

Mgr. Végh László

Programozás Delphi-ben 2

Komárom, 2006. április 13.

© Mgr. Végh László, 2006
<http://www.prog.ide.sk>

Tartalom

1. Objektum orientált programozás.....	1
1.1. Az OOP alapelvei	2
1.2. Adatrejtés	3
1.3. Virtuális mező (property)	7
1.4. Constructor, destructor.....	8
1.5. Öröklés	12
1.6. Virtuális metódus (virtual).....	13
1.7. Dinamikus metódus (dynamic).....	20
1.8. Absztrakt metódus (abstract)	23
1.9. Típuskompatibilitás.....	25
1.10. Típusellenőrzés (is operátor)	26
1.11. Típuskonverzió (as operátor)	27
1.12. Interface-k (interface)	28
1.13. Osztály-változók	31
1.14. Osztály-metódusok.....	35
1.15. Hivatkozás a példányra a self azonosítóval	36
2. OOP a gyakorlatban.....	37
2.1. Pattogó labdák	37
2.2. Úszkáló alakzatok (kör, téglalap, háromszög)	46
3. DLL-ek használata és létrehozásuk	55
3.1. DLL készítése.....	56
3.2. A DLL felhasználása alkalmazásunkban	60
3.3. Statikus és dinamikus importálás.....	63
3.4. A DLL és a memória.....	68
3.5. Form tárolása DLL-ben	68
3.6. A DLL (és a benne tárolt form) felhasználása alkalmazás készítésekor Delphi-ben.....	71
3.7. A DLL (és a benne tárolt form) felhasználása a MS Excel makró nyelvében (Visual Basic).....	73
3.8. Erőforrások tárolása DLL-ben	73
3.9. DLL-ben tárolt erőforrások felhasználása	76
4. Párhuzamos programozás, szálak	79
4.1. TThread osztály.....	80
4.2. Szakaszok párhuzamos kirajzolása	83
4.3. Szálak szinkronizálása – várakozás egy másik programszállra	87
4.4. Programszálak prioritása	91
4.5. Többszálú MDI alkalmazás	97

5. OLE technológia.....	104
5.1. A Delphi és az OLE	105
5.2. Első OLE-t használó alkalmazásunk.....	106
5.3. Az OleContainer tulajdonságai.....	112
5.4. Kulcsszavak lekérdezése és végrehajtása	114
5.5. OLE objektum beolvasása és mentése.....	116
5.6. Menük összekapcsolása	118
5.7. Saját Word, Excel, Paint,	122
6. OLE Automation.....	126
Gyakorlatok:.....	127
Irodalomjegyzék:	132

1. Objektum orientált programozás

Osztály – egy felhasználó által készített típus (pl. TElso).

Objektum – az osztályból hozzuk létre, az osztály egy példánya (pl. ElsoObj).

Attribútum – az osztály egy mezője, melyben adatot tárolhatunk, miután az osztályból egy objektumot készítettünk (pl. x,y).

Metódus – az osztály egy eljárása, mely az adott osztály attribútumaival végez valamilyen műveletet. (pl. Hozzaad)

Példányosítás – egy osztályból egy konkrét objektum készítése (pl. ElsoObj := TElso.Create;).

Inicializálás – az objektum-attribútumok kezdőértékeinek beállítása (pl. ElsoObj.Inicialas;).

```

type TElso = class
    x,y: integer;
    procedure Hozzaad(xp,yp:integer);
    procedure Inicialas;
end;

```

...

```

procedure TElso.Hozzaad;
begin
    x := x + xp;
    y := y + yp;
end;

```

```

procedure TElso.Init;
begin
    x := 0;
    y := 0;
end;

```

...

```
var ElsoObj: TElso;  
begin  
    ElsoObj := TElso.Create;  
    ElsoObj.Inicialas;  
    ...  
    ElsoObj.Free;  
end;
```

1.1. Az OOP alapelvei

Egységbezárás (encapsulation)

Az adatokat és a hozzájuk tartozó eljárásokat egyetlen egységben (osztályban) kezeljük. Az osztály adatmezői tárolják az adatokat, a metódusok kommunikálnak a külvilággal (az osztály adatmezőit csak a metódusokon keresztül változtathatjuk meg).

Öröklés (inheritance)

Az osztály továbbfejlesztése. Ennek során a származtatott osztály öröklí az ősosztálytól az összes attribútumot, metódust. Ezeket azonban újakkal is kibővíthetjük, ill. bizonyos szabályok mellett az örökölt metódusokat (metódusok törzsét) is megváltoztathatjuk.

```
type TMasodik = class(TElso)  
    ...  
end;
```

TElso – az ősosztály (szülő).

TMasodik – az új, származtatott osztály (gyerek).

Egy ősből több származtatott osztályt is létrehozhatunk. Minden származtatott osztálynak csak egy őse lehet.

Sokalakúság (polymorphism)

Ugyanarra a metódusra a különböző objektumok különbözőképpen reagálnak. A származtatás során az ősosztályok metódusai képesek legyenek az új, átdefiniált metódusok használatára újraírás nélkül is.

Ezt virtuális (vagy dinamikus) metódusokkal érhetjük el.

```
type TElso = class  
    ...  
    procedure Akarmi; virtual;  
end;  
  
TMasodik = class(TElso)  
    ...  
    procedure Akarmi; override;  
end;
```

1.2. Adatrejtés

• Public

Ide kerülnek olyan attribútumok, melyek nem igényelnek speciális szabályozást. Az ilyen attribútumok értékeinek megváltoztatása nem okozhat problémát az objektum működésében.

Az itt felsorolt metódusokat a külvilág meghívhatja.

- **Private**

Az itt felsorolt attribútumokhoz a külvilág nem férhet hozzá. Ide írjuk általában azokat a változókat, melyek értékeit szabályozni akarjuk. Továbbá ide kerülnek a segégváltozók is.

Az itt szereplő metódusokat a külvilág nem hívhatja meg, csak az adott osztály metódusaiból érhetők el.

- **Protected**

Olyan attribútumok, melyekhez a külvilág nem férhet hozzá, de a leszármazott osztályok metódusai hozzáférhetnek.

Az itt feltüntetett metódusokat a külvilág nem hívhatja meg, de a leszármaztatott osztályok metódusaiból meghívhatók.

```
type TSzemely = class
    private
        nev: string;
        etekor: integer;
    public
        procedure Valami;
end;

...

var
    Peti: TSzemely;
begin
    ...
    Peti.etekor := 18;    { ez NEM mukodik !!! }
    ...
end;
```

Adatrejtés előnyei:

A mezőkhöz a hozzáférés szabályozható.

Adatrejtés hátrányai:

Nem szabályozható, hogy csak az olvasást, vagy csak az írást engedélyezzük, illetve tiltjuk. Csak egy időben lehet mindkét elérést engedélyezni / tiltani.

Adatok nem elrejtésének a hátránya:

```
type TSzemely = class
    public
        etekor: integer;
    ...
end;

...

begin
    ...
    Peti.etekor := -400;    { HIBÁS érték }
    ...
end;
```

Adatrejtés hátrányai:

```
type TSzemely = class
    private
        etekor: integer;
    ...
end;

...
```

```

begin
  ...
  Peti.eletkor := -400;    { ez nem működik }
  writeln(Peti.eletkor);  { de EZ SEM megy }
  ...
end;

```

Megoldás a Set és Get metódusok bevezetésével:

```

type
  TSzemely = class
    private
      eletkor: integer;
    public
      procedure SetEletkor(e:integer);
      function GetEletkor: integer;
    ...
  end;

...

procedure TSzemely.SetEletkor;
begin
  if (e>0) and (e<100) then eletkor := e
    else ... { hiba jelzése}
end;

function TSzemely.GetEletkor:integer;
begin
  Result := eletkor;
end;

...

begin
  ...
  Peti.SetEletkor(-400);    { hibajelzés }
  Peti.SetEletkor(18);     { rendben }
  writeln(Peti.GetEletkor); { rendben }
  ...
end;

```

1.3. Virtuális mező (property)

A virtuális mezők (property) segítségével jobban megoldhatók az előző problémák az adatrejtésnél:

```

type
  TSzemely = class
    private
      fEletkor: integer;
    public
      procedure SetEletkor(e:integer);
      property Eletkor:integer
        read fEletkor
        write SetEletkor;
    ...
  end;

...

procedure TSzemely.SetEletkor;
begin
  if (e>0) and (e<100) then fEletkor := e
    else ... { hiba jelzése}
end;

...

begin
  ...
  Peti.Eletkor := -400;    { hibajelzés }
  Peti.Eletkor := 18;     { rendben }
  writeln(Peti.Eletkor);  { rendben }
  ...
end;

```

A **read** után meg kell adnunk, hogy ha a virtuális mezőt (Eletkor) olvassák, mit tegyen a program. Itt két lehetőségünk van:

- a read után írhatunk egy ugyanolyan típusú fizikai mezőt,
- a read után írhatunk egy paraméter nélküli függvényt, melynek visszatérési értéke megegyezik a virtuális mező típusával.

A **write** után szintén meg kell adnunk mi legyen a teendő, ha a virtuális mezőbe (Eletkor) új értéket akarnak belerakni. Itt is két lehetőségünk van:

- a write után megadhatunk egy ugyanolyan típusú fizikai mezőt,
- a write után beírhatunk egy eljárást, melynek egyetlen paramétere van és ez ugyanolyan típusú, mint a virtuális mező típusa.

A **property** kulcsszónál nem kötelező megadnunk mindkét módot (read, write). Ha csak az egyiket adjuk meg, akkor a virtuális mező csak olvasható ill. csak írható lesz.

Továbbá nem kötelező, hogy a virtuális mező mögött egy fizikai mező álljon. Lehet az olvasás és az írás is egy függvény ill. eljárás segítségével megoldva, amelyek valami más alapján adnak vissza / állítanak be értékeket.

1.4. Constructor, destructor

A **konstruktor (constructor)** speciális metódus, amely feladata az objektum létrehozása az osztályból és annak alaphelyzetbe állítása (mezők kezdőértékeinek beállítása).

```
type
    TSzemely = class
        private
            fEletkor: integer;
        public
            constructor Create;
            ...
    end;

...

constructor TSzemely.Create;
begin
    fEletkor := 0;
    ...
end;

...

var
    Peti: TSzemely;
begin
    Peti := TSzemely.Create;
    ...
end;
```

A konstruktor neve általában **Create** vagy **Init**, s bár tetszőleges név adható neki, jó betartani ezt a szabályt. Egy osztályban több konstruktor is létezhet, de ilyenkor jellemző, hogy ezeknek más-más a paramétere.

```

type
  TSzemely = class
    private
      fEletkor: integer;
    public
      constructor Create;
      constructor Init(kor:integer);
      ...
  end;

...

constructor TSzemely.Create;
begin
  fEletkor := 0;
  ...
end;

constructor TSzemely.Init;
begin
  fEletkor := kor;
  ...
end;

...

var
  Peti, Jani: TSzemely;
begin
  Peti := TSzemely.Create;
  Jani := TSzemely.Init(20);
  ...
end;

```

A **destruktor (destructor)** szintén egy speciális feladatú metódus, amely feladata az objektum megszüntetése és a memória felszabadítása. Itt még lehetőségünk van a lefoglalt erőforrások (file-ok, memória, hálózati kapcsolatok, ...) felszabadítására.

A destruktor neve általában **Destroy**. A destruktor meghívhatjuk ezzel a névvel, vagy meghívhatjuk a **Free** metódus segítségével, amely leellenőrzi, hogy az objektum létezik, majd meghívja a Destroy nevű destruktort.

```

type
  TSzemely = class
    private
      fEletkor: integer;
    public
      destructor Destroy; override;
      ...
  end;

...

constructor TSzemely.Destroy;
begin
  ...
end;

...

var
  Peti: TSzemely;
begin
  Peti := TSzemely.Create;
  ...
  Peti.Free;    { vagy: Peti.Destroy; }
end;

```

Fontos, hogy a programunkban ha egy objektumot már nem használunk, akkor meghívjuk a **Free** metódust vagy közvetlenül a destruktort mielőtt kilépnénk a programból. Ha ezt nem tesszük meg, beragadhat a memóriába a lefoglalt terület és az erőforrások (file, hálózat, ...) foglalta maradhatnak.

