i: integer;
begin
 { adatok mentese fajlba }
 try
   Rewrite(f);
   try
     for i:=1 to n do Write(f,a[i]);
   finally
     CloseFile(f);
   end;
 except
   MessageDlg('Hiba az adatok mentésénél!',
               mtError,[mbOK],0);
 end;
end;

procedure TForm1.Button4Click(Sender: TObject);
var
  i: integer;
begin
  { a torolt konyv utani konyvek
    eggyel elobbre helyezese }
```

```
  for i := akt to n-1 do a[i] := a[i+1];
  dec(n);
  { kovetkezo, vagy elozo konyv
    megjelenitese, ha van ilyen }
  if akt<=n then begin
                 dec(akt);
                 Button2.Click;
                 end
            else if n>0 then begin
                             akt := n-1;
                             Button2.Click;
                             end
                        else begin
                             akt := 0;
                             Label5.Caption := '';
                             Label6.Caption := '';
                             Label7.Caption := '';
                             Label8.Caption := '';
                             end;
end;

end.
```

Majd a második ablakhoz tartozó forráskódot:

```
…

implementation

{$R *.dfm}

uses Unit1;

procedure TForm2.FormShow(Sender: TObject);
begin
  { kezdeti beallitasok }
  Top := Form1.Top + 30;
  Left := Form1.Left + 30;
  Edit1.Text := '';
  Edit2.Text := '';
  Edit3.Text := '';
```



```
  Edit4.Text := '';
  Edit1.SetFocus;
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
  ModalResult := mrOk;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  ModalResult := mrCancel;
end;

end.
```

A második ablaknál a Form **OnShow** eseményében állítjuk be a kezdeti beállításokat. Ez azért van, mert az OnCreate esemény csak a program indításánál következik be, ugyanis itt hozza létre mindkét form-ot (mivel hagytuk az „Auto-create forms” beállításban a Form2-t is). Így a modális ablak megnyitásakor, majd bezárásakor a Form2 továbbra is létezik, csak nem látható a képernyőn. Ezért szükséges például az Edit komponensekbe beírt szöveg törlése az ablak megjelenítésekor (OnShow eseményben).

23 SDI, MDI alkalmazások

A programozás során találkozhatunk az MDI, SDI rövidítésekkel. Ezek jelentése:

- **SDI – Single Document Interface**: olyan alkalmazás, melyben egyszerre csak egy dokumentumot (objektumot) lehet beolvasni. Ilyen például a Windowsban található Jegyzettömb.
- **MDI – Multiple Document Interface**: olyan alkalmazás, amely egyszerre több dokumentum (objektum) megnyitását teszi lehetővé. Ilyen például az MS Word.

23.1 Alkalmazás, mely több dokumentummal tud egyszerre dolgozni (MDI)

Biztos mindenki ismer olyan alkalmazást, melyben egyszerre több dokumentum (kép, fájl, táblázat, ...) lehet megnyitva. Erre tipikus példa az MS Word. Ebben a fejezetben megmutatjuk, hogyan hozhatunk létre egy ilyen alkalmazást. Egy képnézegető programot fogunk létrehozni, melyben egyszerre több kép lehet nyitva.

Az alkalmazásban két form-ot fogunk használni, egy Button komponenst (Form1-en), egy OpenPictureDialog komponenst (Form1-en) és egy Image komponenst (Form2-n).

MDI alkalmazást legegyszerűbben létrehozhatunk a **File – New – Others** menüpontból a **Delphi Projects – MDI Application** segítségével. Az **OK**-ra kattintás után meg kell adnunk a egy mappát, ahová a projektünket menteni akarjuk. Végül létrejön az MDI

alkalmazásunk váza menüvel, dialógusokkal, stb. együtt. Ezzel a módszerrel pár perc alatt létrehozhatunk MDI alkalmazást.

Ahhoz, hogy megértsük és gyakoroljuk az ilyen alkalmazás létrehozását, most kialakítunk kézileg egy MDI alkalmazást.

Hozzunk létre egy hagyományos alkalmazást (**File – New – VCL Forms Application - Delphi for Win32**). A form-ra helyezzünk el egy Button és egy OpenPictureDialog komponenst. Pelda33

Most létre kell hoznunk egy gyermek (child) form-ot. Ehhez rakjunk be az alkalmazásunkba egy új form-ot (**File – New – Form - Delphi for Win32**). Erre helyezzünk el egy Image komponenst. Továbbá be kell biztosítanunk a Form2 felszabadítását a memóriából a bezáráskor (lásd a program OnClose eseményének kezelését).

Ne felejtsük el a Unit1 modul **Uses** részét kiegészíteni a Unit2 modullal.

Most szüntessük meg a Form2 automatikus létrehozását az alkalmazás indulásakor (**Project – Options – ...**), ahogy azt az előző fejezetekben is már megtettük. Ha ezt nem tennénk meg, az induláskor egy nem szép, üres ablak lenne látható.

Nézzük hogy néz ki a Unit1.pas:

```
unit Unit1;

interface

uses
  …, Unit2;

…

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  if OpenPictureDialog1.Execute then
    with TForm2.Create(Self) do
      begin
      Caption := OpenPictureDialog1.FileName;
      Image1.Picture.LoadFromFile(
                        OpenPictureDialog1.FileName);
      end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  FormStyle := fsMDIForm;
end;

end.
```

Majd nézzük meg a Unit2.pas-t is:

```
unit Unit2;

…

procedure TForm2.FormCreate(Sender: TObject);
begin
  FormStyle := fsMDIChild;
  Image1.Align := alClient;
end;

procedure TForm2.FormClose(Sender: TObject;
                        var Action: TCloseAction);
begin
  Action := caFree;
end;

end.
```


Ha megpróbáljuk az alkalmazásunkat futtatni, észrevehetjük, hogy egy kicsit másképp működik, mint az eddig létrehozott alkalmazások. A második ablakot nem lehet a szülő ablakán kívülre mozgatni, minimalizálásnál csak a szülő ablak aljára teszi le. Egyszerre több képet is megnyithatunk. Figyeljük meg a szülő ablak feliratát, ha valamelyik utód ablakot maximalizáljuk.

A második ablak (utód) maximalizálásával összefügg egy probléma: a maximalizálás után elfoglalja a szülő ablak egész területét és utána már nincs rá módunk megváltoztatni az ablak méretét. Ezt egy kis trükkel fogjuk kijavítani: helyezzünk el a Form1 ablakon egy **MainMenu** komponenst. Semmi mást nem kell beállítanunk, menüt sem kell létrehoznunk. A MainMenu komponens elhelyezése után már ha kinagyítjuk az utód ablakát, a szülő ablak második sorában megjelennek az utódhoz tartozó rendszergombok (minimalizálás, maximalizálás, bezárás).

Az alkalmazás létrehozásának egyik legfontosabb része a **FormStyle** tulajdonság beállítása. A fő ablaknál beállítjuk: **FormStyle := fsMDIForm;** és a gyermek ablaknál beállítjuk: **FormStyle := fsMDIChild;**.

A **FormStyle** további lehetséges értékei: **fsNormal** (klasszikus SDI alkalmazás, amelyet eddig is használtunk), **fsStayOnTop** (olyan ablakokra, melyeknél azt szeretnénk, hogy mindig a többi ablak előtt legyen).

Figyeljük meg, hogyan alakítunk ki gyermek ablakokat. Az előző fejezetben a **változó := TForm2.Create(Self)** felírást használtuk, míg itt megelégedtünk a **with TForm2.Create(Self) do** felírással. Ennek oka, hogy itt az ablak kialakítása után már nincs szükségünk az ablakra

való hivatkozásra, így a **Create** visszaadási értékét nem kell semmilyen változóban tárolnunk.

A kép megjelenítésére az Image komponenst használtuk. A kép beolvasására és megjelenítésére használhattuk ennek a komponensnek a **LoadFromFile** metódusát.

24 A Windows vágólapja

Biztos mindenki ismeri a Ctrl+C, Ctrl+V, illetve a Ctrl+X billentyűzetkombinációkat. Az első a kijelölt szöveg (vagy más objektum) másolására szolgál a vágólapra, a második a vágólap tartalmának beillesztésére a kijelölt helyre. A Ctrl+X rövidítés hasonlóan működik mint a Ctrl+C annyi különbséggel, hogy a kijelölt részt kivágja a dokumentumból. Hasonló műveletek elérhetők az alkalmazás menüjén keresztül is a Szerkesztés – Másolás, Szerkesztés – Beillesztés, illetve Szerkesztés – Kivágás alatt. Ha az általunk készített alkalmazásban szeretnénk használni a vágólapot, elég megismerkednünk néhány alapfogalommal.

A vágólap az egyik leggyakrabban használt eszköz az alkalmazások közti kommunikációra. Ha a Windowsban az egyik alkalmazásból a másikba át szeretnénk rakni valamilyen szöveget vagy képet (esetleg más objektumot), leggyorsabban a vágólap segítségével tehetjük meg. A legtöbb felhasználó a vágólapot rutinosan használja.

A vágólapon egyszerre egy adat lehet. Ha a vágólapra elhelyezünk új adatot, az előző törlődik. Az adat természetesen lehet bármilyen hosszú. A vágólapon nem csak egyszerű szöveget helyezhetünk el, de különböző adattípusokat is, mint pl. bitképet, táblázatot, HTML kódot, stb. Az adatok típusát a vágólapon az **adat formátum**jának nevezzük. Ilyen formátum többféle lehet, ezek közül a leggyakrabban használtak:

- **CF_TEXT** – egyszerű szöveg, amely minden sor végén CR (Carriage Return – sor elejére) és LF (Line Feed – új sor) jeleket tartalmaz. A szöveg végét NUL (Null Character) karakter jelzi.

- **CF_BITMAP** – kép bitmap formátumban.
- **CF_PICTURE** – TPicture típusú objektum.
- **CF_TIFF** – TIFF formátumban levő kép.
- **CF_WAVE** – hang WAV formátumban.

24.1 A vágólap használata a programozásban

A legegyszerűbb műveleteket a vágólappal elvégzhetjük az alábbi három metódus segítségével:

- **CopyToClipboard**: a kijelölt szöveget a vágólapra másolja.
- **CutToClipboard**: a kijelölt szöveget a vágólapra helyezi át (kivágja az eredeti dokumentumból).
- **PasteFromClipboard**: a vágólapról belilleszti az adatokat a kurzor aktuális helyére.

Ezek a metódusok rendelkezésünkre állnak több komponensnél is, mint például az Edit, Memo, RichEdit komponenseknél.

A fenti metódusoknak a használata nagyon egyszerű, de néha előfordulhat, hogy a vágólapra pl. képet vagy más objektumot akarunk elhelyezni. Ebben az esetben a vágólaphoz a **TClipboard** osztály segítségével kell hozzáférnünk.

A Delphi-ben használhatjuk a globális objektumát ennek az osztálynak, ezt **Clipboard** néven érhetjük el. Ezt nem kell külön deklarálnunk és létrehoznunk, elég ha a programunk Uses részét kiegészítjük a **Clipbrd** unittal és máris elérhető lesz számunkra a Clipboard objektum. A munka ezzel az objektummal nagyon egyszerű,

ha például át szeretnénk másolni egy képet (pl. Image1) a vágólapra, azt a következő módon tehetjük meg:

```
Clipboard.Assign(Image1.Picture);
```

A TClipboard osztálynak több tulajdonsága is van, melyek közül a legfontosabbak:

- **AsText** – a vágólap tartalmát repretentálja szövegként.
- **Formats** – tömb, amely az összes olyan formátumot tartalmazza, melyek a vágolapon levő aktuális adatokra vonatkoznak.

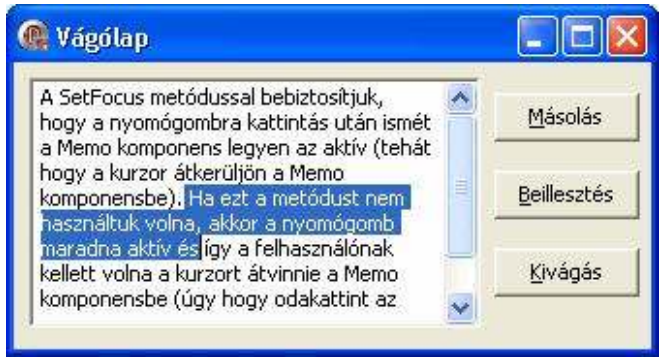
A TClipboard osztály legfontosabb metódusai:

- **Assign** – objektum (leggyakrabban kép) vágólapra való másolására szolgál.
- **Open**, **Close** – a vágólap megnyitására és bezárására szolgáló metódusok több adat vágólapra való helyezésekor.
- **HasFormat** – megállapítja hogy a vágólapon levő adatok adott formátumúak-e.

Vágólappal dolgozó szövegszerkesztő program Pelda34

Létrehozunk egy egyszerű programot, amely szemlélteti, hogyan használhatjuk ki a programunkban a vágólapot. A kijelölt szöveget egy gomb megnyomásával vágólapra másolhatjuk, a vágólapon levő szöveget pedig egy másik nyomógomb megnyomásával beszúrhatjuk a Memo komponensünkbe. A harmadik nyomógomb a kijelölt szöveget áthelyezi (kivágás) a vágólapra. A Memo komponensen

kívül tehát a programunkon használni fogunk még három nyomógombot (Button) is.



Az nyomógombok **OnClick** eseményeihez tartozó programkódok:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.CopyToClipboard;
  Memo1.SetFocus;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo1.PasteFromClipboard;
  Memo1.SetFocus;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Memo1.CutToClipboard;
  Memo1.SetFocus;
end;
```


A **SetFocus** metódussal bebiztosítjuk, hogy a nyomógombra kattintás után ismét a Memo komponens legyen az aktív (tehát hogy a kurzor átkerüljön a Memo komponensbe). Ha ezt a metódust nem használtuk volna, akkor a nyomógomb maradna aktív és így a felhasználónak kellett volna a kurzort átvinnie a Memo komponensbe (úgy hogy odakattint az egérrel).

Szöveges adat van a vágólapon? Pelda35

A következő program meghatározza, hogy a vágólapon levő adat a megadott formátumú-e. A következő alkalmazásunk egyetlen nyomógombot (Button) és egy Memo komponenst fog tartalmazni. Ha a vágólapon szöveges adat van, az a nyomógombra kattintás után a Memo komponensbe lesz beillesztve. Ellenkező esetben hibaüzenetet jelenít meg a programunk.

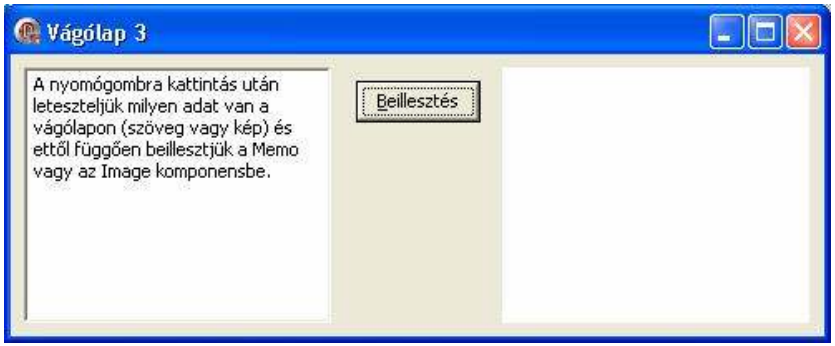
Az alkalmazás létrehozásakor ne felejtsük el beírni a **Clipbrd** modult a programunk Uses részébe.

```
uses
  Windows, Messages, ..., Dialogs, StdCtrls, Clipbrd;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Clipboard.HasFormat(CF_TEXT) then
    Memo1.Text := Clipboard.AsText
  else
    ShowMessage('A vágólapon nincs szöveg!');
end;
```

Arra, hogy a vágólapon szöveges adat van-e, a **HasFormat** függvényt használtuk CF_TEXT paraméterrel.

Hova rakjuk a vágólapról az adatot? Pelda36

Az alábbi programban is a **HasFormat** metódust fogjuk használni. Az alkalmazás egy „Beillesztés” feliratú nyomógombot fog tartalmazni, továbbá egy Memo és egy Image komponenst. A nyomógombra kattintás után leteszteljük milyen adat van a vágólapon (szöveg vagy kép) és ettől függően beillesztjük a Memo vagy az Image komponensbe.




```
uses
  Windows, Messages, …, StdCtrls, Clipbrd;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Clipboard.HasFormat(CF_TEXT) then
    Memo1.PasteFromClipboard
  else
   if Clipboard.HasFormat(CF_PICTURE) then
     Image1.Picture.Assign(Clipboard)
```



```
  else
    ShowMessage('A vágólapon ismeretlen adat van.');
end;
```

Vágólapfigyelő program Pelda37

Végül egy kicsit bonyolultabb példát mutatunk be. Ennek az alkalmazásnak a segítségével minden olyan szöveget evidálni tudunk majd, amely a vágólapon „keresztülment”. A programban néhány Windows API függvényt is fogunk használni.

Az alkalmazásunk egy Memo komponenst fog tartalmazni, melybe folyamatosan kiírunk minden olyan szöveget, amely bármelyik programban a vágólapra lett helyezve. Kezelni fogjuk a form **OnCreate**, **OnDestroy** eljárásait, továbbá kialakítjuk a **WM_DRAWCLIPBOARD** és **WM_CHANGECBCHAIN** üzenetek kezelésére szolgáló eljárásokat.

Hozzunk létre egy új alkalmazást, majd helyezzünk el rajta egy Memo komponenst. Most gondolkodjunk el azon, hogyan érzékeljük, ha megváltozott a vágólap tartalma. A Windows-ban létezik egy vágólapfigyelők lánca. Ez a lánc tartalmazza azokat az ablakokat, melyeknek van valamilyen összefüggésük a vágólappal. Minden ilyen ablakhoz eljut a **WM_DRAWCLIPBOARD** üzenet mindig, amikor a vágólap tartalma megváltozik. A form-unk létrehozásakor tehát besoroljuk ebbe a láncba a mi alkalmazásunkat is (annak fő ablakát) a **SetClipboardViewer** függvény segítségével.

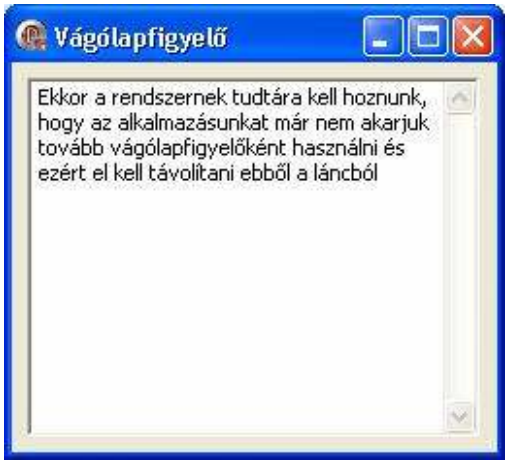
Ezekután ha a vágólap tartalma megváltozik, mindig kapunk egy WM_DRAWCLIPBOARD üzenetet. Ennek kezeléséhez létre kell hoznunk egy **OnDrawClipboard** metódust. A metódus megírásakor figyelembe kell vennünk, hogy a Windows nem küldi el ezt az üzenetet az össze vágólapfigyelőnek a láncban, hanem csak az elsőnek. Neki

tovább kell küldenie a következőnek, annak az utána következőnek és így tovább. Ezért a WM_DRAWCLIPBOARD üzenetet nekünk is tovább kell küldenünk a következő ablaklnak.

Az alkalmazásunk alapműködése már kész is lenne, ha nem kéne még egy fontos dolgot megoldanunk, mégpedig azt, hogy mi történik valamelyik vágólapfigyelő eltávolításakor a láncból. Ha a lánc valamelyik elemét el kell távolítani, akkor a rendszer az első ablaknak a láncban küld egy **WM_CHANGECBCHAIN** üzenetet. Ezért tehát ehhez az üzenethez is lére kell hoznunk egy kezelő eljárást, melynek logikusan **OnChangeCBChain** nevet adunk. Hasonlóan a WM_DRAWCLIPBOARD üzenethez, a WM_CHANGECBCHAIN üzenetet is tovább kell küldenünk a láncban soron következő ablaknak.

Az utolsó dolog, amit még meg kell tennünk összefügg az alkalmazásunk bezárásával. Ekkor a rendszernek tudtára kell hoznunk, hogy az alkalmazásunkat már nem akarjuk tovább vágólapfigyelőként használni és ezért el kell távolítani ebből a láncból. Ehhez a **ChangeClipboardChain** Windows API függvényt fogjuk felhasználni.

Hasonlóan az előző alkalmazásokhoz itt sem felejtsük el a programunk Uses részét kibővíteni a Clipbrd modullal!




```
uses
  Windows, Messages, ..., StdCtrls, Clipbrd;

type
  …
  private
    kovHandle: HWND;
    procedure OnDrawClipboard(
                  var msg: TWMDrawClipboard);
                        message WM_DRAWCLIPBOARD;
    procedure OnChangeCBChain(
                  var msg: TWMChangeCBChain);
                        message WM_CHANGECBCHAIN;
  …

…

procedure TForm1.FormCreate(Sender: TObject);
begin
  {besoroljuk a lancba az ablakunkat}
  kovHandle := SetClipboardViewer(Handle);
end;

procedure TForm1.OnDrawClipboard(
                  var msg: TWMDrawClipboard);
begin
```

```
  {ha szoveg van a vagolapon, berakjuk a memo-ba}
  if Clipboard.HasFormat(CF_TEXT) then
    Memo1.Lines.Add(Clipboard.AsText);
  {tovabbkuldjuk az uzenetet}
  SendMessage(kovHandle, WM_DRAWCLIPBOARD, 0, 0);
end;

procedure TForm1.OnChangeCBChain(
                  var msg: TWMChangeCBChain);
begin
  {ha a mi utanunk kovetkezo jelentkezik ki a
   lancbol, akkor megvaltoztatjuk a kovHandle-t}
  if msg.Remove = kovHandle then
    kovHandle := msg.Next
  else {egyebkent tovabbkuldjuk az uzenetet}
    SendMessage(kovHandle, WM_CHANGECBCHAIN,
                              msg.Remove, msg.Next);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  {kijelentkezunk a lancbol}
  ChangeClipboardChain(Handle, kovHandle);
end;

end.
```

A TForm1 deklarációjában létre kell hoznunk két eljárást két üzenet kezelésére. Ehhez a **message** kulcsszót használhatjuk a metódus deklarálása után megadva.

Az alkalmazás besorolása a vágólapfigyelők láncába a **SetClipboardViewer** függvény segítségével történik. Ennek paramétere az ablakunk azonosítója (handle). A függvény a láncban utánunk következő vágólapfigyelő handle-jét adja vissza, neki kell küldenünk tovább minden vágólappal kapcsolatos üzenetet. Ezt az azonosítót kovHandle változóba tesszük.

Megjegyzés: A Windowsban minden egyes ablaknak és vezérlőnek (gombnak, beviteli mezőnek, jelölőnégyzetnek, stb.) van egy azonosítója, az **ablak-leíró (windows handle)**. Ez a leíró adattípusát tekintve egy LongWord, vagyis egy előjel nélküli 32 bites egész szám. Ezzel a leíróval tudunk azonosítani a rendszerben egy adott objektumot.

Az **OnDrawClipboard** metódus mindig meg van hívva, ha WM_DRAWCLIPBOARD üzenetet kapunk (tehát ha változik a vágólap tartalma). A metódusban leellenőrizzük, hogy a vágólapon szöveg van-e, és ha igen, berakjuk a Memo komponensbe. Utána az üzenetet tovább kell küldenünk az utánunk következő vágólapfigyelőnek. Ehhez a **SendMessage** függvényt használjuk: ennek első paramétere a célablak azonosítója (kovHandle), második paramétere az üzenet, a következő két paramétere pedig az üzenet paramétereinek elküldésére szolgál.

Az **OnChangeCBChain** metódus mindig meg lesz hívva, ha WM_CHANGECBCHAIN üzenet érkezik, ami azt jelenti, hogy valamelyik vágólapfigyelő ki szeretne jelentkezni a láncból. Az üzenet **msg.Remove** paraméteréből megtudhatjuk, hogy a megszűnő ablak a mi utánunk következő-e (ebben az esetben más ablak lesz az utánunk következő, melyet a **msg.Next** paraméterből tudhatunk meg). Ha a megszűnő ablak nem az utánunk következő, akkor az üzenetet továbbküldjük az utánunk következőnek a láncban.