1.5. Öröklés

Az osztály fejlesztését nem kell nulláról kezdenünk, mivel az osztály az őstől öröklí az összes mezőt, metódust. Ehhez mindössze az ősoosztály nevét kell feltüntetnünk az új (származtatott) osztály deklarációsoroknál.

Ősnek legjobb azt az osztályt választani, amely legközelebb áll a mi osztályunkhoz, amelyből a legkevesebb módosítással létre tudjuk hozni az új osztályt.

Ha az osztály deklarációsoroknál nem választunk őst, akkor alapértelmezésből a **TObject** osztály lesz az ősünk. Mindenkinek a közös őse a TObject. Ebben alaptól van néhány hasznos metódus, pl. Destroy nevű destruktork, Free metódus.

Ügyeljünk arra, hogy a származtatott osztályban ne vezessünk be ugyanolyan nevű mezőket, mint amilyen már az ősben szerepel, ezek ugyanis elfednék az örökölteket.

Ugyanolyan nevű metódus bevezetése lehetséges, amennyiben más a paraméterezése. Ekkor mindkét metódus elérhető lesz és az aktuális paraméterezés dönti el melyiket hívjuk éppen meg. A származtatott objektumban az ugyanolyan nevű, de más paraméterű metódust **overload**; szóval kell feltüntetnünk különben elfedi az előtte (ősben) szereplő metódust (ilyenkor, ha az új metódus elfedi a régijt, az ősoosztály nem tudja használni az új metódust).

```
type
  TOs = class
    public
```

```
        function Akarmi:integer;
    end;

TSzarm = class(TOs)
    public
        function Akarmi(x:real):real;
                                overload;
    end;

...

function TOs.Akarmi:integer;
begin
    result := 10;
end;

function TSzarm.Akarmi(x:real):real;
begin
    result := x + 10.5;
end;

...

var
    p:TSzarm;
begin
    p := TSzarm.Create;
    writeln(p.Akarmi);           { eredmény: 10 }
    writeln(p.Akarmi(1));       { eredmény: 11.5 }
    p.free;
end;
```

1.6. Virtuális metódus (virtual)

Gyakran megtörténhet, hogy egy osztály metódusa használ egy másik metódust ugyanabban az osztályban. Mi történik, ha az ilyen osztályból leszármaztatunk egy új osztályt, és megváltoztatjuk azt a metódust, amelyet a másik használ? Például:


```

type TAOszt = class
    public
        function Elso:integer;
        function Masodik:integer;
    end;

    TBOszt = class(TAOszt)
    public
        function Elso:integer;
    end;

...

function TAOszt.Elso:integer;
begin
    Result := 1;
end;

function TAOszt.Madosik:integer;
begin
    Result := Elso + 1;
end;

function TBOszt.Elso:integer;
begin
    Result := 10;
end;

...

var
    pa:TAOszt;
    pb:TBOszt;
begin
    pa := TAOszt.Create;
    pb := TBOszt.Create;
    writeln(pa.Masodik); { Mennyit ír ki? 2-t }
    writeln(pb.Masodik); { Mennyit ír ki? 2-t }
    ...
end;

```

Mi lehetne erre a megoldás? Például leírhatjuk újra a Masodik függvényt is a származtatott osztályba:

```

type TAOszt = class
    public
        function Elso:integer;
        function Masodik:integer;
    end;

    TBOszt = class(TAOszt)
    public
        function Elso:integer;
        function Masodik:integer;
    end;

...

function TAOszt.Elso:integer;
begin
    Result := 1;
end;

function TAOszt.Madosik:integer;
begin
    Result := Elso + 1;
end;

function TBOszt.Elso:integer;
begin
    Result := 10;
end;

...

function TBOszt.Madosik:integer;
begin
    Result := Elso + 1;
end;

...

var
    pa:TAOszt;
    pb:TBOszt;
begin
    pa := TAOszt.Create;
    pb := TBOszt.Create;
    writeln(pa.Masodik); { Mennyit ír ki? 2-t }
    writeln(pb.Masodik); { Mennyit ír ki? 11-t }

```

```
...
end;
```

Jó megoldás ez? Ez működik, de **EZ NEM JÓ MEGOLDÁS**, mivel:

- Az ős osztály (TAOszt) függvényének programkódját nem biztos, hogy a TBOszt programozója ismeri.
- Ha ismeri, ez akkor is felesleges programkód másolás (copy-paste)
- A Masodik függvény kétszer is szerepel a lefordított EXE állományban, ezért ennek a hossza is megnagyobbodott.
- Ha a TAOszt osztály programozója változtat a Masodik függvény kódján, akkor újra az egészet másolni kell a másik osztályba is (copy-paste).

A jó megoldás: **virtuális metódus** használata. Ez egy olyan metódus, mint bármelyik másik metódus, a virtuális jelzőt a programozónak kell rátennie egy **virtual;** szó hozzáírásával az eljáráshoz az ősosztályban. Ez azt jelzi a fordítónak, hogy a metódust a származtatott osztályban valószínűleg felül fogják definiálni. A származtatott osztályban (és abból származtatott osztályokban is) a felüldefiniált eljáráshoz **override;** kulcsszót kell kitenni.

```
type
  TAOszt = class
    public
      function Elso:integer; virtual;
      function Masodik:integer;
    end;
```

```
TBOszt = class(TAOszt)
    public
      function Elso:integer; override;
    end;
```

```
...

function TAOszt.Elso:integer;
begin
  Result := 1;
end;

function TAOszt.Masodik:integer;
begin
  Result := Elso + 1;
end;

function TBOszt.Elso:integer;
begin
  Result := 10;
end;

...

var
  pa:TAOszt;
  pb:TBOszt;
begin
  pa := TAOszt.Create;
  pb := TBOszt.Create;
  writeln(pa.Masodik); { Mennyit ír ki? 2-t }
  writeln(pb.Masodik); { Mennyit ír ki? 11-t }
  ...
end;
```

Mit is jelent valójában ez a kulcsszó? Az ilyen, virtuális metódusokat a fordító másképp kezeli. A virtuális metódus hívásának pillanatában az objektum ismeretében megkeresi a legfejlettebb verziót az adott metódusból, és ez hívódik meg. Tehát az, hogy melyik

változatú „Elso” függvény kerül meghívásra, menet közben dől el. Ezt nevezik **sokalakúságnak (polymorphism)**.

Honnan tudja a program eldönteni, hogy melyik változatot kell meghívnia, melyik a legfrissebb változat? Ebben segít a **virtuális metódus tábla (VMT)**. Minden osztályhoz készül egy ilyen tábla, melyben szerepel a virtuális metódus neve és annak elérhetősége. A VMT-ban csak a virtuális metódusok szerepelnek. Pl.:

```
type TElso = class
    procedure A;
    procedure B;
    procedure C; virtual;
    procedure D; virtual;
end;

TMasodik = class (TElso)
    procedure A;
    procedure C; override;
    Procedure E; virtual;
end;
```

Ezekhez az osztályokhoz tartozó VMT-k:

TElso.VMT	TMasodik.VMT
metódus C = TElso.C	metódus C = TMasodik.C
metódus D = TElso.D	metódus D = TElso.D
	metódus E = TMasodik.E

A leszármazott osztály VMT-je úgy jön létre, hogy:

- lemásuljuk az ős VMT-jét,
- ha a leszármaztatott osztályban van OVERRIDE, akkor azt a sort kicseréljük,

- ha a leszármaztatott osztályban van VIRTUAL, akkor új sort adunk a táblázat végéhez.

Milyen lesz így a virtuális metódusok táblája?

- ugyanazon metódus a táblázatban (ősök és leszármazottak VMT-iban) mindig ugyanazon sorban fog szerepelni,
- a származtatott VMT-ja legalább annyi elemű lesz, mint az ősé, de ennél hosszabb is lehet (ha van benne új virtuális metódus),
- ahogy haladunk lefelé az öröklődési gráfban, úgy a VMT-k egyre hosszabbak és hosszabak lesznek.

A VMT előnye: gyors, mivel nem kell keresni a táblázatban, ha az ős VMT-ben egy metódus az N. helyen van, akkor minden leszármazottjánál is az N. sorban lesz.

A VMT hátránya: sok memóriát igényel (bejegyzésként 4 byte).

Minden osztályhoz 1 VMT tartozik (nem példányonként 1 drb), tehát ugyanazon osztályból létrehozott objektumnak (példánynak) ugyanaz a VMT-ja.

Az objektumhoz a VMT hozzárendelését a **konstruktor** végzi, ezt a kódrészletet a Delphi generálja bele a konstruktor lefordított kódjába.

A konstruktor nem lehet virtuális, mivel a konstruktor hívásakor még nem működik a mechanizmus (késői kötés).

A konstruktorban a VMT hozzárendelése az objektumhoz a konstruktor elején történik (a Delphiben), így a konstruktor belsejében már lehet virtuális metódust hívni.

A **destruktor** (Destroy metódus) mindig virtuális, ezért tudja a Free metódus meghívni mindig a megfelelő destruktort.

A **virtuális metódus paraméterezése és visszatérési értéke** nem változhat, mert akkor az ős metódusa nem lenne képes meghívni ezt a felüldefiniált változatot!

1.7. Dinamikus metódus (dynamic)

A dinamikus metódusok hasonlóan a virtuális metódusokhoz a polimorfizmus futás közbeni támogatására szolgálnak.

A Delphiben a dinamikus metódusokat hasonlóan tudjuk kialakítani, mint a virtuális metódusokat, csak itt a virtual szó helyett a **dynamic** szót használjuk. Pl.

```
type
  TAOszt = class
    public
      function Elso:integer; dynamic;
      function Masodik:integer;
    end;

  TBOszt = class(TAOszt)
    public
      function Elso:integer; override;
    end;
...
```

A működése a programban ugyanaz, mint ha virtuális metódust használtunk volna. A különbség a memóiafoglalásban és a gyorsaságban van.

A dinamikus metódusok a VMT-től egy kicsit eltérő, **Dinamikus Metódus Táblát (Dynamic Method Table)** használnak a legújabb verziójú metódus megtalálásához. Ennek felépítését is egy példán szemléltetjük:

```
type TElso = class
  procedure A;
  procedure B;
  procedure C; dynamic;
  procedure D; dynamic;
end;

TMasodik = class (TElso)
  procedure A;
  procedure C; override;
  Procedure E; dynamic;
end;
```

Ezekhez az osztályokhoz tartozó DMT-k:

TElso.DMT ős = nil	TMasodik.DMT ős = TElso.DMT
metódus C = TElso.C	metódus C = TMasodik.C
metódus D = TElso.D	metódus E = TMasodik.E

Mindegyik táblában szerepel az is, hogy melyik osztályból lett származtatva az új osztály. Ha nincs ilyen osztály, akkor ennek a mutatónak az értéke nil (ős = nil).

A származtatott osztály DMT-ja a következő képpen alakul ki:

- a DMT tartalmazza az ős DMT címét (ős = TElso.DMT),

- kezdetben nem másoljuk le az ős DMT-jét, hanem csak override esetén vagy dynamic esetén egy új sort szúrunk be,

A DMT tábla tehát csak változtatásokat tartalmaz. Így lehetséges, hogy valamelyik metódusra utaló bejegyzés nem is szerepel csak az ősből, vagy annak az ősében (amennyiben a leszármazott osztályokban nem lett fölüldefiniálva az override segítségével).

A fentiekből adódik, hogy futás közben a DMT-k között lehet, hogy keresni kell. Ha nincs az adott dinamikus metódusra utalás az objektumhoz tartozó DMT táblában, akkor meg kell nézni az ős DMT-jében, ha ott sincs, akkor annak az ősének a DMT-jében, stb.

A DMT előnye: kevesebb memóriát igényel, mivel nincs az ős egész DMT-je lemásolva, csak a változtatások vannak bejegyezve.

A DMT hátránya: a keresés miatt lassabb.

A dinamikus és virtuális metódusokat lehet keverni egy osztályon belül. Tehát egy osztályban lehetnek dinamikus és virtuális metódusok is. Azt, hogy valamelyik metódus milyen lesz, a metódus legelső bevezetésekor kell megadni a „virtual” vagy „dynamic” kulcsszó segítségével. A későbbiekben (leszármazottakban) a felüldefiniáláshoz mindig az „override”-t kell használni.

Melyik metódus legyen dinamikus és melyik virtuális?

A DMT akkor kerül kevesebb memóriába, ha ritkán van felüldefiniálva. Ha minden leszármazott osztályban felüldefiniáljuk, akkor több memóriát igényel! Másik fontos szempont a választáskor, hogy a DMT mindig lassabb!