Saját magunk kijelentkezését a láncból a **ChangeClipboardChain** Windows API függvény segítségével hajtjuk végre, melynek első paramétere a kijelentkező ablak, tehát a mi ablakunk azonosítója (handle), a második paramétere pedig a láncban utánunk következő ablak azonosítója.

25 A Windows üzenetei

Ez a fejezet a Windows operációs rendszer üzeneteivel fog foglalkozni. Az üzenetek nem a Delphi specialitásai, ezek kizárólag az operációs rendszerhez kötődnek. Sokat programozhatunk a Delphi-ben az nélkül, hogy az üzenetekkel foglalkoznunk kellene. Sok esetben azonban a Delphi programozónak is a Windows üzeneteiteinek a használatához kell nyúlnia.

Az **üzenet** az egyik legfontosabb fogalom a Windows operációs rendszerben. Az üzenet egy információ arról, hogy a rendszerben valahol valami történt. Ha a felhasználó kattint az egérrel, beír valamilyen szöveget, megnyom egy billentyűt, vagy bármilyen más esemény történik, a Windows létrehoz egy üzenetet, amely minden fontos információt tartalmaz arról hol és mi történt.

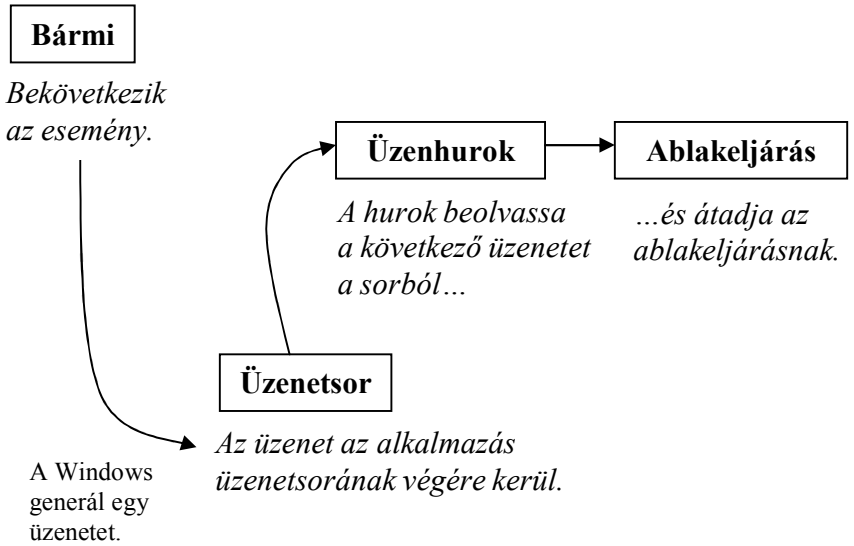
A Windows ezt az üzenetet elküldi azoknak az alkalmazásoknak, melyeket ez érint. Minden alkalmazásnak van **üzenetsor**a (message queue), melynek végére bekerül az új üzenet. Az alkalmazás „belsejében” fut egy ú.n. **üzenethurok** (message loop), melynek feladata csupán az, hogy lekérjen egy üzenetet az üzenetsorból és továbbítsa azt feldolgozásra. Az üzenet feldolgozást speciális eljárás, az ú.n. **ablakeljárás** (window procedure) hajtja végre. Az ablakeljárás „kezeli” az üzenetet, tehát megfelelő módon reagál rá.

Észrevehetjük a hasonlóságot ez között és a Delphi eseménykezelt programozása között. Ez nem véletlen, mivel az események között, amelyeket a Delphi-ben kezelünk, és az itt leírt üzenetek között van kapcsolat: a Delphi számunkra egyszerűen megérthető eseményekbe foglalja az üzeneteket, melyeket a

Windowstól kap. Ezeknek az eseményeknek a kezelő eljárásait programozzuk.

Az üzenetek a rendszerben a következő helyeken mennek keresztül:

1. **Valamilyen hely** – itt valamilyen esemény bekövetkezik, tehát ez az üzenet generálásának a helye.
2. **Az alkalmazás üzenetsora** – az 1. pontban generált üzenet az „érintett” alkalmazásban az üzenetsor végére kerül.
3. **Az alkalmazás üzenethurokja** – a sor elejéről az első üzenetet lekéri és továbbadja az ablakeljárásnak.
4. **Ablakeljárás** – végrehajt valamilyen reakciót az üzenetre, például kiír valamilyen információt, átrajzolja az ablakot, stb.



A Delphi-ben az üzenetek elsősorban a TMessage osztály segítségével vannak reprezentálva. Ezen kívül a Delphi tartalmaz speciális adattípust minden üzenettípusnak.

25.1 Üzenet kezelése Delphi-ben

Az első példa bemutatja, hogyan lehet Delphi-ben „elkapni" a rendszerüzeneteket. Figyelni fogjuk a WM_DELETEITEM üzenetet, amelyet a ListBox-ot vagy ComboBox-ot tartalmazó ablaknak küld a rendszer akkor, ha a lista elemeiből egy vagy több elem törlődni fog.

Az alkalmazás egy ListBox komponenst és egy nyomógombot fog tartalmazni. A nyomógomb megnyomásakor töröljük a listából az aktuális (kijelölt) elemet. Ez a törölt elemünk azonban nem fog elveszni, hanem az üzenet érkezésekor elmentjük egy *elemek.log* nevű állományba a törlés dátumával és idejével együtt. Pelda38

A nyomógomb OnClick eseményéhez tartozó eljáráson kívül létrehozunk egy kezelő eljárást a WM_DELETEITEM üzenet kezelésére. Ez az eljárás két részből fog állni: a TForm1 osztály private részében szereplő deklarációból és az eljárás implementációjából (eljárás törzséből).


```
…
type
  TForm1 = class(TForm)
    ListBox1:TListBox;
    Button1:TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    procedure OnListBoxDelete(var msg:
                TWMDeleteItem); message WM_DELETEITEM;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.OnListBoxDelete(var msg:
                                        TWMDeleteItem);
const
  LOGFORMAT = '%s - törölve: %s, %s';
var
  F: TextFile;
begin
  AssignFile(F, 'elemek.log');
  if not FileExists('elemek.log') then
    begin
    Rewrite(F);
    WriteLn(F, 'Törölt elemek listája');
    WriteLn(F, '**********************************');
    end
  else
    Append(F);
  Writeln(F, Format(LOGFORMAT,
       [ ListBox1.Items[msg.DeleteItemStruct.ItemID],
         DateToStr(Date), TimeToStr(Time) ]));
  CloseFile(F);
  inherited;
```

```
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Delete(ListBox1.ItemIndex);
end;
…
```

Az üzenetet kezelő eljárás létrehozásánál a deklarációban szerepelnie kell az üzenetet pontosító paraméternek. Ennek a paraméternek vagy egy általános **TMessage** típusúnak kell lennie, vagy az üzenethez tartozó konkrét típusnak. Ha tehát a **WM_DELETEITEM** üzenetet kezeljük, akkor a **TWMDeleteItem** típust használhatjuk. A deklaráció végén a **message** kulcsszót kell használnunk, amely után annak az üzenetnek a nevét kell megadnunk, amelyet kezelni fogunk.

Az eljárás implementációjában a fejléc után már nem kell megadnunk a message kulcsszót.

Az üzenetet kezelő eljárás végén használtuk az **inherited** kulcsszót. Ezt a kulcsszót ajánlatos minden üzenetet kezelő eljárás végén használnunk (ha csak nem készítünk valamilyen speciális eljárást az üzenet kezelésére). Ennek a kulcsszónak a segítségével a mi eljárásunk lefutása után meghívjuk a WM_DELETEITEM standard kezelését úgy, ahogy az az ősében van definiálva. Miért? Ha például „kijavítjuk” a WM_PAINT üzenetek kezelését és elfelejtjük meghívni az ős kezelését az **inherited** kulcsszó segítségével, akkor nem lesz kirajzolva semmi sem (ha csak a mi kezelésünkben nem végezzük el kézileg a kirajzolást). Bizonyos esetekben készakarva nem hívjuk meg az eljárás ősben levő kezelését, de leggyakrabban ennek ellentettje az igaz. Az **inherited** kulcsszó után nem kell feltüntetnünk semmilyen további információkat, a Delphi tudja, melyik metódusról (melyik ősről)

van szó. Ezt az üzenet specifikációjából tudja meg, amely a **message** kulcsszó után van megadva a deklarációban.

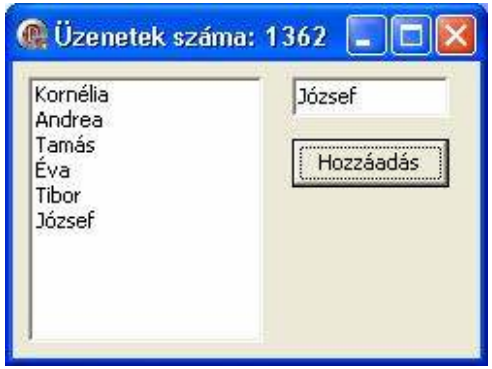
25.2 Beérkező üzenetek számlálása

A következő példa szemlélteti, hogy megközelítőleg mennyi üzenet érkezik egy alkalmazáshoz a futása során. Mondtuk, hogy a Delphi-ben a legtöbb standard üzenet eseményekbe van áttranszformálva. Ez azonban nem jelenti azt, hogy az alkalmazás a rendszertől ne kapna üzeneteket. Hasonlóan kap üzeneteket, mint ahogy az előző példában az alkalmazásunk kapta a WM_DELETEITEM üzeneteket, de az üzenetek fogadásához nem kell kézileg eljárásokat definiálnunk, mivel a Delphi-ben az események segítségével megoldhatjuk a reakciókat az üzenetekre.

Az alábbi példában egy érdekes eseménnyel is megismerkedhetünk, mégpedig az Application objektum **OnMessage** eseményével. Ez az esemény bekövetkezik mindig, ha az alkalmazásunk valamelyik komponense valamilyen üzenetet kap a rendszertől. Ebből adódik, hogy az **Application.OnMessage** esemény kezelése az egyik legleterheltebb eljárása az egész alkalmazásnak. Ezért soha ne írjunk bele több programkódot, mint amennyi szükséges.

A készülő alkalmazásunk egy Edit komponenst, egy listát és egy nyomógombot fog tartalmazni. Az ablak (form) feliratában (Caption) folyamatosan megjelenítjük az alkalmazás által kapott üzenetek számát. A form OnCreate eseményén és a nyomógomb OnClick eseményén kívül tehát kezelni fogjuk az Application.OnMessage eseményét is. Pelda39

A bejövő üzenetek megszámlálásához kihasználjuk azt, hogy az Application.OnMessage eseménye bármilyen bejövő üzenetről „értesül”. Létrehozzuk tehát ennek az eseménynek a kezelésére szolgáló eljárást. Ehhez meg kell írnunk az AppOnMessage eljárást és „össze kell kapcsolnunk” az Application objektum OnMessage eseményével. Szükségünk lesz még egy „UzenetekSzama” nevű private változóra, melyben az üzenetek számát fogjuk számolni.




```
…
type
  TForm1 = class(TForm)
  …
  private
    { Private declarations }
    UzenetekSzama: Integer;
    procedure AppOnMessage(var Msg: TMsg; var
                                   Handled: Boolean);
  …

…
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := AppOnMessage;
  UzenetekSzama := 0;
```



```
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Edit1.Text <> '' then
    ListBox1.Items.Add(Edit1.Text);
end;

procedure TForm1.AppOnMessage(var Msg: TMsg; var
                                    Handled: Boolean);
begin
  Inc(UzenetekSzama);
  Caption := 'Üzenetek száma: '
                            + IntToStr(UzenetekSzama);
end;
…
```

Az OnMessage esemény kezelése az üzenetről hamarabb értesül, mint maga a címzett. Azok az üzenetek, melyek nincsenek az OnMessage esemény kezelésében „leállítva“, mennek tovább az üzenet címzettjéhez. Természetesen van arra is lehetőség, hogy az üzenetet mi is kezeljük és az eredeti címzetthez is eljusson. Ezt a **Handled** paraméterrel állíthatjuk be, amely azt jelzi, hogy az üzenet menjen-e tovább mint kezeletlen, vagy nem. Ha nincs kezelve, akkor az üzenet fel lesz dolgozva „standard úton“. Ez alatt vagy az esemény kezelését értjük (pl. OnKeyDown), vagy a „default“ reakciót, amely a komponens programozója által van definiálva. Érdekességképpen próbáljuk meg az AppOnMessage eljárást kiegészíteni a következő sorral:

```
if Msg.message = $0100 then Handled := true;
```

Észrevettük a különbséget? Ha az efektus nem elegendő számunkra, beírhatjuk helyette csak a `Handled := true;` parancsot, előtte azonban mentsük el az össze munkánkat!

Az Application.OnMessage esemény összekapcsolását az AppOnMessage eljárással a már megszokott módon tesszük meg az ablak OnCreate eseményének kezelésében.

Egy kis munka után az alkalmazásunkban láthatjuk, hogy az üzenetek száma, melyet az alkalmazásunk kapott egyáltalán nem kevés. Figyelembe kell vennünk azt is, hogy ez tényleg csak egy nagyon egyszerű alkalmazás, bonyolultabb alkalmazásoknál az üzenetek száma több tízezer lehet.

Megjegyzés: Az OnMessage esemény kezelését (összekapcsolását az AppOnMessage eljárással) nem csak programilag realizálhatjuk, de tervezési időben az Object Inspectoron keresztül is. Ehhez azonban előbb az alkalmazásunkba be kell raknunk egy ApplicationEvents komponenst az Additional kategóriából.

25.3 Felhasználó által definiált üzenetek küldése

Tekintettel arra, hogy definiálhatunk saját (felhasználói) üzeneteket, felhasználhatjuk az üzenetek küldésének mechanizmusát alkalmazások közti komunikációra vagy egy alkalmazáson belüli komunikációra is.

Ahogy eddig is láthattuk, az üzenetek küldése remek lehetőség az információ átadására az operációs rendszertől az egyes alkalmazásoknak. Amint a rendszerben valami történik, a Windows küld az alkalmazásnak egy üzenetet. Ezt a az alapelvet általánostíthatjuk: miért ne cserélhetne az üzenetek segítségével két alkalmazás is információt egymás között? Továbbmenve: létezik valamilyen ok, amiért ne használhatná ki ezt a rendszert egy alkalmazás információk

küldésére például egy egyik és másik ablaka között a program futása alatt?

Felmerülhet a kérdés: miért küldjön egy alkalmazás saját magának üzenetet (pl. a számítás befejezéséről), ha egyenesen meghívhat valamilyen eljárást vagy függvényt? Ez igaz, de az üzenetek használata néhány előnnyel jár a kalsszikus metódusokkal szemben:

- az üzenetet el lehet küldeni az nélkül, hogy pontosan ismernénk a címzettjének a típusát,
- nem történik semmi sem, ha a címzett nem reagál az üzenetre,
- az üzenetet el lehet küldeni egyszerre több címzettnek is (ú.n. broadcasting).

Saját üzenet definiálása nagyon egyszerű. A Windows lehetőséget ad felhasználói üzenetek küldésére. Erre a WM_USER-től a $7FFF sorszámú üzeneteket használhatjuk. A WM_USER egy a rendszerben definiált konstans, amely segítségünkre van saját konstansok létrehozására (úgy használhatjuk mint alapot, melyhez hozzáadunk valamennyit). Az üzenet létrehozásához nem kell semmi mást tennünk, mint definiálnunk saját konstans. Az alábbi példában definiálink egy WM_KESZ üzenetet:

```
const
  WM_KESZ = WM_USER + 100;
```

A következő alkalmazás a nyomógombra kattintás után elindít egy számítást (egy változó növekedését 1-től 10000-ig). A változó értéke mindig ki lesz írva az alkalmazás ablakának feliratában. A számítás befejezésekor a számítást végző eljárás küld egy üzenetet az

ablaknak (form) a számítás befejezéséről. A form erre az üzenetre egy információs üzenetablak megjelenítésével reagál. Pelda40

A programozás során definiálnunk kell a WM_KESZ konstanst. Ennek a definiálása meg kell hogy előzze a TForm1 osztály deklarálását, mivel benne már használjuk ezt a konstanst. Deklatáljuk az üzenet kezelését végző eljárást (WMKesz) is. A Szamitas metódus impementációjában valamilyen számítás szimulálását végezzük el. A számítás elvégzése után ez a metódus küld egy üzenetet az alkalmazásnak. Az utolsó lépés a WMKesz eljárás imlementálása, amely kiírja az információt a számítás befejezéséről.

```
…
const
  WM_KESZ = WM_USER + 100;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    procedure WMKesz(var Msg: TMessage);
                                    message WM_KESZ;
    procedure Szamitas;
  public
    { Public declarations }
  end;

…

procedure TForm1.Szamitas;
var
  i,j: integer;
begin
  // szamitas szimulalasa
  for i:=1 to 10000 do
    begin
```



```
    Caption := IntToStr(i);
    j:=i;
    end;
  // felhasznaloi uzenet kuldese
  SendMessage(Self.Handle, WM_KESZ, j, 0);
end;

procedure TForm1.WMKesz(var Msg: TMessage);
begin
  ShowMessage('A számítás befejeződött. Eredmény = '
                              + IntToStr(Msg.WParam));
end;

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

Az üzenet elküldésére a **SendMessage** rendszerfüggvényt használjuk. A függvény meghívásakor csak az ablak-leírót (handle) kell ismernünk, ami a mi esetünkben a Self.Handle. A függvény átadja az üzenetet egyenesen a címzett üzenetkezelő eljárásának (nem az üzenetsornak) és addig nem tér vissza, amíg az üzenet kezelése be nem fejeződik. A SendMessage-nak négy paramétere van:

- a címzett azonosítója (handle),
- az üzenet azonosítója,
- az üzenet két további paramétere (a kezelő eljárásban a Msg.WParam ill. Msg.LParam segítségével hivatkozhatunk rájuk).

Az üzenet küldésére használhatjuk a **Perform** metódust is, amelyet minden komponens tartalmaz. Segítségével üzenetet küldhetünk egyenesen az adott komponensnek.

Továbbá üzenet küldésére használható még a **PostMessage** eljárás is, amely hasonló a SendMessage eljáráshoz, csak ez nem vár az üzenet kezelését végző eljárásból való visszatérésre, hanem a program azonnal folytatódik a PostMessage eljárás urán.

25.4 A képernyő felbontásának érzékelése

Ha szükségünk van az alkalmazásban a képernyő felbontásának, színmélységének változását figyelni, használhatjuk erre a **WM_DISPLAYCHANGE** nevű beérkező üzenet kezelését. Az alábbi alkalmazás mindig megjelenít egy információs ablakot, ha változott a képernyő beállítása. Pelda41

```
…
type
  TForm1 = class(TForm)
  private
    { Private declarations }
    procedure WMDisplayChange(var Msg: TMessage);
                            message WM_DISPLAYCHANGE;
  public
    { Public declarations }
  end;

…

procedure TForm1.WMDisplayChange(var Msg: TMessage);
begin
  ShowMessage('A képernyő beállítása megváltozott.');
  inherited;
end;
…
```


25.5 A Windows néhány kiválasztott üzenete

A Windowsban rendgeteg fajta üzenet van. Ezek többségét megtalálhatjuk a súgóban ill. utána nézhetünk az Interneten. Itt most ezekből felsorolunk néhányat:

Üzenet azonosítója:	Értéke:	Mikor kapunk ilyen üzenetet?
WM_ACTIVATE	$0016	Ha az ablak aktiválva vagy deaktiválva van.
WM_KEYDOWN	$0100	Ha egy billentyű le lett nyomva a billentyűzeten.
WM_KEYUP	$0101	Ha egy billentyű fel lett engedve a billentyűzeten.
WM_LBUTTONDOWN	$0201	Ha a felhasználó lenyomja a bal egérgombot.
WM_MOUSEMOVE	$0200	Ha a felhasználó mozgatja az egeret.
WM_PAINT	$000F	Ha az ablaknak át kell rajzolni magát.
WM_TIMER	$0113	Ha bekövetkezik az időzítő eseménye.
WM_QUIT	$0012	Ha kérés érkezett az alkalmazás bezárására.

26 További hasznos programrészek

Ebben a részben rövid programrészek, illetve olyan eljárások, metódusok találhatók, melyek a programozás során hasznunkra lehetnek, de eddig még nem esett róluk szó.

26.1 Hang lejátszása az alkalmazásban

Sokszor szeretnénk programunkat színesebbé tenni különféle rövid hangok lejátszásával, például egy gomb megnyomásakor vagy egy játékprogramban a kincs megtalálásakor, stb. Ezt egyszerűen megtehetjük a **PlaySound** eljárással, amely az **MMSystem** unitban található. Ennek az eljárásnak a segítségével WAV formátumban levő hangot, zenét játszhatunk le.

A következő program csak egy nyomógombot fog tartalmazni, melynek megnyomásakor egy hangot játszunk majd le. Pelda42

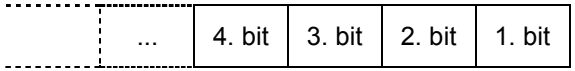
Az alkalmazás forráskódjában ne felejtsül el beírni a programunk uses részébe az MMSystem modul használatát. Továbbá a nyomógomb OnClick eseményéhez tartozó eljárást fogjuk megírni:

```
…

uses
  Windows, Messages, … , StdCtrls, MMSystem;

…

procedure TForm1.Button1Click(Sender: TObject);
begin
  PlaySound('boink.wav',0,SND_ASYNC);
```



```
end;
```

A hangot a már említett **PlaySound** eljárással játszuk le, melynek első paramétere a fájl neve (elérési útvonallal együtt, ha nem ugyanabban a mappában található, ahol az alkalmazásunk), második paramétere mindig 0, harmadik paramétere pedig egy ú.n. flag. Ez utóbbi azt jelenti, hogy egy olyan szám, melynek mindegyik bitje valamilyen beállítási lehetőség, pl.:



- 1. bit (**SND_ASYNC**) – ha értéke 1, akkor a lejátszás aszinkron, különben szinkron (SND_SYNC). Aszinkron lejátszásnál elindul a hang lejátszása és az alkalmazás fut tovább. Szinkron lejátszásnál az alkalmazás leáll és vár a hang lejátszásának befejeződésére.
- 2. bit (**SND_NODEFAULT**) – ha értéke 1 és a lejátszás során hiba következik be (nincs meg a lejátszandó hang), akkor nem fogja lejátszani az alapértelmezett hangot.
- 3. bit (**SND_MEMORY**) – ha értéke 1, akkor a hangot a memóriából (nem külső fájlból) fogja lejátszani. Ebben az esetben az eljárás első paramétere nem a fájl nevét, hanem a memória azon részére mutató pointer, ahol a hang van.
- 4. bit (**SND_LOOP**) – ha értéke 1, akkor a hangot körbe-körbe fogja lejétszani mindaddig, amig az eljárás nem lesz újból meghívva. A hang lejátszását ebben az esetben a

PlaySound(NIL, 0, SND_ASYNC); paranccsal lehet. Az SND_LOOP-al együtt az SND_ASYNC-t is szükséges beállítani.

Mindegyik bitre használható egy konstans, ezeknek a zárójelben megadott SND-vel kezdődő neve van. Ha egyszerre több bitet szeretnénk beállítani, ezeket az **or** művelet segítségével kapcsolhatjuk össze. Pl:

PlaySound('boink.wav', 0 , SND_LOOP or SND_ASYNC);

Meg kell hogy jegyezzük, hogy az ilyen fajta lejátszásnál mindig a futtatható állománnyal együtt a WAV állományt is át kell másolnunk a programunk terjesztésekor, mivel innen játsza le a hangokat.

26.2 Erőforrás (resource) állományok használata

Eddig ha valamilyen bitképet akartunk felhasználni az alkalmazásunkban, három lehetőségünk volt:

1. A BMP fájlokat külön tároltuk és pl. a LoadFromFile metódussal beolvastuk az állományba – ennek hátránya, hogy a BMP fájlokat is mindig az EXE mellé kell másolnunk, és ha véletlenül nem tettük oda, a program nem tudta beolvasni – hibát írt ki vagy nem jelent meg a programban a kép.
2. Ha kevés BMP állományunk volt, akkor azokat berakhattuk egy-egy Image komponensbe, így a fordítás után az belekerült az EXE fájlba, elég volt ezt átmásolnunk a programunk terjesztésénél.

3. Több, hasonló BMP állomány esetén használhattunk egy ImageList komponenst és ebben tárolhattuk a képeinket.

Most megismerkerünk a negyedik lehetőséggel is, az ú.n. **erőforrás (resource) fájlok** használatával.

Ezek segítségével nem csak BMP, de bármilyen más típusú állományok is (akár az előző fejezetben használt WAV fájlok is) csatolhatók a lefordított EXE állományunkhoz, így azokat terjesztésnél nem kell majd külön hozzámásolgatnunk, egy EXE-ben benne lesz minden.

Első példánkban hozzunk létre egy form-ot, melyen helyezzünk el egy Image komponenst és három nyomógombot (Button). Az egyes nyomógombok megnyomásakor mindig más képet akarunk majd megjeleníteni az Image komponensben. A képeket a lefordított futtatható (EXE) állományhoz fogjuk csatolni és innen fogjuk őket használni (betölteni az Image komponensbe). Pelda43

Ehhez először is létre kell hoznunk egy resource fájlt. Ehhez valamilyen egyszerű szövegszerkesztő (pl. jegyzettömb) segítségével

hozzunk lérte **valami.rc** állományt (resource script), amely a következő sorokat tartalmazza:

```
kep1 RCDATA "jeghegy.bmp"
kep2 RCDATA "rozsa.bmp"
kep3 RCDATA "sivatag.bmp"
```

Majd használjuk Borland erőforrás-szerkesztőjét (**brcc32.exe**) a létrehozott RC fájl lefordításához: `brcc32.exe valami.rc`

Ha a brcc32.exe program mappája nincs benne a számítógép PATH-jában, akkor a teljes útvonala segítsével érhetjük el ("c:\Program Files\Borland\Bds\3.0\Bin\brcc32.exe"). Ekkor létrejött egy **valami.res** nevű állomány.

A következő fordítási direktívával utasítjuk a fordítót, hogy az elkészült erőforrás-fájlt építse bele a programba:

```
{$R *.DFM}
{$R PELDA.RES}
```

A programból e képet a következőképpen érhetjül el (tölthetjük be az Image komponensbe):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Stream : TResourceStream;
begin
  Stream := TResourceStream.Create(HInstance,
```



```
                                    'kep1',RT_RCDATA);
  try
    Image1.Picture.Bitmap.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

Az resource stream létrehozásakor a második paraméter adja meg a kép azonosítóját (ahogy a valami.rc állományban megadtuk).

A másik három nyomógombhoz tehát ugyanilyen az eljárás fog kerülni annyi különbséggel, hogy ott a kep2 ill. kep3 lesz megadva második paraméternek.

Próbjuk meg hasonlóan megoldani, hogy a Pelda42-es feladatban levő hang (WAV állomány) az exe állományhoz legyen csatolva. Pelda44

Ehhez először is hozzuk létre az RC állományt, nevezzük el például hangok.rc-nek. Tartalma:

```
boink RCDATA "boink.wav"
```

Ezt fordítsuk le a `brcc32.exe hangok.rc` parancs segítségével. Így kapunk egy hangok.res állományt. Ezt felhasználjuk a programunkban a hang memóriából való lejátszására a következő képpen:

```
…

uses
  Windows, Messages, … , StdCtrls, MMSystem;

…
```

```
{$R *.dfm}
{$R hangok.res}

procedure TForm1.Button1Click(Sender: TObject);
var
  ResStream: TResourceStream;
begin
  ResStream := TResourceStream.Create(HInstance,
                                'boink', RT_RCDATA);
  try
   PlaySound(ResStream.Memory, 0,
                           SND_MEMORY or SND_ASYNC);
  finally
   ResStream.Free;
  end;
end;

…
```

26.3 Kép mozgatása a kurzor billentyűk segítségével

A következő programban csupán egy képet (Image) helyezzünk el a form-on. Ezt a képet mozgassuk a nyilak segítségével. Pelda45

Ehhez elég megírnunk az OnKeyDown eseményhez tartozó eljárást:

```
…

procedure TForm1.FormKeyDown(Sender: TObject; var
Key: Word;
  Shift: TShiftState);
begin
 case Key of
  VK_DOWN: if Image1.Top+Image1.Height < ClientHeight
                      then Image1.Top:=Image1.Top+3;
```



```
  VK_UP: if Image1.Top > 0 then
                              Image1.Top:=Image1.Top-3;
  VK_RIGHT: if Image1.Left+Image1.Width < ClientWidth
                     then Image1.Left:=Image1.Left+3;
  VK_LEFT: if Image1.Left > 0 then
                             Image1.Left:=Image1.Left-3;
 end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DoubleBuffered := true;
end;

…
```

26.4 Objektumokból álló tömb

Objektumokat (komponenseket) a program futása során is létrehozhatunk. Ha több komponenst szeretnénk használni az alkalmazásunkban, akkor sokszor célszerű egy olyan tömb létrehozása, amely komponensekből (objektumokból) áll.

A következő példában egy TImage komponensből álló tömböt használunk. Ha egérrel az ablakba kattintunk, a kattintás helyén létrehozunk egy TImage komponenst (mely egy csillagot ábrázol). Ezeket a komponenseket egy tömbben tároljuk. A példában maximum 50 ilyen komponenst tartalmazó tömböt használunk. A létrehozott komponenseket egy Timer segítségével lefele mozgatjuk, közben jobbra-balra is mozgatva egy sin(5x) függvény segítségével. Pelda46

A programunk tervezési időben csak a Timer komponenst fogja tartalmazni, a TImage komponenseket (melyeket a tömbben tárolunk) a program futása során hozzuk majd létre.

A komponensekben megjelenítendő képet az előző fejezet szerint egy erőforrás (resource) fájl segítségével a lefordított programhoz csatoljuk és innen olvassuk be.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms, Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject;
                Button: TMouseButton;
                Shift: TShiftState; X, Y: Integer);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    a: array[1..50] of TImage;
    n: integer;
    cs: TBitmap;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
{$R kepek.res}

procedure TForm1.FormCreate(Sender: TObject);
var
  res: TResourceStream;
```



```
begin
  DoubleBuffered := true;
  n := 0;
  res := TResourceStream.Create(HInstance, 'csillag',
                                          RT_RCDATA);
  cs := TBitmap.Create;
  cs.LoadFromStream(res);
  res.Free;
end;

procedure TForm1.FormMouseDown(Sender: TObject;
                Button: TMouseButton;
                Shift: TShiftState; X, Y: Integer);
begin
  if n<50 then begin
              a[n+1] := TImage.Create(Self);
              a[n+1].Parent := Self;
              a[n+1].Enabled := false;
              a[n+1].Autosize := true;
              a[n+1].Transparent := true;
              a[n+1].Picture.Bitmap := cs;
              a[n+1].Left := X - a[n+1].Width div 2;
              a[n+1].Top := Y - a[n+1].Height div 2;
              inc(n);
              end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  for i:=1 to n do a[i].Free;
  cs.Free;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: integer;
begin
  for i:=1 to n do
    begin
    a[i].Top := a[i].Top+1;
    if a[i].Top>Height then a[i].Top := -a[i].Height;
    a[i].Left := a[i].Left
              - round(sin((a[i].Top-1)*PI/180*5)*90)
```

```
            + round(sin(a[i].Top*PI/180*5)*90);
    if a[i].Left<-a[i].Width then a[i].Left := Width;
    if a[i].Left>Width then a[i].Left := -a[i].Width;
    end;
end;

end.
```

26.5 Aktuális dátum, idő lekérdezése

A programozás során gyakran előfordulhat, hogy szükségünk van az aktuális dátum és idő lekérdezésére. Erre több adatszerkezetet, függvényt és eljárást találhatunk, melyek segítségével a dátummal és az idővel dolgozhatunk.

A *Delphi*-ben a dátum és idő megjegyzésére szolgáló alaptípus a **TDateTime**. Ez a típus a Double lebegőpontos szám típussegítségével van definiálva. Képzeljük el, hogy valamilyen TDateTime típusú változóban tároljuk az aktuális dátumot és időt. Ekkor valójában a tizedesszám egész részében van elhelyezve az 1899.12.30. óta eltelt napok száma, a tizedes részben pedig az tárolódik, hogy a nap hányad része telt el éjfél óta (a 24 óra hányad része telt el éjfél óta). Ezért ha például két időpont között eltelt időre van szükségünk, elég ha kivonjuk egymásból a két dátumot.

Néha a dátummal és az idővel való munkánk során szükségünk lehet valamelyik Windows API függvény használatára (pl. a SetSystemTime-ra, melyel beállíthatjuk a rendszeridőt). Ebben az esetben szükséges, hogy a dátumot és az időt olyan formátumban tároljuk, amely „tetszik” a Windows-nak. Ez a formátum (adattípus) a *Delphi*-ben a **TSystemTime**.

A két adattípus közötti átváltásra egy függvény és egy eljárás szolgál:

```
function SystemTimeToDateTime
      (SystemTime: TSystemTime): TDateTime;

procedure DateTimeToSystemTime
      (DateTime: TDateTime;
            var SystemTime: TSystemTime);
```

Néhány további metódus a dátummal és idővel való munkához:

Now	Aktuális időt és dátumot adja vissza.
Date	Aktuális dátumot adja vissza.
Time	Aktuális időt adja vissza.
DateTimeToStr	A TDateTime értéket szöveggé alakítja a formátum megadásának lehetőségével (Format paraméter).
DateToStr	A TDateTime adattípusból a a dátumot alakítja szöveggé.
TimeToStr	A TDateTime adattípusból az időt alakítja szöveggé.
DayOfWeek	A megadott TDateTime adattípusból visszaadja a nap sorszámát a hétben. A eredmény 1 (vasárnap) és 7 (szombat) közötti szám.
IsLeapYear	Értéke egy logikai változó, mely megadja hogy a függvény paraméterében levő év (Word típusú – egész szám) szökőév e.

Aktuális dátum és idő lekérdezése Pelda47

A következő program három nyomógomb segítségével lekérdezi az aktuális dátumot, időt, mindkettőt és kiírja egy Label komponensbe.

Az egyes nyomógombokhoz tartozó programkód:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := 'Mai dátum: ' + DateToStr(Date);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Label1.Caption := 'Idő: ' + TimeToStr(Time);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Label1.Caption := 'Dátum és idő: ' +
                                DateTimeToStr(Now);
end;
```


A számítás idejének mérése Pelda48

Ha az alkalmazásunk egy hosszabb számítást tartalmaz, lemérhetjük a számítás idejét és kiírhatjuk a felhasználónak. Ehhez a **GetTickCount** függvényt fogjuk használni:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i, startido, ido: Cardinal;
begin
 startido := GetTickCount;
 for i:=1 to 5000 do
   begin
   Label1.Caption := IntToStr(i);
   Application.ProcessMessages;
   end;
 ido := GetTickCount - startido;
 ShowMessage('A számítás ' + FloatToStr(ido/1000) +
                              ' másodpercig tartott.')
end;
```

A **GetTickCount** Windows API függvény megadja a Windows utolsó indítása óta eltelt időt milliszekundumokban. Ha ezt az időt elrakjuk egy változóba a számítás előtt, majd a számítás után kiszámoljuk a különbséget, megkapjuk a számítás idejét milliszekundumokban. Ezt az eredményt elég elosztanunk 1000-rel és megkapjuk a számítás idejét másodpercekben.

26.6 INI állományok, rendszerleíró adatbázis (regiszterek) használata

A felhasználó az alkalmazásunk használatakor sokszor beállít különféle beállításokat, melyeket szeretné, ha legközelebb is beállítva maradnának. Például, beállítja az ablak elhelyezkedését a képernyőn, az ablak háttérszínét, a kezdeti könyvtárat a dokumentumok megnyitásához és mentéséhez, stb.

Ahhoz, hogy a programunk ezeket a beállításokat megjegyezze, nekünk mint programozónak két lehetőségünk van:

- A beállításokat megjegyezzük valamilyen saját formátumban, például elmentjük egy szöveges vagy bináris állományba. Ez a felhasználó számára problámamentes, viszont a programozónak plusz munkát jelent.
- A beállításokat valamilyen általánosan működő mechanizmus segítségével mentjük el. Ez a felhasználó számára nem jelent semmilyen változást, viszont a programozó munkáját megkönnyíti. Ha ezt a módszert választjuk, két lehetőségünk van: a beállításokat **inicializációs (*.ini) állományokba** mentjük el vagy a beállítások tárolására felhasználjuk a Windows **rendszerleíró adatbázisát (regiszterek)**.