Ezért **dinamikus legyen egy metódus**, ha számítunk arra, hogy ritkán lesz csak felüldefiniálva, továbbá úgy gondoljuk, hogy a

metódus ritkán kerül meghívásra, és ezért nem számít a sebességcsökkenés.

Virtuális legyen egy metódus minden más esetben, tehát ha sűrűn felüldefiniáljuk, vagy a metódust sűrűn hívjuk (pl. egy ciklusban) és fontos a sebesség.

1.8. Absztrakt metódus (abstract)

Nézzük meg, hogyan nézhetne ki egy osztály, amelyet azért hozunk létre, hogy később ebből származtassunk új osztályokat, pl. kör, négyzet, téglalap grafikus objektumok osztályait.

```
type TGrafObj = class
    private
        x,y: integer;
    public
        procedure Kirajzol; virtual;
        procedure Letorol; virtual;
        procedure Mozgat(ujx, uyy: integer);
end;

procedure TGrafObj.Mozgat;
begin
    Letorol;
    x := uyx;
    y := uyy;
    Kirajzol;
end;
```

A kérdés az, hogyan nézzen ki a „Kirajzol” és „Letöröl” metódusok programkódja. Konkrét esetben (tehát az ebből leszármaztatott osztályokban) tudjuk, hogy egy kör, négyzet, téglalap,

stb., de általános esetben ezt nem tudjuk. Ezért itt ez a következő módon nézhetne ki, mivel ez a leszármazottakban úgys felül lesz definiálva (kör, négyzet, téglalap, stb. kirajzolására):

```
procedure TGrafObj.Kirajzol;  
begin  
end;
```

De ez **NEM JÓ MEGOLDÁS**, mert amikor ebből továbbfejlesztünk pl. egy TKor osztályt, akkor a fordítóprogram nem fog szólni, hogy ezen két metódust kötelező nekünk felüldefiniálni! Így elfeledkezhetünk róla!

Erre az esetre szolgál az **absztrakt metódus**, amely egy olyan metódus, melynek törzsét nem kell megírunk, csak az osztály definiálásánál kell felsorolnunk:

```
type  
  TGrafObj = class  
    private  
      x,y: integer;  
    public  
      procedure Kirajzol; virtual; abstract;  
      procedure Letorol; virtual; abstract;  
      procedure Mozgat(ujx, ujy: integer);  
    end;
```

Az ilyen osztályból tilos objektum létrehozása (példányosítás), az ilyen osztály csak azért van létrehozva, hogy abból származtassunk új osztályakat. Ha mégis megpróbálkoznánk vele, a fordító figyelmeztet bennünket, hogy az osztály absztrakt metódust tartalmaz.

A származtatott osztályoknál az absztrakt metódusokat az **override** kulcsszó segítségével írhatjuk felül. Nem kötelező rögtön mindegyiket felülírni, de akkor megint nem lehet példányosítani (objektumot létrehozni belőle).

Az absztrakt metódusnak mindig „virtual” vagy „dynamic”-nak kell lennie, különben hibaüzenetet kapunk.

Ezzel elértük azt, hogy a fordítóprogram figyelmeztet bennünket, ha elfelejtettük felüldefiniálni a származtatott osztályokban az absztrakt metódusokat (Kirajzol, Letorol) – legkésőbb akkor, amikor a TKor osztályból objektumot akarunk létrehozni.

1.9. Típuskompatibilitás

Milyen objektumokat lehet átadni az alábbi eljárásnak?

```
procedure JobbraMozdit(graf: TGrafObj);  
begin  
  graf.Mozgat(graf.x + 1, graf.y);  
end;
```

Az eljárásnak átadható minden olyan objektum, amely a TGrafObj osztályból vagy annak valamelyik gyerekosztályából készült.

```
var  
  k: TKor;  
  n: TNegyzet;  
begin  
  ...  
  JobbraMozdit(k);  
  JobbraMozdit(n);  
end.
```

Tehát: a gyermek-osztályok felülről kompatibilis típusok az ősz-osztályukkal, és azok őseivel is: biztos, hogy megvannak nekik is ugyanazon nevű mezők és metódusok, mint az ősenek (mivel örökölte őket). Ebből következik, hogy **minden osztály kompatibilis a TObject-el**.

Viszont a JobbraMozdit eljárás az alábbi paraméterezéssel **nem működik**, mivel a TObject-nek nincs „Mozgat” metódusa.

```
procedure JobbraMozdit(graf: TObject);
begin
  graf.Mozgat(graf.x + 1, graf.y);
end;
```

Tehát a típus megválasztásánál mindig azon osztály nevét kell megadni, amelyik már tartalmazza az összes olyan metódust és mezőt, amelyekre az eljárás törzsében hivatkozunk.

1.10. Típusellenőrzés (is operátor)

Előfordulhat azonban, hogy nem sikerül találni a fenti követelménynek megfelelő típust. Ilyenkor olyan típust kell megadni, amelyikkel minden szóba jöhető példány kompatibilis (ez legrosszabb esetben a TObject), és az eljáráson belül if-ek (elágazások) segítségével kell eldönteni, hogy a paraméterként megadott példány valójában milyen típusba tartozik. A típus vizsgálatához az **is** operátor használható a következő képpen: *objektumnév IS osztálynév*. Pl.:

```
procedure JobbraMozdit(graf: TObject);
begin
  if graf is TKor then ...
end;
```

A then ágra akkor kerül a vezérlés, ha a „graf” objektum kompatibilis a TKor osztállyal. Vagyis ha a „graf” egy TKor vagy annak valamelyik leszármazott osztályából létrehozott objektum.

De az alábbi módon az eljárás még mindig nem működik, ugyanis a graf-ról az eljárás még mindig úgy tudja, hogy TObject és ennek nincs Mozgat metódusa.

```
procedure JobbraMozdit(graf: TObject);
begin
  if graf is TKor then
    graf.Mozgat(graf.x + 1, graf.y);
end;
```

Az **is** operátor csak egy ellenőrző függvény, nem alakítja át a TObject-et TKor-é!

1.11. Típuskonverzió (as operátor)

A fenti probléma megoldására alkalmazható az **as** operátor. Ennek használata: *objektumnév AS osztálynév*. Pl.:

```
procedure JobbraMozdit(graf: TObject);
begin
  if graf is TKor then
    (graf as TKor).Mozgat(graf.x + 1, graf.y);
end;
```

vagy egy másik megoldás:

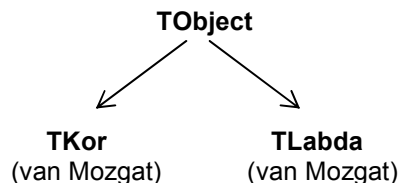
```
procedure JobbraMozdit(graf: TObject);
begin
  if graf is TKor then
    TKor(graf).Mozgat(graf.x + 1, graf.y);
end;
```

Ilyenkor a kifejezés idejére az objektumot a fordítóprogram úgy tekinti, mintha az adott osztályból készült példány lenne.

Ha az **as** operátort helytelenül alkalmazzuk (pl. az objektum nem kompatibilis a megadott osztállyal), akkor futás közben hibaüzenetet kapunk! Ezért az **as** operátort az **is** operátorral való vizsgálat után alkalmazzuk!

1.12. Interface-k (interface)

Képzeliük el, hogy a következő helyzet áll elő:



Van egy TKor osztályunk, melyet a TObject-ből származtattunk és van egy TLabda osztályunk, melyet szintén a TObject-ből származtattunk. Mindkét származtatott osztálynak van Mozgat

metódusa, mellyel a megadott osztályból készített objektumot adott x,y koordinátákra helyezzük. A TObject osztálynak (ős osztálynak) viszont nincs ilyen Mozgat metódusa. Milyen típust adjunk meg ilyenkor az alábbi eljárás paramétereiként?

```
procedure ArrebRak(x: ..... );
begin
  x.Mozgat(12,17);
end;
```

Eddigi ismereteink alapján az egyetlen szóba jöhető osztály a TObject, viszont ekkor az eljárás belsejét meg kell változtatnunk a következőre:

```
if (x is TKor) then (x as TKor).Mozgat(12,17);
if (x is TLabda) then
  (x as TLabda).Mozgat(12,17);
```

Ez azonban nem a legjobb megoldás, ugyanis később ha újabb osztályok kerülnek elő, melyeknek van Mozgat metódusuk, azokra újabb if-ekkel kell kiegészíteni az eljárást.

Az ilyen esetekben a megoldás az **interface** használata lehet. Segítségével azt lehet leírni, hogy különböző ágakon levő osztályokban mi a közös:

```
type
  IMozgathato = interface
    procedure Mozgat(x,y: integer);
  end;
```


Az interface olyan osztálynak látszó valami, melyben:

- csak a metódusok fejrésze van leírva, de soha nincs kidolgozva egyetlen metódus sem,
- tartalmazhat property-ket is, de szintén nincsenek kidolgozva, csak a property neve van megadva, típusa, és az, hogy írható vagy olvasható,
- mezőket viszont nem tartalmazhat az interface,
- szintén nem tartalmazhat konstruktort és destruktort,
- minden rész az interface-ben automatikusan publikus, más nem lehetséges,
- a metódusokat nem lehet megjelölni virtual, dynamic, abstract, override kulcsszavakkal,
- az interface-eknek lehetnek ősei, melyek szintén interface-ek.

Miután definiáltuk az interface-t, az objektumokat, melyek implementálják ezt az interface-t a következő képpen definiálhatjuk:

```
type
TKor = class(IMozgathato)
    public
    procedure Mozgat(x,y: integer);
end;

TLabda = class(IMozgathato)
    public
    procedure Mozgat(x,y: integer);
end;
```

Az ArrebRak eljárásunk pedig így módosul:

```
procedure ArrebRak(x: IMozgathato)
begin
    x.Mozgat(12,17);
end;
```

A típuskompatibilitást tehát kibővíthetjük: **az osztály kompatibilis az őseivel és az interface-ekkel is**, melyeket megvalósít.

Egy osztálynak mindig csak egy őse lehet, de teszőleges számú interface-t implementálhat. Pl.:

```
type
TKor = class(TGrafObj, IMozgathato, ITorolheto)
    ...
end;
```

1.13. Osztály-változók

Hogyan hozhatunk létre olyan mezőt, amely pl. azt számolja, hogy mennyi objektumot készítettünk az adott osztályból. Legyen ennek a mezőnek a neve „darab”. Ekkor az osztály a következő képpen nézhet ki:

```
type TAkarmi = class
    ...
    constructor Create;
    ...
end;

constructor TAkarmi.Create;
begin
```

```
darab := darab + 1;
end;
```

De hol deklaráljuk a „darab” mezőt? A TArkmi osztályon belül nem deklarálhatjuk, mivel ekkor minden példányosításnál egy új mező jön létre, melynek értéke kezdetben meghatározatlan.

A megoldás, hogy nem az osztályban, hanem a Unit-ban deklaráljuk:

```
unit Sajat;

interface

type TArkmi = class
    ...
    constructor Create;
    ...
end;

implementation

var darab: integer;

constructor TArkmi.Create;
begin
    darab := darab + 1;
end;

...

end.
```

A Delphi-ben tehát az ilyen mezők valójában egyszerű globális (statikus) változók.

Más nyelveken, mint pl. a Java, C# az ilyen jellegű mezőket az osztály részévé lehet tenni, természetesen megjelölve, hogy azok csak egy példányban kelljenek. Erre általában a „static” módosító szó használható.

A következő kérdés, hogy hogyan lehet az ilyen változók kezdőértékét megadni. Ezt a unit inicializációs részében tehetjük meg:

```
unit Sajat;

interface

type TArkmi = class
    ...
    constructor Create;
    ...
end;

implementation

var darab: integer;

constructor TArkmi.Create;
begin
    darab := darab + 1;
end;

...

initialization

    darab := 0;

end.
```

Más programozási nyelveken erre létezik egy ún. osztály szintű konstruktor, mely a program indulásának elején hívódik meg.

Hogyan tudjuk az ilyen mezők értékét lekérdezni?

Ha a mező a unit **interface** részében lenne deklarálva, akkor egyszerűen elérhető a unit-on kívülről is.