A beállítások tárolása INI állományokban Pelda49

Az alábbi program szemlélteti, hogyan tárolhatunk beállításokat inicializációs (*.ini) fájlokban. Tárolni fogjuk az ablak pozícióját a képernyőn, méretét és az beviteli dobozban található szöveget. A

programból való kilépéskor az adatokat elmentjül INI fájlokba, a program indításakor pedig beolvassuk onnan.

```
…

uses
  Windows, Messages, SysUtils, … , IniFiles;

…

procedure TForm1.FormCreate(Sender: TObject);
var
  IniFajl: TIniFile;
begin
  // letrehozunk egy TIniFile tipusu objektumot
  IniFajl := TIniFile.Create(
          ChangeFileExt(Application.ExeName,'.ini'));
  // megprobaljuk beolvasni az adatokat a fajlbol
  try
    Edit1.Text :=
                IniFajl.ReadString('Edit','Text','');
    Top := IniFajl.ReadInteger('Form','Top',100);
    Left := IniFajl.ReadInteger('Form','Left',100);
    Width := IniFajl.ReadInteger('Form','Width',153);
    Height :=
            IniFajl.ReadInteger('Form','Height',132);
    if IniFajl.ReadBool('Form','Maximized',false)
    then
      WindowState := wsMaximized
    else
```

```
      WindowState := wsNormal;
  // vegul felszabaditjuk az objektumot a memoriabol
  finally
    IniFajl.Free;
  end;
end;

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

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
var
  IniFajl: TIniFile;
begin
  // letrehozunk egy TIniFile tipusu objektumot
  IniFajl := TIniFile.Create(
          ChangeFileExt(Application.ExeName,'.ini'));
  // megprobaljuk kiirni az adatokat a fajlbol
  try
    IniFajl.WriteString('Edit','Text',Edit1.Text);
    IniFajl.WriteInteger('Form','Top',Top);
    IniFajl.WriteInteger('Form','Left',Left);
    IniFajl.WriteInteger('Form','Width',Width);
    IniFajl.WriteInteger('Form','Height',Height);
    IniFajl.WriteBool('Form','Maximized',
                           WindowState=wsMaximized);
  // vegul felszabaditjuk az objektumot a memoriabol
  finally
    IniFajl.Free;
  end;
end;

…
```

Ahhoz, hogy dolgozhassunk az INI fájlokkal, programunk uses részét egészítsük ki az **IniFiles** unittal.

A programban ezután létrehozhatun egy **TIniFile** típusú objektumot a **Create** metódus segítségével, melynek paramétereként megadjuk az inicializációs állomány nevét.

A beállítások elmentéséhez a **WriteString**, **WriteInteger** és **WriteBool** függvényeket használjuk, az állományból való beolvasáshoz pedig a **ReadString**, **ReadInteger** és **ReadBool** függvényeket.

Végül, ha már nincs szükségünk a létrehozott TIniFile típusú objektumra, felszabadítjuk azt a memóriából a **Free** metódus segítségével.

A programban használtuk még az **Application.ExeName** és **ChangeFileExt** függvényeket is. Ezeket csupán azért alkalmaztuk, hogy az ini fájlunknak ugyanaz a neve legyen, mint a futtatható állománynak, exe helyett ini kiterjesztéssel.

Ha most megnézzük, mit tartalmaz a programunk által létrehozott ini állomány, ezt láthatjuk:

```
[Edit]
Text=Szia

[Form]
Top=416
Left=396
Width=153
Height=132
Maximized=0
```

Rendszerleíró adatbázis (regiszterek) használata Pelda50

Most megoldjuk az előző feladatot mégegyszer azzal a különbséggel, hogy az adatokat nem inicializációs állományokban

fogjuk tárolni, hanem a Windows rendszerleíró adatbázisában. Programunk ekkor így néz ki:

```
…

uses
  Windows, Messages, SysUtils, … , Registry;

…

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

procedure TForm1.FormCreate(Sender: TObject);
var
  Reg: TRegistry;
begin
  // letrehozzunk egy TRegistry tipusu objektumot
  Reg := TRegistry.Create(KEY_READ);
  try
    // beallitjuk a fo kulcsot
    Reg.RootKey := HKEY_CURRENT_USER;
    // megprobaljuk megnyitni a mi alkalmazasunk
    // Edit kulcsat
    if Reg.OpenKey('\Software\Mi alkalmazasunk\Edit',
                                          False) then
      begin
        // ha sikerult, beolvassuk a szoveget
        Edit1.Text := Reg.ReadString('Text');
      end;
    // megprobaljuk megnyitni a mi alkalmazasunk
    // Form kulcsat
    if Reg.OpenKey('\Software\Mi alkalmazasunk\Form',
                                          False) then
      begin
        // ha sikerult, beolvassuk az ablak mereteit
        Top := Reg.ReadInteger('Top');
        Left := Reg.ReadInteger('Left');
        Width := Reg.ReadInteger('Width');
        Height := Reg.ReadInteger('Height');
```



```
        if Reg.ReadBool('Maximized') then
          WindowState := wsMaximized
        else
          WindowState := wsNormal;
      end;
  finally
    // felszabaditjuk az objektumot a memoriabol
    Reg.Free;
  end;
end;

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
var
  Reg: TRegistry;
begin
  // letrehozzunk egy TRegistry tipusu objektumot
  Reg := TRegistry.Create(KEY_WRITE);
  try
    // beallitjuk a fo kulcsot
    Reg.RootKey := HKEY_CURRENT_USER;
    // megprobaljuk megnyitni a mi alkalmazasunk
    // Edit kulcsat
    if Reg.OpenKey('\Software\Mi alkalmazasunk\Edit',
                                          True) then
      begin
        // ha sikerult, beirjuk a szoveget
        Reg.WriteString('Text',Edit1.Text);
      end;
    // megprobaljuk megnyitni a mi alkalmazasunk
    // Form kulcsat
    if Reg.OpenKey('\Software\Mi alkalmazasunk\Form',
                                          True) then
      begin
        // ha sikerult, beirjuk az ablak mereteit
        Reg.WriteInteger('Top',Top);
        Reg.WriteInteger('Left',Left);
        Reg.WriteInteger('Width',Width);
        Reg.WriteInteger('Height',Height);
        Reg.WriteBool('Maximized',
                           WindowState=wsMaximized);
      end;
  finally
    // felszabaditjuk az objektumot a memoriabol
    Reg.Free;
```

```
  end;
end;

…
```

Ahhoz, hogy dolgozhassunk a rendszerleíró adatbázissal, mindenekelőtt a programunk uses részét ki kell egészítenünk a **Registry** unittal.

Majd a **TRegistry.Create** metódus segítségével létrehozunk egy objektumot a **TRegistry** osztályból. Paraméterként megadjuk, hogy olvasni vagy írni akarunk-e a rendszerleíró adatbázisba. Olvasáshoz **KEY_READ**, íráshoz **KEY_WRITE** konstanst adjuk meg paraméterként.

A **RootKey** tulajdonság segítségével megadjuk a fő kulcsot, melyhez viszonyulnak majd a továbbiakban megadott utak. Itt általában a **HKEY_CURRENT_USER** vagy **HKEY_LOCAL_MACHINE** kulcsokat szokás megadni.

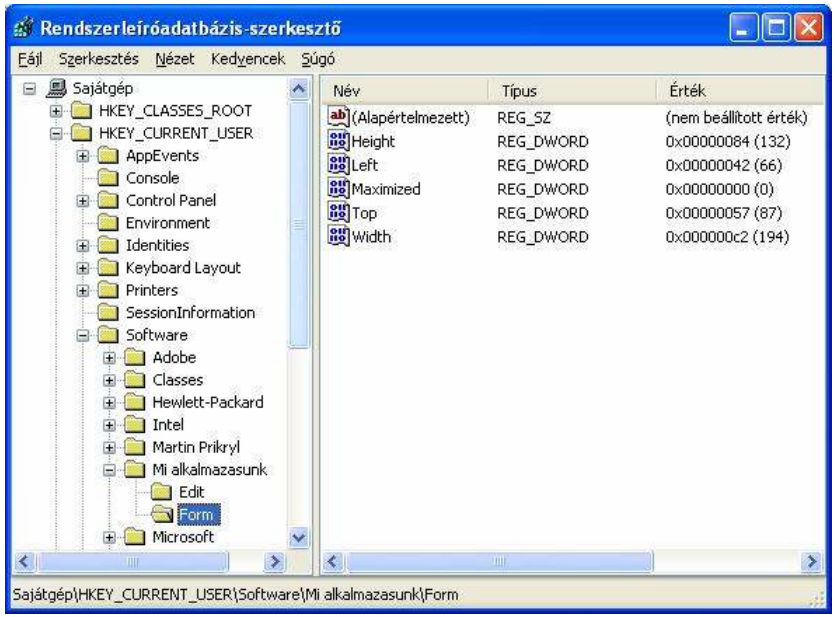
Az **OpenKey** metódus segítségével megnyitjuk azt a kulcsot, melyből szeretnénk az egyes adatokat beolvasni. A mi alkalmazásunkban két kulcsot használunk: „\ Software \ Mi alkalmazasunk \ Edit” melyben a szövegdoboz tartalmát tároljuk, és „\ Software \ Mi alkalmazasunk \ Form” melyben az alkalmazásunk ablakának méretét és pozícióját tároljuk.

Az **OpenKey** második paramétere megadja, hogy a megadott kulcsot az objektum létre hozhatja-e, ha az még nem létezik. Olvasáskor nem szeretnénk ha létrehozná, ezért ekkor itt false paramétert adunk meg, íráskor viszont true paramétert, mivel a kulcsot létre akarjuk hozni, ha még nem létezik.

Adatok írásához és olvasásához az inicializációs fájloknál használt metódusokhoz hasonlóan **WriteString**, **WriteInteger**, **WriteBool** illetve **ReadString**, **ReadInteger** és **ReadBool** függvényeket használuk.

Végül ha már nincs szükségünk az objektumra, felszabadítjuk azt a **Free** metódus segítségével.

Az alábbi ábrán láthatjuk hogyan hozta létre az alkalmazásunk a kulcsokat a rendszerleíró adatbázisban és hogyan helyezte el benne az egyes adatokat:



Gyakorlatok

1. Készítsük el a 3. fejezetben leírt „Első programunkat”. Gyak01
2. Próbáljunk meg különféle komponenseket elhelyezni az ablakunkban, majd futtassuk le a programot és figyeljük, hogyan jelennek meg, ill. milyen értékeket tudunk megadni nekik. Gyak02
3. Hozzunk létre egy alkalmazást, amelyen két gomb lesz (Kiírás, Kilépés) és egy címke. Az egyik megnyomásakor átírja a címke feliratát (ezt a programkódban: **Label1.Caption := ‘Uj felirat’;** formában adhatjuk meg), a másik gomb megnyomására kilép a programból. Gyak03
4. Készítsünk programot, amely egy címkét és egy nyomógombot tartalmaz (Sorsolás). A gomb megnyomásakor a számítógép a címke feliratába írjon ki 5 véletlenszerű lottószámot 1-től 90-ig (ilyen véletlenszámokat a **random(90)+1** függvénnyel tudunk generálni, majd a számot az **IntToStr()** függvénnyel tudjuk szöveggé alakítani). Ne felejtsük el előtte beállítani a véletlenszám generátort (**randomize;**), hogy minden indítás után ne kapjuk ugyanazokat a számokat. A program tervezésekor állítsuk be az Objektum felügyelőben, hogy a címke betűmérete nagyobb legyen (ezt a címke **Font.Size** tulajdonságával tehetjük meg). Gyak04
5. Próbáljunk meg készíteni egy alkalmazást, amelyen három gomb (Páros, Páratlan, Fibonacci) és egy címke szerepel.

Az első gomb megnyomásakor a címke feliratát átírja az első 10 páros számra (2, 4, 6, …), a második megnyomásakor az első 10 páratlan számra (1, 3, 5, …), a harmadik megnyomásakor kiírja az első 10 Fibonacci számot (1, 1, 2, 3, 5, 8, … - mindegyik szám az előző kettő összege). A számokat ciklus segítségével próbáljuk meg generálni. Gyak05

A következő néhány megoldott feladatban (6.-21.) a tulajdonságokat a könnyebb megértés végett, ha lehet, nem az Objektum felügyelőben, hanem az ablak (form1) **OnCreate** eseményében állítunk be. Így a forráskódból érthetőbb lesz, hogy melyik tulajdonságokat állítottuk át. Természetesen a saját program elkészítésekor ezeket ugyanúgy beállíthatjuk az Objektum felügyelőben is, ahogy eddig tettük az **OnCreate** esemény helyett. A 22. feladattól, amikor már remélhetőleg természetes lesz számunkra, hogy a komponensek melyik tulajdonságát kell beállítanunk az Objektum felügyelőben, a példaprogramoknál is visszatérünk a komponensek alapvető kezdeti tulajdonságainak az Objektum felügyelőben való beállításához.

6. Jelenjen meg a képernyőn két nyomógomb Belevágok! és *Kilépés* felirattal. A belevágok gombra való kattintás után jelenjen meg az *Üdvözöllek a programozás világában!* üzenet. Gyak06 (tulajdonság: **Caption**, esemény: **OnClick**, metódus: **Form1.Close**)
7. Jelenjen meg a képernyőn egy gomb *Kilép* felirattal. Ha a felhasználó rákattint, jelenjen meg egy üzenet

Meggondolta? kérdéssel. Majd ha „leokézza”, egy másik üzenet *Biztos benne?* kérdéssel, stb. Legalább ötször egymás után. Gyak07 (ismétlés Turbo Pascalból: egy **i** változó deklarálása a unit implementation részében, **case** elágazás használata)

8. Bővítsük ki az előző feladatot úgy, hogy az ablak helye minden gombnyomás után máshol legyen a képernyőn véletlenszerűen kiválasztva. Gyak08 (új tulajdonságok: **Left**, **Top**, **Width**, **Height**, **Screen.Width**, **Screen.Height**, események: **OnCreate**, ismétlés Turbo Pascalból: **random**, **randomize**)

9. Próbáljuk meg a programot úgy átírni, hogy ha a felhasználó máshogy (X-szel a jobb felső sarokban, ALT+F4-gyel, stb.) akarja bezárni az alkalmazást, akkor se tudja és jelenjen meg neki ebben az esetben az *Így nem fog menni, csak a gombbal!* felirat. Gyak09 (új esemény: **Form1.OnCloseQuery**, ennek **CanClose** paramétere)

10. A képernyőn jelenjen meg egy adatlap (ábra). Ha az *Edit1* beviteli mezőbe beírjuk a nevünket, akkor a *Label3* címkébe kerüljön be a bevitt adat! Gyak10 (új tulajdonságok: **Edit1.Text**, **Font.Style** halmaz)

11. Bővítsük az előző feladatot egy újabb kérdéssel (*Életkora:*), ami csak akkor jelenjen meg, amikor a felhasználó válaszolt az előző kérdésre. Gyak11 (új tulajdonság: **Visible**)

12. Jelenjen meg a képernyőn két beviteli mező és egy *Csere* feliratú gomb. A gombra kattintáskor a két beviteli mező tartalma cserélődjön meg. Gyak12

13. Zöldséges standunkon háromféle terméket árulunk: burgonyát, répát és káposztát. Egységárukat egy-egy címke jeleníti meg, a vásárolt mennyiséget egy-egy beviteli mezőbe írjuk. Egy gomb megnyomása után számítsuk ki és jelenítsük meg a fizetendő összeget! Gyak13 (új tulajdonság: **Font.Size**, függvények: **StrToFloat**, **FloatToStr**, **Round**)

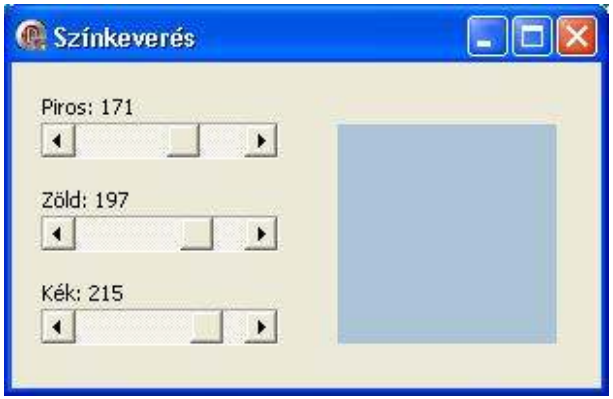
14. A programablak bal felső sarkában jelenjen meg egy nyomógomb. Ha a felhasználó rákattint, menjen a gomb a jobb felső sarokba, majd a jobb alsó, bal alsó, végül újra a bal felső sarokba, stb. Gyak14 (új tulajdonságok: **Form1.ClientWidth**, **Form1.ClientHeight**)

15. Találjuk ki a gép által gondolt egész számot tippeléssel, ha a gép minden tipp után megmondja, hogy az kicsi vagy nagy! Gyak15 (új tulajdonságok: **Button1.Default, Button1.Cancel**, új metódus: **Edit1.SelectAll**)

16. Készítsünk programot elektronikus pizza rendeléshez! A kért összetevőket jelölőnégyzetekkel lehessen megadni. A program ezek alapján automatikusan a jelölés közben jelenítse meg a pizza árát! Gyak16 (új tulajdonságok: **CheckBox1.Checked**, saját eljárás létrehozása, az összes CheckBox OnClick eseményére ugyannak az eljárásnak a megadása, mint az CheckBox1-nek)

17. Készítsünk szoftvert kávé automatához! Rádiógombokkal lehessen megadni az italt (kávé, tea, kakaó), jelölőnégyzetekkel a hozzávalókat (citrom, cukor, tej, tejszín). A szoftver számolja ki és jelenítse meg a fizetendő összeget! Teához ne lehessen tejszínt, kávéhoz citromot, kakaóhoz se citromot, se tejszínt kérni! (ábra) Gyak17 (új tulajdonságok: **Enabled**, **RadioButton1.Checked**)

18. Színkeverés RGB színmodell alapján. A képernyőn jelenjen meg három görgetősáv, amely az RGB színmodell három alapszínét állítja be 0 és 255 között. A kikevert szín egy címke hátterében jelenjen meg! (ábra) Gyak18 (új tulajdonságok: **ScrollBar1.Min**, **ScrollBar1.Max**, **ScrollBar1.Position**, **Form1.DoubleBuffered**, új esemény: **OnChange**, új Windows API függvény: **RGB**)



19. Készítsünk csúszkás számológépet! A kért számot egy-egy vízszintes görgetősáv tologatásával lehessen bevinni, majd a megfelelő nyomógombra (feliratuk: *Összeadás, Kivonás*, *Szorzás, Osztás*) való kattintáskor jelenjen meg egy címkében az eredmény! Gyak19

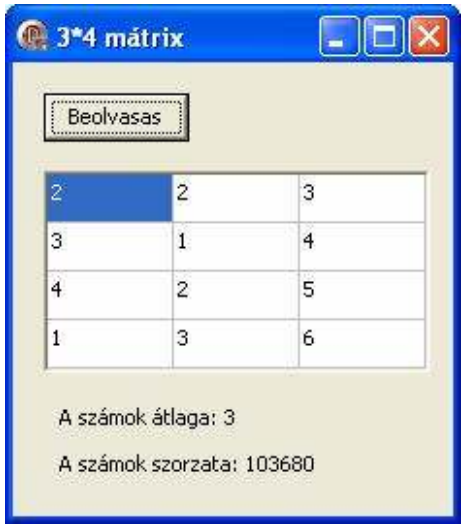
20. Készítsünk programot, amely egy ListBox-ot tartalmaz. Ha rákattintunk a form-ra egérrel, duplán rákattintunk, vagy megnyomunk egy billentyűt, írassuk ki a ListBox-ba az OnMouseDown, OnClick, OnMouseUp, OnDblClick, OnKeyDown, OnKeyPress, OnKeyUp események neveit olyan sorrendben, ahogy bekövetkeznek. Gyak20 (tulajdonság: **Form1.KeyPreview**, metódus: **ListBox1.Items.Add**)

21. Verem demonstrálása: készítsünk egy alkalmazást, amely tartalmaz egy listát és egy beviteli mezőt. A beviteli mező adata a Push gomb hatására kerüljön a lista tetejére, míg a Pop gomb hatására a lista felső eleme kerüljön a beviteli mezőbe, és törlődjön a listáról (ábra). A lista legfeljebb 10

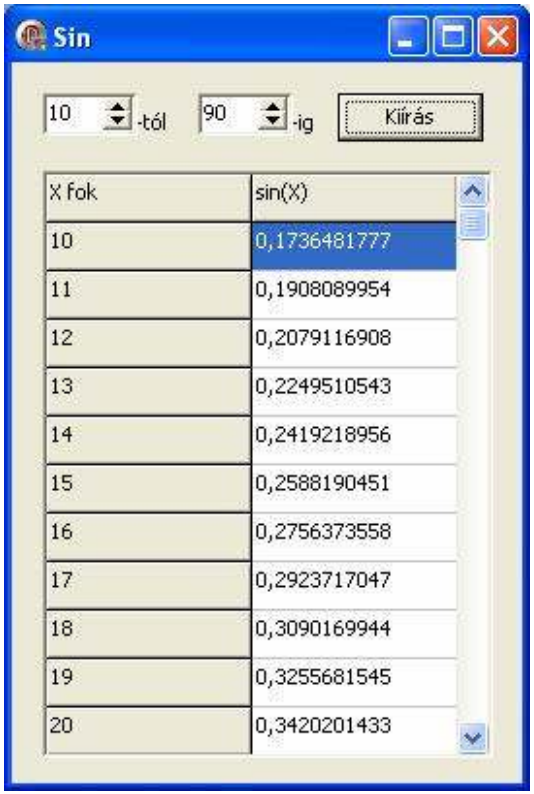
elemű lehet. Ha a lista tele van (Full) vagy üres (Empty), akkor a megfelelő gomb hatására kapjunk hibajelzést (üzenet ablak)! Gyak21 (új tulajdonság: **ListBox1.Items[0]**, új metódusok: **ListBox1.Items.Insert**, **ListBox1.Count**, **ListBox1.Items.Delete**)

22. Sor bemutatása: a képernyőn jelenjen meg egy lista és egy beviteli mező. A Push gomb hatására a beviteli mező tartalma kerüljön a lista tetejére, a Pop gomb hatására a lista alsó eleme kerüljön a beviteli mezőbe. A lista legfeljebb 10 elemű lehet. Ha a lista tele van vagy üres, akkor a megfelelő gomb generáljon hibajelzést! Gyak22

23. Olvassunk be az InputBox függvény segítségével egy 3*4-es mátrixot, melyet egy StringGrid komponensbe jelenítsünk meg. Számoljuk ki az elemek átlagát és szorzatát. Gyak23



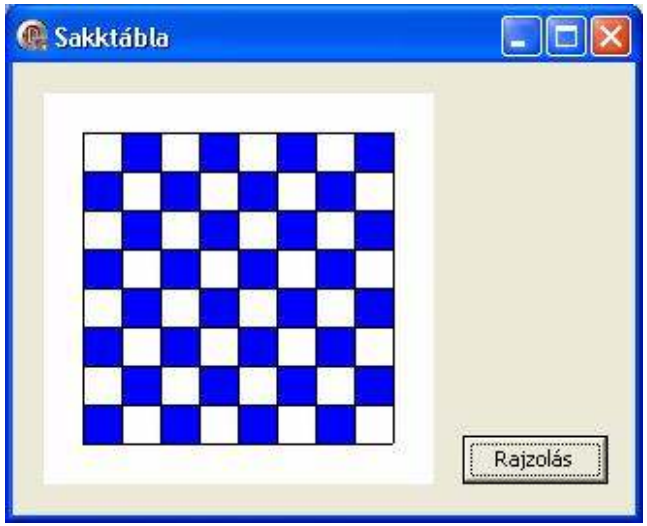
24. Írjuk ki a Sin függvény értékeit táblázatosan egy StringGrid komponensbe előre megadott intervallumban fokonként. Ne engedjük, hogy az intervallum alsó értéke nagyobb legyen, mint a felső. Gyak24