Ha az **implementation** részben van deklarálva, mint a mi esetünkben is, akkor nem érhető el kívülről, tehát kell hozzá készítenünk egy függvényt, amely kiolvassa az értékét:

```
unit Sajat;

interface

type TAkarmi = class
    ...
    constructor Create;
    function Darabszam: integer;
    ...
end;

implementation

var darab: integer;

constructor TAkarmi.Create;
begin
    darab := darab + 1;
end;

function TAkarmi.Darabszam;
begin
    result := darab;
end;

...

initialization

    darab := 0;

end.
```

Ez azonban nem a legjobb megoldás, mivel a Darabszam függvény használatához létre kell hoznunk egy objektumot a TAkarmi osztályból:

```
var x: TAkarmi;
begin
    x := TAkarmi.Create;
    Write('Eddig ', x.Darabszam, ' objektum van. ');
    ...
end;
```

1.14. Osztály-metódusok

Az osztály szintű metódusok segítségével megoldható a fenti probléma, mivel az ilyen metódusok hívhatók példányosítás nélkül is. Az osztály-metódusok a **class** módosító szóval vannak megjelölve.

```
type
    TAkarmi = class
        ...
        class function Darabszam: integer;
        ...
    end;

...

class function TAkarmi.Daraszam;
begin
    result := Darab;
end;

...
```

Ezt a függvényt most már példányosítás nélkül is tudjuk használni:

```
begin
  ...
  Write('Eddig ', TArkami.Darabszam, ' drb van. ');
  ...
end;
```

A ilyen osztály szintű metódusok belsejében csak osztály szintű mezőket használhatunk, példány szintűt (melyek az osztály belsejében vannak definiálva) nem használhatunk!

1.15. Hivatkozás a példányra a self azonosítóval

Az objektumra a saját metódusának belsejében a **self** (más nyelvekben „this”) azonosítóval hivatkozhatunk. Pl:

```
type
  TJatekos = class
    ...
    nev: string;
    penz: integer;
    procedure
      Vesztett(nyertes: TJatekos);
    procedure
      Gratulal(gratulalo: TJatekos);
    ...
  end;

procedure TJatekos.Vesztett;
begin
  penz := penz - 1000;
  nyertes.penz := nyertes.penz + 1000;
```

```
nyertes.Gratulal(self);
end;

procedure TJatekos.Gratulal;
begin
  writeln('Gratulalt nekem: ', gratulalo.nev);
end;
```

2. OOP a gyakorlatban

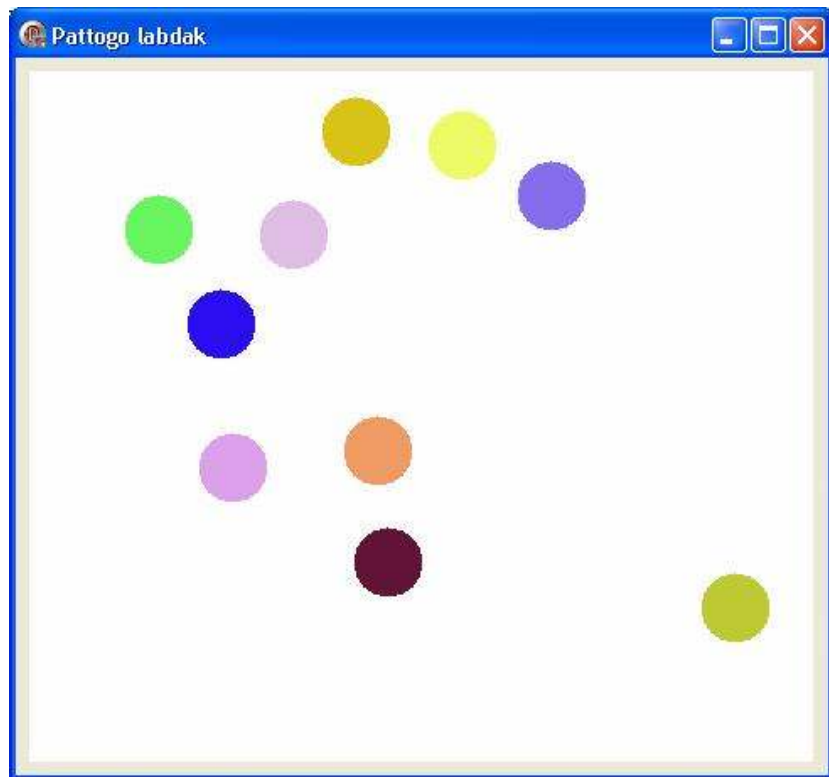
Ebben a részben néhány gyakorlati példán keresztül szemléljük az objektum orientált programozás alapjait.

A gyakorlatban az osztályokat (azok deklarációját és metódusainak implementációját) mindig külön Unit-ba tesszük, így programunk áttekinthető lesz és be lesz biztosítva az adatretetés is az alkalmazáson belül.

Új unitot az alkalmazásba a **File – New – Unit - Delphi for Win32** menüpont segítségével tehetünk.

2.1. Pattogó labdák

Ebben az alkalmazásban készítünk egy TLabda osztályt, melyet kihasználva írunk egy olyan programot, melyben labdákat jelenítünk meg. Mindegyik labda valamilyen irányba mozog, ha eléri a kép szélét, akkor visszapattan. Ha valamelyik labda ütközik egy másik labdával, akkor elpattannak egymástól. 002



Nézzük először is mit tartalmaz a TLabda osztályunk, melyet a Unit2 modulban írtunk meg:

```
unit Unit2;

interface

uses ExtCtrls, Graphics;

type TLabda = class
    private
        hova:TImage;
        sz:integer;

```

```
public
    x,y,dx,dy:integer;
    constructor Create(kx, ky,
                      kdx, kdy, ksz:integer;
                      khova:TImage);
    procedure Kirajzol;
    procedure Letorol;
    procedure Mozgat;
end;
```

implementation

```
constructor TLabda.Create;
begin
    x:=kx;
    y:=ky;
    dx:=kdx;
    dy:=kdy;
    sz:=ks;
    hova:=khova;
end;

procedure TLabda.Kirajzol;
begin
    hova.Canvas.Pen.Color := sz;
    hova.Canvas.Pen.Width := 2;
    hova.Canvas.Brush.Color := sz;
    hova.Canvas.Ellipse(x-20,y-20,x+20,y+20);
end;

procedure TLabda.Letorol;
begin
    hova.Canvas.Pen.Color := clWhite;
    hova.Canvas.Pen.Width := 2;
    hova.Canvas.Brush.Color := clWhite;
    hova.Canvas.Ellipse(x-20,y-20,x+20,y+20);
end;

procedure TLabda.Mozgat;
begin
    Letorol;
    x:=x+dx;
    y:=y+dy;
    if (x+dx+20>hova.Width) or (x+dx-20<0) then
        dx:=-dx;
```

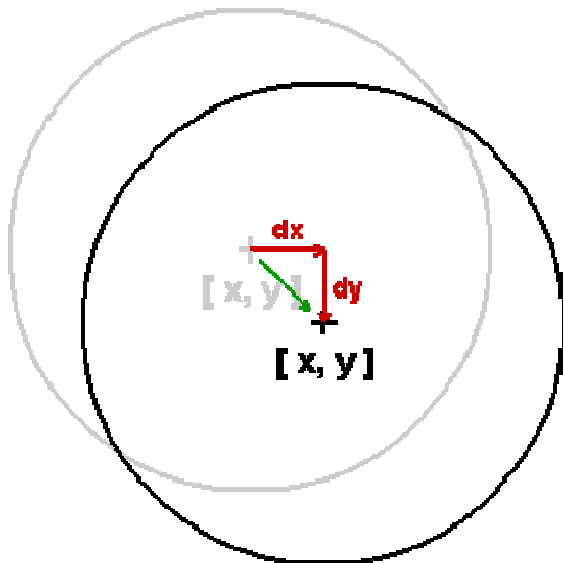
```

if (y+dy+20>hova.Height) or (y+dy-20<0) then
    dy:=-dy;
    Kirajzol;
end;

end.

```

A TLabda osztályunk **hova** mezőjében jegyezzük meg azt az Image komponens, ahová a labdát kirajzoljuk. Az **sz** mezőben a labda színet tároljuk. Az **x, y** mezőkben vannak a labda aktuális koordinátái a képhez viszonyítva, a **dx** és **dy** mezőkben pedig azt jegyezzük meg, hogy a labda új helyzete az előző helyzetéhez képest mennyivel legyen arrébb (ezek értéke valójában csak -1, 0, 1 pixel lehet, melyek megadják egyben a labda mozgásának az irányát is).



Az ábrán a szürke kör jelöli a labda régi helyét, a fekete a labda az új helyét egy elmozdulás után. Láthatjuk, hogy a labda új **x, y** koordinátáit úgy számíthatjuk ki, hogy az **x**-hez hozzáadjuk a **dx**-et, **y**-hoz pedig a **dy**-t.

Az osztályban definiáltunk egy **Create** konstruktort, melynek paramétereként megadjuk a kezdeti x, y, dx, dy értékeket, ahol a labda a létrehozáskor megjelenjen, a labda színét és azt, hogy hova akarjuk a labdát kirajzolni (melyik Image komponensre). A konstruktorunk valójában csak annyiból áll, hogy ezeket a paraméterben megadott értékeket hozzárendeli az objektum megfelelő mezőjéhez.

A **Kirajzol** metódus kirajzol a hova mezőben megjegyzett image komponensre egy 20 pixel sugarú kört az sz-ben tárolt színnel.

A **Letorol** metódus egy ugyanilyen kört rajzol ki, csak fehér színnel. Tehát valójában letörli a kört.

A **Mozgat** metódus letörli a kört a Letorol metódus meghívásával, majd kiszámolja a labda új x és y koordinátáit figyelembe véve azt, hogy ha a labda már kimenne a kép szélén, akkor a dx vagy dy-t az ellentétesre változtatja. Végül kirajzolja a labdát a Kirajzol metódus meghívásával.

Ezt a TLabda osztályt fogjuk most az alkalmazásunkban felhasználni. Ehhez az alkalmazásunk formján hozzunk létre egy Image komponens, melyre a labdákat fogjuk kirajzoltatni (Image1) és egy Timer komponens (Timer1), amely a labdák arrébb mozdtítását a megadott időközökben elvégzi. Ezt az intervallumot állítsuk be 10 ezredmásodpercre.

Ne felejtsük el a programunk uses részét kibővíteni a Unit2 modullal, melyben a TLabda osztályt készítettük el.

Az alkalmazásunkban lesz egy 10 elemű TLabda típusú tömb, mely valójában a 10 pattogó labda objektumunk lesz.

Ezt a 10 objektumot a **Form – OnCreate** eseményében fogjuk létrehozni.

A labdák mozgatását a **Timer – OnTimer** eseményében végezzük el.

Végül a programból való kilépéskor a Form megszüntetésekor a **Form – OnDestroy** eseményben felszabadítjuk az objektumoknak lefoglalt memóriát.

Alkalmazásunk forráskódja így néz ki:

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Image1: TImage;
    Timer1: TTimer;
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

{ \$R *.dfm }

var l:array [1..10] of TLabda;

function Utkozik(a,b:TLabda):boolean;

begin

if Sqrt(Sqr(a.x-b.x)+Sqr(a.y-b.y))<=40 then

result := true

else

result := false;

end;

procedure TForm1.FormCreate(Sender: TObject);

var

i,j,kdx,kdy:integer;

ok:boolean;

begin

randomize;

for i:=1 to 10 do

begin

repeat

kdx:=random(3)-1;

kdy:=random(3)-1;

until (kdx<>0) or (kdy<>0);

l[i]:=TLabda.Create(random(Image1.Width-60)+30,

random(Image1.Height-60)+30,

kdx, kdy,

RGB(random(256),

random(256),

random(256)),

Image1);

repeat

ok:=true;

for j:=1 to i-1 do

if utkozik(l[i],l[j]) then ok:=false;

if not ok then

begin

l[i].x := random(Image1.Width-60)+30;

l[i].y := random(Image1.Height-60)+30;

end;

until ok;

l[i].Kirajzol;

end;

```

end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  i,j,k:integer;
begin
  for i:=1 to 10 do
    begin
      l[i].Mozgat;
      for j:=i+1 to 10 do
        if Utkozik(l[i],l[j]) then begin
          k:=l[i].dx;
          l[i].dx:=l[j].dx;
          l[j].dx:=k;
          k:=l[i].dy;
          l[i].dy:=l[j].dy;
          l[j].dy:=k;
        end;
      end;
    end;
  end;

  procedure TForm1.FormDestroy(Sender: TObject);
  var
    i:integer;
  begin
    for i:=1 to 10 do l[i].Free;
  end;

end.

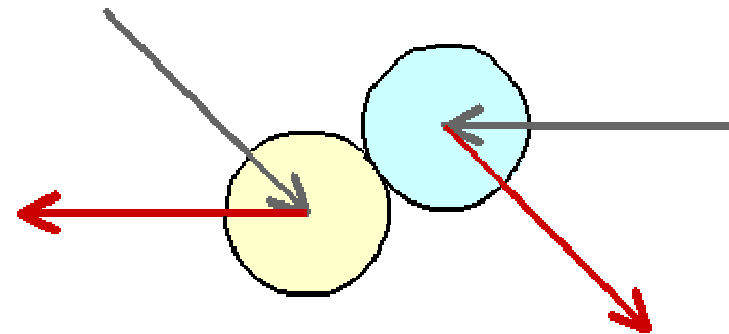
```

A programban létrehoztunk egy **Utkozik** nevű függvényt, melynek mindkét paramétere TLabda típusú. Ez a függvény megállapítja, hogy a paraméterekben megadott labdák ütköznek-e. Valójában ezt úgy állapítjuk meg, hogy kiszámoljuk a két labda közötti távolságot, és ha ez kisebb vagy egyenlő mint 40 (mivel minden labdának a sugara 20), akkor ütközés áll fenn.

A **Form – OnCreate** eseményében sorban létrehozuk a labdákat véletlenszerű adatokkal. Minden labda létrehozása után

ellenőrizzük, hogy nem ütközik-e valamelyik már előtte létrehozott labdával. Ha ütközés van, akkor az új labdának más véletlen koordinátákat generálunk ki és újból leellenőrizzük, nem ütközik-e valamelyik labdával. Ezt mindaddig ismételgetjük, amíg nem sikerül olyan koordinátákat kigenerálnunk, melynél már a labda nem ütközik egyik előtte létrehozott labdával sem. Végül a Kirajzol metódus segítségével kirajzoljuk a labdát, és jöhet a következő labda létrehozása...

A **Timer – OnTimer** eseményében mindegyik labdának meghívjuk a Mozgat metódusát, majd leellenőrizzük hogy a labda az új helyen nem ütközik-e valamelyik utána következő labdával. Ha ütközés áll fenn, akkor egyszerűen kicseréljük a két labda dx, dy adatait, így úgy néz ki, mintha a két labda elpattanna egymástól (amelyik irányba mozgott az egyik labda, abba fog mozogni a másik és amelyikbe mozgott a másik, abba az irányba fog mozogni az előző labda).



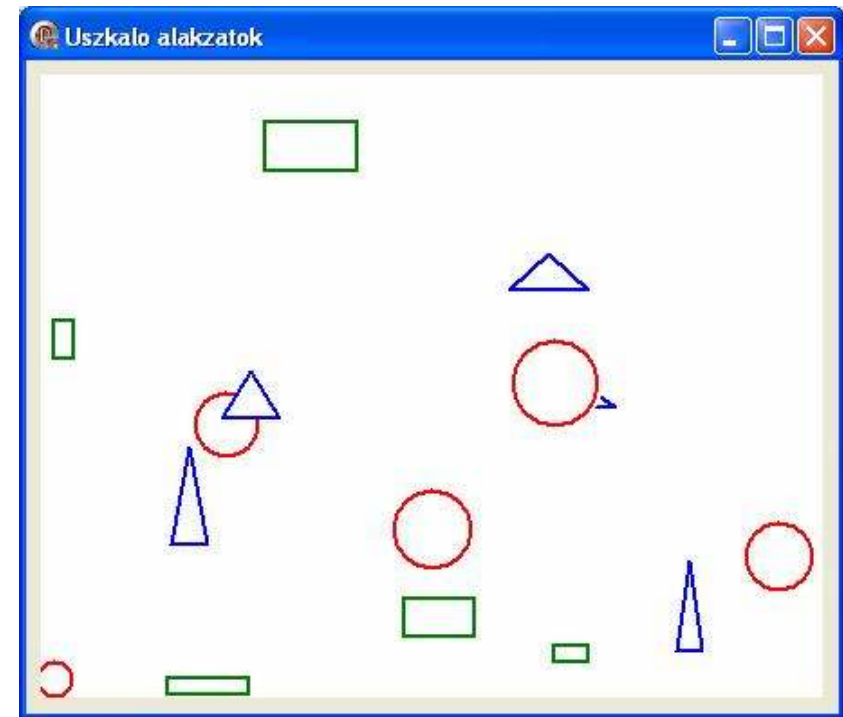
A szürke nyilak jelölik az ütközés előtti régi irányokat (melyek dx, dy segítségével vannak meghatározva), a piros nyilak jelölik az új

irányokat (melyek valójában a két labda dx , dy mezőinek felcserélésével jöttek létre).

A **Form – OnDestroy** eseményében csupán egy ciklus segítségével meghívjuk az összes objektum `Free` metódusát, így felszabadítjuk a lefoglalt memóriát.

2.2. Úszkáló alakzatok (kör, téglalap, háromszög)

A következő feladatban készítünk egy alkalmazást, melynek ablakán (egy `image` komponensen) különböző alakzatok (körök, téglalapok, háromszögek) fognak mozogni, mindegyik alakzatnak lesz egy saját iránya. Az alakzatok egymáson „keresztülmehetnek”, nem fogjuk figyelni az egymással való ütközésüket. Ha azonban az alakzat közepe kimenne az alkalmazás ablakának (pontosabban az `image` komponensnek) valamelyik szélén, akkor az irányát megváltoztatjuk úgy, hogy az alakzat „visszapattanjon”. 001



Az egyes alakzatok valójában objektumok lesznek. Mindegyik objektumnak lesz x , y koordinátája, dx , dy mezői, melyek megadják az alakzat mozgásának irányát (ahogy az előző feladatban). Továbbá mindegyik objektumnak lesz egy **mozdul** metódusa, amely ugyanúgy fog kinézni a körnél, téglalaphoz és háromszöghöz is. Ennek egyetlen feladata lesz: letörölni az alakzatot, kiszámolni az új koordinátákat és kirajzolni az alakzatot.

Láthatjuk, hogy valójában a kör, téglalap és háromszög objektumok nagyon hasonlítanak egymásra, csupán abban különböznek, hogy mit rajzolnak ki a képernyőre. Ezért mindenekelőtt létrehozunk egy általános grafikus osztályt, melyből majd származtatjuk

a kört, téglalapot és háromszöget megvalósító osztályokat. Ennek az általános grafikus osztálynak a programunkban a **TGrafObj** nevet adtuk. Ebből az osztályból hozzuk létre az öröklést felhasználva a **TKor**, **TTeglalap** és **THaromszog** osztályokat.

Az osztályokat tartalmazó Unit2 modulunk így néz ki:

```
unit Unit2;

interface

uses ExtCtrls,Graphics;

type
  TGrafObj = class
  private
    x,y: integer;
    dx,dy: integer;
    kep: TImage;
  public
    constructor Create(kx, ky,
                      kdx, kdy: integer; kkep: TImage);
    procedure kirajzol; virtual; abstract;
    procedure letorol; virtual; abstract;
    procedure mozdul;
  end;

  TKor = class (TGrafObj)
  private
    s:integer;
  public
    constructor Create(kx, ky, kdx, kdy,
                      ks: integer; kkep: TImage);
    procedure kirajzol; override;
    procedure letorol; override;
  end;

  TTeglalap = class (TGrafObj)
  private
    a,b:integer;
  public
```

```
    constructor Create(kx, ky, kdx, kdy,
                      ka, kb: integer; kkep: TImage);
    procedure kirajzol; override;
    procedure letorol; override;
  end;
```

```
  THaromszog = class (TGrafObj)
  private
    a,m:integer;
  public
    constructor Create(kx, ky, kdx, kdy,
                      ka, km: integer; kkep: TImage);
    procedure kirajzol; override;
    procedure letorol; override;
  end;
```

implementation

```
constructor TGrafObj.Create;
begin
  x := kx;
  y := ky;
  dx := kdx;
  dy := kdy;
  kep := kkep;
  kirajzol;
end;

procedure TGrafObj.mozdul;
begin
  letorol;
  if (x+dx>kep.Width) or (x+dx<0) then dx := -dx;
  x := x + dx;
  if (y+dy>kep.Height) or (y+dy<0) then dy := -dy;
  y := y + dy;
  kirajzol;
end;

constructor TKor.Create;
begin
  s := ks;
  inherited Create(kx,ky,kdx,kdy,kkep);
end;

procedure TKor.kirajzol;
```

```

begin
  kep.Canvas.pen.Width := 2;
  kep.Canvas.Pen.Color := clRed;
  kep.Canvas.Ellipse(x-s,y-s,x+s,y+s);
end;

procedure TKor.letorol;
begin
  kep.Canvas.pen.Width := 2;
  kep.Canvas.Pen.Color := clWhite;
  kep.Canvas.Ellipse(x-s,y-s,x+s,y+s);
end;

constructor TTeglalap.Create;
begin
  a := ka;
  b := kb;
  inherited Create(kx,ky,kdx,kdy,kkep);
end;

procedure TTeglalap.kirajzol;
begin
  kep.Canvas.pen.Width := 2;
  kep.Canvas.Pen.Color := clGreen;
  kep.Canvas.Rectangle(x - a div 2, y - b div 2,
                      x + a div 2, y + b div 2);
end;

procedure TTeglalap.letorol;
begin
  kep.Canvas.pen.Width := 2;
  kep.Canvas.Pen.Color := clWhite;
  kep.Canvas.Rectangle(x - a div 2, y - b div 2,
                      x + a div 2, y + b div 2);
end;

constructor THaromszog.Create;
begin
  a := ka;
  m := km;
  inherited Create(kx,ky,kdx,kdy,kkep);
end;

procedure THaromszog.kirajzol;
begin

```

```

  kep.Canvas.pen.Width := 2;
  kep.Canvas.Pen.Color := clBlack;
  kep.Canvas.MoveTo(x - a div 2, y + m div 2);
  kep.Canvas.LineTo(x + a div 2, y + m div 2);
  kep.Canvas.LineTo(x, y - m div 2);
  kep.Canvas.LineTo(x - a div 2, y + m div 2);
  kep.Canvas.FloodFill(x,y,clBlack,fsBorder);
  kep.Canvas.Pen.Color := clBlue;
  kep.Canvas.MoveTo(x - a div 2, y + m div 2);
  kep.Canvas.LineTo(x + a div 2, y + m div 2);
  kep.Canvas.LineTo(x, y - m div 2);
  kep.Canvas.LineTo(x - a div 2, y + m div 2);
end;

procedure THaromszog.letorol;
begin
  kep.Canvas.pen.Width := 2;
  kep.Canvas.Pen.Color := clWhite;
  kep.Canvas.MoveTo(x - a div 2, y + m div 2);
  kep.Canvas.LineTo(x + a div 2, y + m div 2);
  kep.Canvas.LineTo(x, y - m div 2);
  kep.Canvas.LineTo(x - a div 2, y + m div 2);
end;

end.

```

Ami érdekes a **TGrafObj** osztálynál és még nem találkoztunk vele az előző feladatban, az a Letorol és Kirajzol metódusok, melyek itt virtuális, absztrakt metódusok. Tehát itt csak felsoroljuk őket, de azt, hogy konkrétan mit csináljanak ezek a metódusok, nem adjuk meg (nem implementáljuk őket). Ez érthető, hiszen itt ez csak egy általános grafikus osztály, melynél még nem tudjuk mit fog kirajzolni és letörölni. Ezt a TGrafObj osztályt csupán azért készítettük, hogy ebből könnyen létrehozassuk a másik három osztályt.

A **TKor** osztálynak lesz egy plussz mezője, ami a kör sugarát jelöli (**s**). A Create metódusban beállítjuk ennek a mezőnek a kezdeti

értékét, majd az `inherited Create(kx,ky,kdx,kdy,kkep);` parancs segítségével meghívjuk az `ős (TGrafObj)` osztály `Create` metódusát, amely elvégzi a többi beállítást és kirajzolja az alakzatot.

Továbbá megírjuk a `TKor` **Kirajzol** és **Letorol** metódusait. A `Kirajzol` metódus egy fehér belsejű, piros körvonalú, 2 vonalvastagságú kört rajzol ki. A `Letorol` metódus valójában ugyanazt a kört rajzolja ki, csak piros helyett fehér színnel.