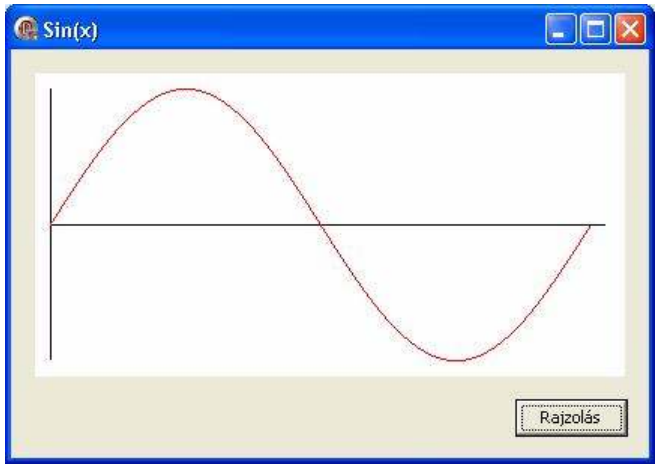
25. Olvassunk be egy 3*3-as mátrixot, majd ellenőrizzük, hogy a mátrix bűvös négyzet-e, azaz sorainak, oszlopainak és átlóinak összege azonos-e (az eredményt egy MessageBox segítségével jelenítsük meg). Az alábbi példában szereplő mátrix bűvös négyzet. Gyak25

26. Programunk írja ki mely billentyűt kell lenyomni, és írja ki a megtalálás idejét. Folyamatosan értékelje sebességünket (átlagos sebesség egy billentyű lenyomására). Gyak26

27. Készítsünk programot, amely egy nyomógomb megnyomásakor kirajzol egy sakktáblát egy image komponensre. Gyak27



28. Készítsünk egy alkalmazást, amely egy nyomógomb megnyomásakor kirajzolja egy image komponensbe a sin(x) függvény grafikonját. Gyak28



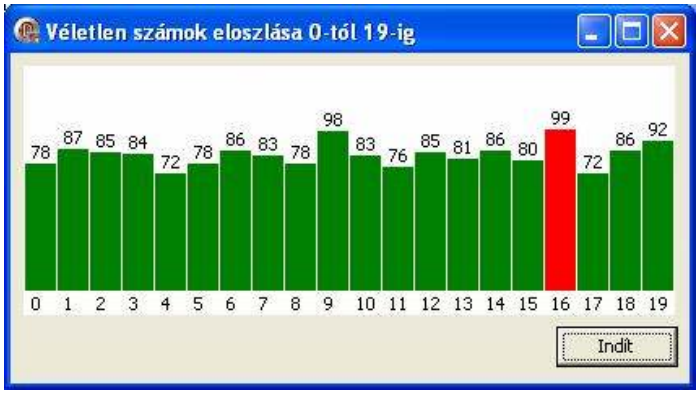
29. Készítsünk egy alkalmazást, amely tartalmaz egy nagyobb méretű üres Image komponenst és négy kisebb Image komponenst, melyekben különböző háttérmintákat jelenítünk meg. Ha valamelyik háttérmintára rákattintunk egérrel, a program töltse ki a megadott mintával a nagyobb Image komponenst. Gyak29

30. Készítsünk egy "pecsételő programot". A program tartalmazzon néhány kép kicsinyített változatát. Ha valamelyik képre rákattintunk egérrel, majd a rajzlapra kattintunk (nagyobb méretű Image komponens), akkor minden egyes kattintás helyére a program "pecsételje oda" a kiválasztott rajzot. A rajzot úgy rakjuk ki a rajzlapra, hogy a kattintás helye (koordinátái) a kirajzolandó kép közepén legyen. Az alkalmazásunk tartalmazzon még egy nyomógombot is, mellyel letörölhetjük a rajzlapot. Gyak30

31. Készítsünk alkalmazást, amely szemlélteti a véletlen számok eloszlását. A számítógép 0 és 19 közötti véletlen számokat generáljon ki és számolja az egyes számok előfordulását, melyet oszlopokkal szemléltessen. Mindegyik oszlop fölé írja oda, hogy mennyiszer volt az adott szám kigenerálva. Amelyik szám(ok) az adott pillanatban a legtöbbször fordulnak elő, azokat zöld oszlop helyett mindig pirossal szemléltessük. A számok generálását egy nyomógomb segítségével lehessen elindítani. Ha újra megnyomjuk a nyomógombot, a számok generálása elölről kezdődjön. A program tehát a nyomógomb megnyomása után minden szám oszlopának magasságát beállítja nullára, majd:

- Kigenerál egy 0-19 közötti véletlen számot.
- Az adott szám oszlopának magasságát megnöveli egy pixellel és fölé kiír eggyel nagyobb számot.
- Figyeli, melyik számok előfordulása a legnagyobb, ezeket piros oszloppal szemlélteti, a többit zölddel.
- Kigenerálja a következő véletlen számot…

A program a nyomógomb megnyomása után automatikusan működjön és növelje bizonyos időközönként (pl. 0,01 sec-ként) a kigenerált szám oszlopának magasságát mindaddig, amíg valamelyik nem éri el a 99-et. Ekkor a számok generálása álljon le. Gyak31



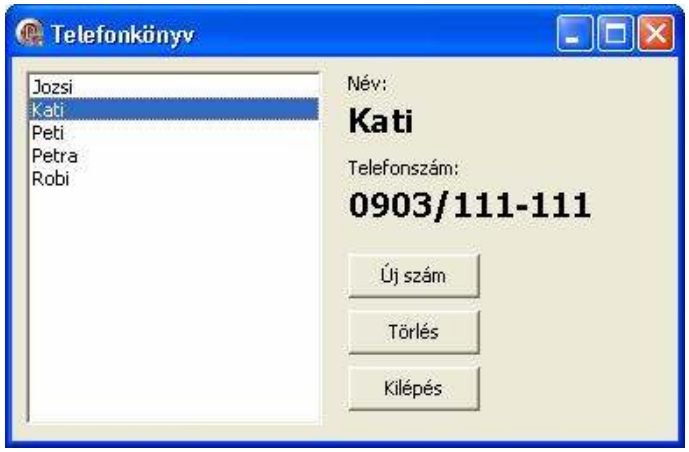
32. Készítsünk programot, amely tartalmazni fog egy Memo komponenst és három nyomógombot. Az első nyomógomb egy dialógusablak segítségével válasszon ki egy TXT fájlt, majd olvassa be a program a Memo komponensünkbe a fájl

tartalmát. A második nyomógomb mentse el a fájlt (dialógusablakkal lehessen megadni a fájl nevét és helyét), a harmadik nyomógomb segítségével lehessen megváltozatni a Memo komponens betűtípusát. Az alkalmazást bővítsük ki menüvel (MainMenu), ahonnan szintén elérhető legyen ez a három funkció. Gyak32

33. Készítsünk telefonkönyvet. Az alkalmazás tartalmazzon egy ListBox-ot, melyben nevek találhatók ABC sorrendben. Ha valamelyik névre rákattintunk (kijelöljük), a jobb oldalon jelenjen meg a név és a hozzá tartozó telefonszám.

    Az „**Új szám**” nyomógombra kattintáskor egy új (modális) ablakban kérjünk be egy nevet és egy telefonszámot, melyet helyezzünk el a névsorban a megfelelő helyre (úgy, hogy a nevek ABC sorrendben maradjanak). A „**Törlés**” gombra kattintáskor a kijelölt nevet töröljük a névsorból. Ilyenkor a jobb oldalon a törölt elem után következő (ha nincs akkor az előtte levő) név jelenjen meg (ha nincs előtte levő sem, akkor a jobb oldalon ne jelenjen meg semmilyen név és telefonszám).

    Az összes nevet és telefonszámot a programból való kilépéskor mentsük el egy külső állományba. A program indításakor olvassuk be ebből a fájlból a neveket. Gyak33



34. Készítsünk alkalmazást, amely megjeleníti és folyamatosan mutatja (frissíti) az aktuális időt. Gyak34

35. Készítsünk „csigák versenye” játékot. A csigák valamelyik gomb megnyomásával induljanak el. Mindegyik csiga véletlenszerű hellyel menjen jobbra mindaddig, amíg valamelyik nem éri el az ablak jobb szélét. Ha az a csiga nyert, amelyre tippeltünk, akkor a pontszámunk növekedjen 3-mal, különben csökkenjen 1-gyel. A nyertes csiga színét egy MessageBox segítségével írjuk ki, majd a csigák álljanak újra a rajtvonalra, és újból lehessen tippelni valamelyik nyomógomb megnyomásával! Gyak35

Melléklet:
Leggyakrabban használt változók

Egész számok típusai:

Típus	Értéktartomány	Helyigény
Shortint	- 128 .. 127	1 bájt
Byte	0 .. 255	1 bájt
Smallint	- 32 768 .. 32 767	2 bájt
Word	0 .. 65 535	2 bájt
Integer	- 2 147 483 648 .. 2 147 483 647	4 bájt
Longint	- 2 147 483 648 .. 2 147 483 647	4 bájt
Cardinal	0 .. 4 294 967 295	4 bájt
Longword	0 .. 4 294 967 295	4 bájt
Int64	$-2^{63} + 1 .. 2^{63}$	8 bájt

Valós számok típusai:

Típus	Értéktartomány	Pontosság	Helyigény
Single	$-1{,}5 \times 10^{45} .. 3{,}4 \times 10^{38}$	7 - 8	4 bájt
Real48	$-2{,}9 \times 10^{39} .. 1{,}7 \times 10^{38}$	11 - 12	6 bájt
Real	$-5{,}0 \times 10^{324} .. 1{,}7 \times 10^{308}$	15 - 16	8 bájt
Double	$-5{,}0 \times 10^{324} .. 1{,}7 \times 10^{308}$	15 - 16	8 bájt

Comp	$-2^{63} + 1 .. 2^{63}$	19 - 20	8 bájt
Currency	- 922337203685477,5808 .. 922337203685477,5807	19 - 20	8 bájt
Extended	$-3{,}6 \times 10^{4951} .. 1{,}1 \times 10^{4932}$	19 - 20	10 bájt

Egykarakteres szöveges változók:

Típus	Értéktartomány	Helyigény
Char	1 karakter	1 bájt
PChar	változó	változó

Többkarakteres szöveges változó:

Típus	Értéktartomány	Helyigény
String	felhasználó deklarálhatja (pl. String[50], String[255])	aktuális hossz + 1 bájt

Logikai változók:

Típus	Értéktartomány	Helyigény
Boolean	False, True	1 bájt
ByteBool	False, True	1 bájt
WordBool	False, True	2 bájt
LongBool	False, True	4 bájt

Melléklet:
Magyar - Angol - Szlovák szótár

Magyar	Angol	Szlovák
integrált fejlesztői környezet	integrated development environment (IDE)	integrované vývojové prostredie
menü	menu	hlavná ponuka
eszköztár	toolbar	panel nástrojov
ablak tervező	form designer	návrhár formuláru
elempaletta	tool palette	paleta komponent
objektum felügyelő	object inspector	object inspector
forráskód szerkesztő	code editor	editor programového kódu
tulajdonságok	properties	vlastnosti
események	events	udalosti
metódusok	methods	metódy
eseménykezelés	handle event	obsluha udalosti
vizuális komponenskönyvtár	visual component library (VCL)	knižnica vizuálnych komponentov
alkalmazás program interfész (alkalmazás-programozási felület)	application program interface (API)	rozhranie pre vývoj aplikácií

Irodalomjegyzék:

[1] Václav Kadlec: **Delphi Hotová řešení**, ISBN: 80-251-0017-0, Computer Press, Brno, 2003

[2] Steve Teixeira, Xavier Pacheco: **Mistrovství v Delphi 6**, ISBN: 80-7226-627-6, Computer Press, Praha, 2002

[3] Kuzmina Jekatyerina, Dr. Tamás Péter, Tóth Bertalan: **Programozzunk Delphi 7 rendszerben!**, ISBN: 963-618-307-4, ComputerBooks, Budapest, 2005

[4] Marco Cantú: **Delphi 7 Mesteri szinten**, **I. kötet**, ISBN: 963-9301-66-3, Kiskapu Kft., Budapest, 2003

[5] József Holczer, Csaba Farkas, Attila Takács: **Informatikai feladatgyűjtemény**, ISBN: 963-206-6391, Jedlik Oktatási Stúdió, Budapest, 2003