Hasonlóan készítjük el a **TTeglalap** és **THaromszog** osztályakat, melyeket szintén a `TGrafObj` osztályból származtatunk.

Miután elkészítettük az egyes osztályokat, megírhatjuk az alkalmazást, amely ezeket használja fel. Hasonlóan az előző feladathoz itt is egy `Image` komponensen fogjuk megjeleníteni az objektumokat és az objektumok mozgását egy `Timer` komponens segítségével valósítjuk meg.

Az alkalmazás forráskódja:

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Image1: TImage;
    Timer1: TTimer;
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
```

```
private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

var k: array [1..15] of TGrafObj;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: integer;
begin
  for i:=1 to 15 do k[i].mozdul;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
  mx,my: integer;
begin
  randomize;
  for i:=1 to 15 do
    begin
      repeat
        mx:=random(5)-2;
        my:=random(5)-2;
      until (mx<>0) or (my<>0);
      if i mod 3 = 0 then
        k[i]:=TKor.Create(random(Image1.Width),
          random(Image1.Height),
          mx, my, random(20)+10, Image1)
      else if i mod 3 = 1 then
        k[i]:=TTeglalap.Create(random(Image1.Width),
          random(Image1.Height),
          mx, my, random(50)+10,
          random(50)+10, Image1)
      else
        k[i]:=THaromszog.Create(random(Image1.Width),
```

```

        random(Image1.Height),
        mx, my, random(50)+10,
        random(50)+10, Image1);

    end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
var
    i:integer;
begin
    for i:=1 to 15 do k[i].Free;
end;

end.

```

Megfigyelhetjük, hogy a programban ugyanazt a tömböt használjuk a TKor, TTeglap, THaromszok objektumokra is. Ez azért lehetséges, mivel ezt a tömböt TGrafObj típusúnak deklaráltuk, és a TGrafObj kompatibilis az összes leszármaztatott osztállyal.

A **Form – OnCreate** eseményében létrehozuk az egyes objektumokat véletlenszerű értékekkel. Attól függően, hogy a ciklusváltozó 3-al való osztási maradéka 0, 1 vagy 2, a tömb adott eleméhez TKor, TTeglap vagy THaromszog osztályokból hozunk létre objektumot.

A **Timer – OnTimer** eseményében mindegyik objektumnak meghívjuk a Mozdul metódusát, amely az adott alakzatot „eggyel arrébb” helyezi.

A **Form – OnDestroy** eseményében megszüntetjük az egyes objektumokat a Free metódusuk segítségével, ezzel felszabadítjuk a lefoglalt területet a memóriában.

3. DLL-ek használata és létrehozásuk

A Windows-ban kétféle futtatható állományok léteznek: programok (EXE fájlok) és dinamikus csatolású könyvtárak (Dynamic Link Libraries = DLL fájlok). Bár a DLL fájlok közvetlenül nem futtathtók, mégis tartalmaznak programkódot (függvényeket, eljárásokat), továbbá tartalmazhatnak erőforrásokat (képeket, hangokat, szövegtáblákat), akárcsak a programok.

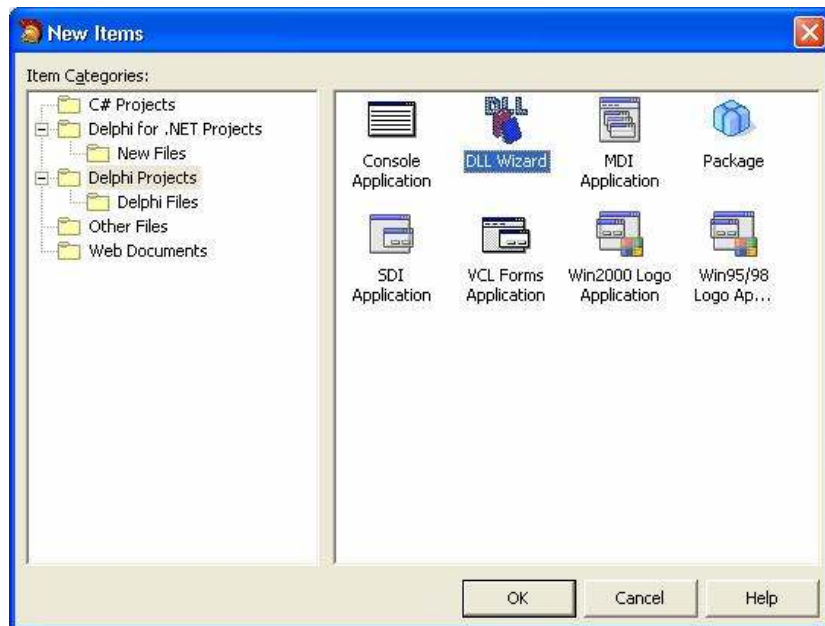
Amikor egy Delphi alkalmazást írunk, akkor egy programfájlt készítünk. Ez a programfájl felhasználhatja a DLL-ekben található függvényeket, eljárásokat, erőforrásokat. De miért is jó, ha nem mindent egy programfájlba rakunk, hanem DLL-be is, és mikor érdemes DLL-eket használni? Ehhez nézzük meg a DLL-ek felhasználhatóságát és azok főbb előnyeit:

- Több program használhatja ugyanazt a DLL-t. Ha tehát készítünk egy DLL-t olyan függvényekkel, eljárásokkal, erőforrásokkal, melyeket gyakran használunk a programjainkban, akkor később bármelyik alkalmazásban felhasználhatjuk ezt.
- Ha több program fut egyszerre, melyek ugyanazt a DLL-t használják, akkor a DLL csak egyszer töltődik be a memóriába. Így tehát memóriát is takaríthatunk meg.
- A Delphi-ben elkészített DLL-eket más programozási nyelven készült alkalmazásokban is felhasználhatjuk. Tehát a DLL-ben található függvényeket, eljárásokat felhasználhatjuk akár C, Visual Basic, stb. nyelvekben írt alkalmazások készítésénél is.

A következő néhány oldalon megismerkedhetünk azzal, hogyan készíthetünk dinamikus csatolású könyvtárakat, és hogyan használhatjuk ezeket alkalmazásainkban.

3.1. DLL készítése

Ha dinamikus csatolású könyvtárat szeretnénk létrehozni, válasszuk ki a Delphi fejlesztőkörnyezetében a **File – New – Other...** menüpontot. A megnyíló ablakban válasszuk ki a **Delphi Projects** kategória alatt a **DLL Wizard**-ot, majd klikkeljünk az OK gombra.



Ekkor megjelent egy üres dinamikus csatolású könyvtár forráskódja. Ebbe írhatjuk bele azokat a függvényeket és eljárásokat, melyeket szeretnénk, ha a DLL-ünk tartalmazna.

Készítsünk most egy olyan dinamikus csatolású könyvtárat, amely tartalmazni fog egy függvényt két szám legnagyobb közös osztójának meghatározására (**lko**), egyet pedig a legkisebb közös többszörös meghatározására (**lkt**). Továbbá a DLL tartalmazzon még egy paraméter nélküli függvényt, amelynek meghívásával megkapjuk a Fibonacci számsor (1, 1, 2, 3, 5, 8, 13, 21, ...) következő elemét (**fib**) és egy paraméter nélküli eljárást, melynek segítségével beállítjuk a Fibonacci számok generálását az elejére (**fiboinit**). **005**

Az egyes eljárások és függvények megírása után fontos, hogy azokat a függvényeket és eljárásokat, melyeket kívülről (a programokból, melyek a DLL-t használni fogják) is meg kívánunk hívni, felsoroljuk az **exports** záradék után. Itt adhatunk a függvényeknek és eljárásoknak más nevet is (ahogy ezt tettük a fibo-nál és fiboinit-nél), vagy hagyhatjuk azt a nevet, melyen a DLL-ben a függvényt megírtuk. Ha új nevet adunk, akkor az alkalmazásokból, melyekben a DLL-függvényt vagy eljárást használni akarjuk, az új néven érhetjük majd el.

Ha más fejlesztői rendszerből is el szeretnénk érni a DLL exportált eljárásait és függvényeit, akkor a Delphi-ben alapértelmezett **register** paraméterátadási mód helyett a **stdcall** szabványos Win32 paraméterátadási módot kell használnunk.

Nézzük most meg, hogyan is néz ki a könyvtár forráskódja:

```
library dll_pelda;  
  
uses  
    SysUtils, Classes, Math;
```

```

{$R *.res}

{ket szam legnagyobb kozos osztaja}
function lko(a,b: integer): integer; stdcall;
var
  i: integer;
begin
  i := min(a,b);
  while (a mod i > 0) or (b mod i > 0) do dec(i);
  result := i;
end;

{ket szam legkisebb kozos tobszorose}
function lkt(a,b: integer): integer; stdcall;
var
  i: integer;
begin
  i := max(a,b);
  while (i mod a > 0) or (i mod b > 0) do inc(i);
  result := i;
end;

{Fibonacci szamok}
var
  a1, a2, ssz: int64;

{a Fibonacci szamsor kovetkezo eleme}
function fibo: int64; stdcall;
begin
  if ssz<=2 then result := 1
  else begin
    result := a1 + a2;
    a1 := a2;
    a2 := result;
  end;

  inc(ssz);
end;

{a Fibonacci szamgenerator inicializalasa}
procedure fiboinit; stdcall;
begin
  a1 := 1;
  a2 := 1;
  ssz := 1;

```

```

end;

{exportalando alprogramok listaja}
exports
  lko, lkt,
  fibo name 'Fibonacci',
  fiboinit name 'StartFibonacci';

{a DLL inicializalasa}
begin
  fiboinit;
  beep();
end.

```

A forráskód végén a begin és end között történik a DLL inicializálása. Az ide leírt program csak egyszer fut le, mielőtt bármelyik függvényt használnánk. A mi programunkban itt inicializáljuk a Fibonacci számgenerátort majd a beep függvénnyel megszólaltatjuk a hangszórót – így majd halljuk mikor fut le az alkalmazásunkban ez az inicializációs rész.

Amik nem közvetlenül a dinamikus csatolású könyvtárakhoz tartoznak, de eddig még nem biztos, hogy találkoztunk velük, az a **min** és **max** függvények, melyek a **Math** unitban találhatók. Ezek két szám közül visszaadják a kisebb, illetve nagyobb számot.

Ami még a forráskódban új lehet, az az **int64** típus. Ez a Delphi-ben használható legnagyobb egész szám típus. Azért használtuk ezt az integer típus helyett, mivel a Fibonacci számok generálásakor nagyon hamar elérhetünk olyan értékeket, melyeket már az integer típusban nem tudunk tárolni.

A megírt forráskódot a **Project – Compile ... (Ctrl+F9)** menüpont segítségével tudjuk lefordítani és így létrehozni a DLL kiterjesztésű fájlt.

Ha már megírtuk volna azt az alkalmazást, amely a dinamikus csatolású könyvtárat használja, akkor azt futtatva tesztelhetjük a DLL futását is a közvetlenül a Delphi-ből. Ehhez elég a DLL mappájába (ahova elmentettük) átmásolnunk a lefordított EXE fájlt és a **Run – Parameters...** menüpontot kiválasztva a **Host application** alatt megadni ezt az alkalmazást. Ezután a DLL-t tesztelhetjük a beállított alkalmazásunk segítségével a **Run** főmenüpontot kiválasztva (pl. Run – Run menüponttal).

3.2. A DLL felhasználása alkalmazásunkban

Ahhoz, hogy a kész dinamikus csatolású könyvtár függvényeit és eljárásait fel tudjuk használni alkalmazásainkban, a lefordított DLL állományt MS Windows XP rendszer esetében az alábbi alkönyvtárak valamelyikében kell elhelyeznünk:

- abba a könyvtárba, ahonnan az alkalmazás betöltődik,
- az aktuális könyvtárban,
- **Windows\System32** könyvtárban,
- **Windows\System** könyvtárban,
- **Windows** könyvtárban
- a **path** környezeti változóban felsorolt könyvtárak valamelyikében

A mi esetünkben most a legegyszerűbb a lefordított DLL állományt oda átmásolni, ahova az alkalmazásunkat menteni fogjuk és ahol létrejön majd az alkalmazásunk EXE fájlja (tehát ahonnan betöltődik az alkalmazásunk). Gondolom természetes, hogy ha a kész

alkalmazásunkat majd terjeszteni szeretnénk, akkor az EXE állománnyal együtt a DLL állományt is terjesztenünk kell, mivel az nélkül a programunk nem fog működni.

De nézzük meg, hogyan is tudjuk az alkalmazásunkban felhasználni a megírt dinamikus csatolású könyvtárat.

Készítsük el az alábbi alkalmazást, melyben felhasználjuk az előző feladatban megírt DLL állományt. Ne felejtjük el a lefordított DLL fájlt (dll_pelda.dll) átmásolni abba a mappába, ahová az alkalmazásunkat menteni fogjuk! 006



Ahhoz, hogy a DLL-ben definiált alprogramokat elérhetővé tegyük, importálnunk kell őket. Minden DLL-ben található rutin külső definíció a programunk számára (**external**).

A Delphi-ben az importálás lehet az alprogram neve alapján, pl.:

```
function lko(a,b: integer):integer; stdcall;
    external 'dll_pelda.dll';
```

vagy akár általunk megadott név alapján (átnevezéssel), pl.:

```
function LegkKozTobb(a,b: integer):integer;
    stdcall; external 'dll_pelda.dll' name 'lkt';
```

Ezek után az **lko** és **LegkKozTobb** függvényeket bárhol felhasználhatjuk a programunkban.

A fenti két példa esetében fontos, hogy az **lko** és **lkt** nevek szerepeljenek a **dll_pelda** modul **exports** részében!

Az alkalmazásunkhoz tartozó forráskód fontosabb része:

```
...
implementation
{$R *.dfm}

function lko(a,b: integer):integer; stdcall; external
    'dll_pelda.dll';

function LegkKozTobb(a,b: integer):integer; stdcall;
    external 'dll_pelda.dll' name 'lkt';

procedure StartFibonacci; stdcall; external
    'dll_pelda.dll';

function Fibonacci:int64; stdcall; external
    'dll_pelda.dll';

procedure TForm1.Button1Click(Sender: TObject);
var
    x,y: integer;
begin
```

```
    x := StrToInt(Edit1.Text);
    y := StrToInt(Edit2.Text);
    ShowMessage('Legnagyobb közös osztó: '
        + IntToStr(lko(x,y)));
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    x,y: integer;
begin
    x := StrToInt(Edit1.Text);
    y := StrToInt(Edit2.Text);
    ShowMessage('Legkisebb közös többszörös: '
        + IntToStr(LegkKozTobb(x,y)));
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    StartFibonacci;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    Memo1.Text := Memo1.Text
        + IntToStr(Fibonacci) + ', ';
end;

end.
```

3.3. Statikus és dinamikus importálás

A DLL eljárásaink és függvényeink importálását statikus és dinamikus módon is elvégezhetjük.

Az eddigiek során a **statikus importálást** használtuk. A statikus importálás esetén a rendszer a dinamikus csatolású könyvtárat akkor tölti be a memóriába, amikor az azt használó program első példánya elindul. Minden további a könyvtárra hivatkozó alkalmazás, a már betöltött könyvtár alprogramjait használja.

A **dinamikus importálás** esetén az alkalmazás futása során saját magunk töltjük be a memóriába a dinamikus csatolású könyvtárat. Ha a könyvtárra már nincs szükségünk, akkor egyszerűen felszabadítjuk azt. A betöltés és felszabadítás műveletéhez a Windows API-függvényeket használjuk, melyek a Windows unitban találhatók.

A dinamikus importálás első lépése a könyvtár betöltése a memóriába. Ezt a **LoadLibrary()** függvény segítségével tehetjük meg:

```
function LoadLibrary(KönyvtarNeve: PAnsiChar): HModule;
```

A függvény paramétereként csak a könyvtár nevét kell megadni elérési útvonal nélkül. Ha a függvény 0 (nulla) értékkel tér vissza, akkor sikertelen volt a betöltés, egyébként a betöltött modul azonosítóját kapjuk vissza.

Ha a DLL-t sikerült betöltenünk a memóriába, akkor hivatkozhatunk a DLL-ben tárolt függvényekre és eljárásokra. A hivatkozáshoz azonban meg kell tudnunk a DLL-függvény vagy eljárás belépési pontjának címét. A cím lekérdezéséhez a **GetProcAddress()** függvényt használhatjuk:

```
function GetProcAddress(hLibModule: HModule;  
                        AlprogNeve: LPCStr): FARProc;
```

A függvény paramétereként meg kell adnunk az előző függvény visszatérési értékeként kapott DLL modul azonosítóját és az alprogram (DLL-függvény vagy eljárás) nevét. A függvény visszatérési értékeként

megkapjuk a paraméterben megadott alprogram címét. Ha nincs ilyen alprogram a DLL-ben, akkor visszatérési értéként **nil**-t kapunk.

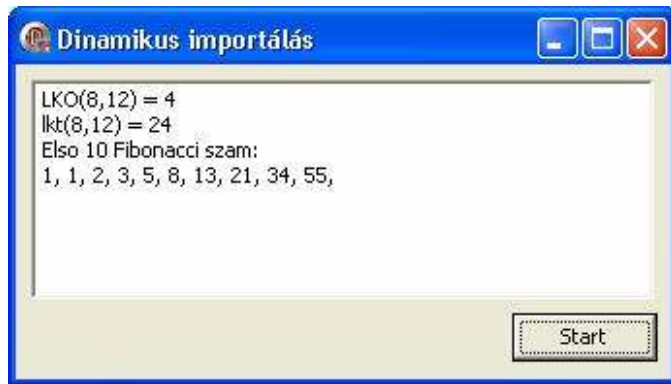
Ha már nincs szükségünk az alkalmazásban a betöltött dinamikus csatolású könyvtárra, akkor azt a **FreeLibrary** függvény meghívásával szabadíthatjuk fel a memóriából:

```
function FreeLibrary(hLibModule: HModule): Bool;
```

Paraméterként a LoadLibrary függvény meghívásakor kapott azonosítót kell megadnunk.

Nézzük meg egy példa segítségével, hogyan használhatjuk a dinamikus importálást az alkalmazásunkban. A programban a 3.1. fejezetben létrehozott DLL-t fogjuk felhasználni, ezért ezt a fájlt (dll_pelda.dll) ne felejtsük el átmásolni abba a könyvtárba, ahová az alkalmazást el fogjuk menteni (ahonnan futtani fogjuk).

Az alkalmazásunk most csak egy üres Memo és egy Button komponenseket tartalmazzon. A nyomógomb megnyomásakor a DLL-függvény segítségével kiszámoljuk, majd kiírjuk a Memo komponensbe 8 és 12 legnagyobb közös osztóját, legkisebb közös többszörösét és az első 10 Fibonacci számot. **007**



Az alkalmazásunk nyomógombjának **OnClick** eseményéhez tartozó forráskód:

```
...

procedure TForm1.Button1Click(Sender: TObject);

{ megfelelo mutatotipusok es fuggvények
  az eljárások hivasához }

type
  TLkoLkt = function (a,b:integer):integer; stdcall;
  TFibo = function:int64; stdcall;
  TFiboStart = procedure; stdcall;

var
  lko, lkt: TLkoLkt;
  fibo: TFibo;
  fibostart: TFiboStart;
  i: integer;
  HLib: THandle;

begin
  Mem1.Clear;
  { könyvtár betöltése }
  HLib := LoadLibrary('dll_pelda.dll');
```

```
if hlib = 0 then
  ShowMessage('A dll_pelda.dll betöltése
              sikertelen volt!')
else
  try
    { belepesi pontok lekerdezese }
    addr(lko) := GetProcAddress(HLib,'lko');
    addr(lkt) := GetProcAddress(HLib,'lkt');
    addr(fibostart) := GetProcAddress(HLib,
                                     'StartFibonacci');
    addr(fibo) := GetProcAddress(HLib,'Fibonacci');
    { fuggvények hivasá }
    if addr(lko) <> nil then
      Mem1.Lines.Add('LKO(8,12) = '
                    + IntToStr(lko(8,12)));
    if addr(lkt) <> nil then
      Mem1.Lines.Add('lkt(8,12) = '
                    + IntToStr(lkt(8,12)));
    if addr(fibostart) <> nil then
      fibostart;
    if addr(fibo) <> nil then
      begin
        Mem1.Lines.Add('Első 10 Fibonacci szám: ');
        for i:=1 to 10 do Mem1.Text := Mem1.Text
                               + IntToStr(fibo) + ', ';
      end;
    finally
      { könyvtár felszabadítása }
      FreeLibrary(HLib);
    end;
  end;
...
```

Amivel az eddigi programozásunk során nem találkoztunk, az a függvény és eljárás típusok definiálása. Az így definiált függvényeknek és eljárásoknak a programban meg kell adnunk a belépési címüket (ahol a memóriában a függvény és alprogram található). A függvény címét az **addr** operátor segítségével adhatjuk meg, illetve kérdezhetjük le.

3.4. A DLL és a memória

A dinamikus csatolású könyvtárak alaphelyzetben nem támogatják a hosszú sztringek paraméterként, illetve függvényértékként való átadását. Amennyiben erre szükség van, mind a DLL, mind pedig az alkalmazás **uses** részében az első helyen meg kell adnunk a **ShareMem** unitot, a lefordított programhoz pedig csatolnunk kell a memóriakezelést végző **borlndmm.dll** könyvtárat.

Egyszerűbb megoldáshoz jutunk, ha az **AnsiString (String)** típus helyett a **PChar** (nullvégű sztring) vagy **ShortString** (max. 255 hosszú string) típust használjuk. A nullvégű sztring használata különbözik a megszokott „string” típustól. A ShortString típus használata hasonló, a különbség csupán az, hogy ebben a típusban maximum 255 karakter hosszú sztringet tárolhatunk.

3.5. Form tárolása DLL-ben

A Form DLL-be való helyezésekor ugyanúgy kell eljárunk, mint ha normál alkalmazásba szeretnénk új formot tenni. Tehát miután létrehoztuk a dinamikus csatolású könyvtárat (**File – New – Other... – Delphi Projects – DLL Wizard**), hozzáteszünk ehhez a könyvtárhoz egy formot a **File – New – Form - Delphi for Win32** menüpont segítségével.

A dinamikus könyvtár azon függvényében vagy eljárásában, melyben szeretnénk használni ezt a formot, létrehozuk, megnyitjuk (általában modálisan), majd megszüntetjük.

Készítsünk egy DLL-t, amely tartalmaz egy **AdatBekeres** függvényt. Ez a függvény nyisson meg modálisan egy formot, amelyen egy ScrollBar segítségével beállíthatunk 0-255 közötti számot, majd az „O.k.” gomb megnyomásával a függvény ezt az értéket adja vissza. Ha a formból a „Mégsem” nyomógommbal lépünk ki, akkor a függvény -1 értéket adjon vissza. 008



A fent leírt módon hozzuk létre előbb a DLL-t, majd ehhez adjunk hozzá egy Form-ot. Rakjuk rá a formra a szükséges komponenseket és állítsuk be az eseményeket. A Form és rajta levő komponensek eseményeihez tartozó programkód:

```
...

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
    Label2.Caption := IntToStr(ScrollBar1.Position);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ModalResult := mrOk;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```

begin
  ModalResult := mrCancel;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
  Left := (Screen.Width - Width) div 2;
  Top := (Screen.Height - Height) div 2;
end;

```

...

Ezzel megírtuk a **ScrollBar1 – OnChange**, **Button1 – OnClick**, **Button2 – OnClick** eseményekhez tartozó eljárásokat. A **Form1 – OnShow** eseményéhez is írtunk programkódot, amely a form helyét állítja be, amikor megnyitjuk modálisan (az ablakunkat a képernyő közepén jeleníti meg).

Nézzük most a DLL-t, melyben ezt a Form-ot létrehozzuk, megnyitjuk modálisan, beállítjuk a függvény visszatérési értékét, majd felszabadítjuk a Form-nak lefoglalt memóriát:

```

library dll_form;

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

{$R *.res}

function AdatBekeres:Integer; stdcall;
var
  Form: TForm1;
begin
  { form létrehozasa }
  Form := TForm1.Create(Application);
  { form megjelenitese - adatbekeres }
  if Form.ShowModal = mrOk then

```

```

    result := Form.ScrollBar1.Position
  else
    result := -1;
  { form felszabaditasa }
  Form.Free;
end;

```

Exports

```
AdatBekeres;
```

```

begin
end.

```

A form létrehozásakor a form tulajdonosának az Application objektumot adjuk meg, ami valójában az az alkalmazás lesz, amely a DLL-függvényt használni fogja.

Ahhoz, hogy az Application objektumot és az mrOk konstanst használni tudjuk a DLL-függvényben, ki kellett egészítenünk a dinamikus csatolású könyvtár uses részét a **Forms** és **Controls** unitokkal. A **Unit1** modult a Delphi automatikusan beírta a könyvtár uses sorába amikor hozzáadtuk a formot.

3.6. A DLL (és a benne tárolt form) felhasználása alkalmazás készítésekor Delphi-ben

Az előző fejezetben létrehozott DLL-t felhasználva készítsünk egy egyszerű alkalmazást, amely egy gomb megnyomása után a DLL-függvény (form) segítségével bekér egy számot. A DLL-függvény által visszaadott számot jelenítsük meg egy Label komponensben. 009



Az alkalmazásunk forráskódja:

```
...
implementation
{$R *.dfm}

function AdatBekeres:integer; stdcall;
external 'dll_form.dll';

procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Caption := 'Beolvasott szám: '
                    + IntToStr(AdatBekeres);
end;

end.
```

Láthatjuk, hogy ezt a DLL-függvényt is ugyanúgy használhatjuk, mint az első dinamikus csatolású könyvtárunkban tárolt függvényeket és eljárásokat. A form modális megjelenítését valójában már a maga DLL-függvény végzi el.

3.7. A DLL (és a benne tárolt form) felhasználása a MS Excel makró nyelvén (Visual Basic)

Próbáljuk felhasználni az így megírt DLL-t más programozási nyelvben. Készítsünk a MS Excel-ben egy egyszerű makrót (**Eszközök – Makró – ...**), amely csupán annyit tesz, hogy az aktív cellába beír egy számot, pl. a 15-öt. Mentsük el az állományunkat és az aktív könyvtárba másoljuk mellé a 3.5. fejezetben elkészített DLL állományunkat. Majd szerkesszük a makrót és írjuk át benne azt a részt, ami a 15-ös számot írja be az aktív cellába úgy, hogy a DLL-függvényünk által bekért számot rakja be az aktív cellába a 15 helyett. **010**

A feladat megoldásához a makró Visual Basic-ben írt forráskódját így módosítsuk:

```
Declare Function AdatBekeres
    Lib "dll_form.dll" () As Integer

Sub Makro1()
    ActiveCell.FormulaR1C1 = AdatBekeres
End Sub
```

Látjuk, hogy hasonlóan a Delphi programozási nyelvéhez, itt is előbb deklarálnunk kell a függvényt. A deklarációnál meg kell adnunk, melyik DLL állomány tartalmazza a függvényünket (**dll_form.dll**).

Ezek után a makróban már egyszerűen meghívhatjuk a függvényünket.

3.8. Erőforrások tárolása DLL-ben

Az alkalmazásunk hatékony működése érdekében néha a programkód helyett erőforrásokat (kép, hang, szöveg, ...) tárolunk a dinamikus csatolású könyvtárakban.

Készítsünk két DLL-t (dll_en.dll, dll_hu.dll), melyekben az adott nyelven a program feliratait és az adott nyelvhez tartozó zászlót (hu.bmp és en.bmp képek) tároljuk. A következő feladatban (3.9. fejezetben) majd az alkalmazásunk ezekből a DLL-ekből tölti be a feliratokat a kiválasztott nyelven és a nyelvhez tartozó zászlót. 011



A feliratokat és a képet erőforrás fájlokban fogjuk tárolni, melyeket hozzacsatolunk a DLL-ekhez. A feliratokhoz sztringtáblát definiálunk.

Az **eng.rc** fájl tartalma:

```
kep BITMAP "en.bmp"
```

```
STRINGTABLE DISCARDABLE
BEGIN
```

```
1 "Calculation"
2 "Exit"
3 "Datainput"
4 "Result"
```

```
END
```

A **hun.rc** fájl tartalma:

```
kep BITMAP "hu.bmp"
```

```
STRINGTABLE DISCARDABLE
BEGIN
1 "Számolás"
2 "Kilépés"
3 "Adatbevitel"
4 "Eredmény"
END
```

Ezeket a fájlokat a Delphi részeként megtalálható **brcc32.exe** fordítóval lefordítjuk **.res** kiterjesztésű állományokká:

brcc32.exe eng.rc

brcc32.exe hun.rc

Ezek után a bináris erőforrásfájlokat (.res) a \$R fordítási direktívával beépítjük a dinamikus csatolású könyvtárakba:

A magyar nyelvűt a **dll_hu.dll**-be:

```
library dll_hu;
```

```
uses
  SysUtils,
  Classes;

{$R *.res}
```

```
{ $R hun.res }
```

```
begin
end.
```

Az angol nyelvűt pedig a **dll_en.dll**-be:

```
library dll_en;
```

```
uses
  SysUtils,
  Classes;
```

```
{ $R *.res }
```

```
{ $R eng.res }
```

```
begin
end.
```

Ezzel elkészítettük a két DLL állományt (magyar és angol), melyeket a következő feladatban fogunk felhasználni.

3.9. DLL-ben tárolt erőforrások felhasználása

Készítsük el a fenti két DLL-t használó alkalmazást. A nyelvet főmenü segítségével lehessen változtatni. Az alkalmazás a „Számolás” nyomógomb megnyomásakor az egyik Edit-be beírt számot emelje

négyszetre, majd az eredményt jelenítse meg a másik Edit komponensben. **012**



A formra helyezzük el a szükséges komponenseket (MainMenu1, Label1, Label2, Edit1, Edit2, Button1, Button2, Image1), majd állítsuk be ezek fontosabb tulajdonságait és írjuk meg az eseményekhez tartozó programkódot:

```
...

implementation

{$R *.dfm}

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

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit2.Text := IntToStr(Sqr(StrToInt(Edit1.Text)));
```



```

end;

procedure LoadResource(lib:string);
var
  DLLleiro: THandle;
  Puffer: PChar;      { nullvegu (#0) string }
  Bmp: TBitmap;
begin
  { könyvtár beolvasása }
  DLLleiro := LoadLibrary(PChar(lib));
  if DLLleiro = 0 then
    ShowMessage('Nincs meg a(z) '+lib+' fájl.')
  else
    begin
      { string-ek beolvasása }
      Puffer := StrAlloc(100+1);
      LoadString(DLLleiro,1,Puffer,100);
      Form1.Button1.Caption := Puffer;
      LoadString(DLLleiro,2,Puffer,100);
      Form1.Button2.Caption := Puffer;
      LoadString(DLLleiro,3,Puffer,100);
      Form1.Label1.Caption := Puffer;
      LoadString(DLLleiro,4,Puffer,100);
      Form1.Label2.Caption := Puffer;
      StrDispose(Puffer);
      { kép beolvasása }
      Bmp := TBitmap.Create;
      Bmp.Handle := LoadBitmap(DLLleiro,'kep');
      Form1.Image1.Canvas.Draw(0,0,Bmp);
      Bmp.Free;
      { könyvtár felszabadítása }
      FreeLibrary(DLLleiro);
    end;
end;

procedure TForm1.MagyarHungarian1Click(
                               Sender: TObject);
begin
  LoadResource('dll_hu.dll');
end;

procedure TForm1.EnglishAngol1Click(Sender: TObject);
begin
  LoadResource('dll_en.dll');
end;

```

end.

A kiválasztott DLL-ből az erőforrások (feliratok, kép) betöltéséhez létrehoztunk egy LoadResource nevű eljárást, melynek paramétereként megadjuk a DLL fájl nevét.

Ez az eljárás betölti a DLL-t a memóriába, majd ebből beolvassa a szükséges adatokat a LoadString(), LoadBitmap() függvények segítségével, végül felszabadítja a dinamikus csatolású könyvtárnak lefoglalt memóriát.

4. Párhuzamos programozás, szálak

A többprogramos 32-bites Windows rendszerek működésük közben nem az egyes programokat, hanem a programszálakat kezelik. A **programszál** a Windows szempontjából önálló program, az operációs rendszer határozza meg, hogy mikor kap a szál processzoridőt. Többprocesszoros rendszer esetében az egyes programszálak futtatását külön processzor is elvégezheti.

Egy alkalmazás állhat egy programszálból (mint pl. az eddigi alkalmazásaink) vagy több programszálból is. A programszálak párhuzamosan végzik tevékenységüket. Például a MS Word alkalmazás működése közben a helyesírás ellenőrző egy külön programszálon fut, ezért tud folyamatosan (párhuzamosan) dolgozni, miközben a felhasználó szerkeszti a dokumentumot.

4.1. TThread osztály

Ha többszálú alkalmazást szeretnénk készíteni a Delphi-ben, egyszerűen létrehozunk egy származtatott osztályt a **TThread** (programszál) osztályból. Ezt megtehetjük két féle képpen.

Az első lehetőségünk, hogy manuálisan beírjuk a szükséges programkódot (legjobb egy külön unit-ba).

A második lehetőségünk, hogy miután létrehoztuk az alkalmazást (File – New – VCL Form Application - Delphi for Win32), a Delphi főmenüjéből kiválasztjuk a **File – New – Other...** menüpontot, majd a megnyíló ablakban a **Delphi Projects – Delphi Files** alatt kiválasztjuk a **Thread Object**-ot. Ez után megadjuk annak az osztálynak a nevét, melyet a TThread osztályból szeretnénk származtatni:



Majd rákattintunk az OK gombra. Ezzel létrehoztunk egy új unit-ot, melyben mindjárt szerepel a mi osztályunk vázlata.

```
unit Unit2;
```

```
interface

uses
  Classes;

type
  TVonalSzal = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ ... }

procedure TVonalSzal.Execute;
begin
  { Place thread code here }
end;

end.
```

Nekünk ezek után csak ezt kell kiegészítenünk, illetve módosítanunk. Fontos, hogy a programszál fő részét, tehát azt a programot, amit a programszálnak el kell végeznie az **Execute** metódus implementációjában kell megadnunk.

A létrehozott szálból elérhetjük, módosíthatjuk a vizuális komponenseket (VCL) is. Ha csak olvasni akarjuk valamelyik komponens tulajdonságát, azt megtehetjük az Execute metódusban is. Ha viszont meg szeretnénk változtatni vizuális komponensek bármelyik tulajdonságát, azt nem végezhetjük el közvetlenül az Execute metódusban, mivel a képernyő aktualizálásáért kizárólag az alkalmazás fő programszálja a felelős. A programszálunknak létezik azonban egy **Synchronize** metódusa, melynek meghívásával jelezhetjük a fő

programszálnak, hogy mit szeretnénk elvégezni a vizuális komponenseken (a Synchronize metódus paraméterében azt az eljárást kell megadnunk, amelyben a vizuális komponensekkel való műveletek szerepelnek). A fő programszál a Synchronize meghívása után a megadott műveleteket a komponenseken automatikusan elvégzi.

Amint már említettük, TThread osztály legfontosabb metódusa az **Execute**, amely azt a programot tartalmazza, melyet a szálnak el kell végezni. Ez a metódus felelős azért, hogy TThread **Terminated** tulajdonságát folyamatosan ellenőrizze, és ha ennek a tulajdonságnak az értéke igaz (true), akkor az Execute metódus befejeződik, és ezzel a programszálunk is befejeződik.

A programszálunk (Execute metódus) a szál létrehozása után automatikusan elindul, ha a konstruktor paraméterében false (hamis) értéket adtunk meg. Ha a létrehozáskor a konstruktor paramétereként igaz (true) értéket adunk meg, akkor a programszál létrehozása után a szál felfüggesztett állapotban marad, tehát nem kezdődik el automatikusan az Execute metódus futása. Ebben az esetben a programszál futását a **Resume** metódus meghívásával indíthatjuk el. A futó szálat a **Suspend** metódus meghívásával szüneteltethetjük (függeszthetjük fel). Az Execute metódust soha ne hívjuk meg közvetlenül! Azt, hogy egy programszál éppen fut-e vagy szüneteltetve van, a **Suspended** tulajdonság segítségével állapíthatjuk meg.

A programszál teljes leállítását a **Terminate** metódus segítségével végezhetjük el, amely valójában a **Terminated** tulajdonságot állítja igaz (true) értékre. Az Execute metódusban ezért fontos, hogy időközönként ellenőrizzük a Terminated tulajdonság értékét, és ha ez igaz, akkor fejezzük be a program futását.

A programszálat, miután befejeződött a futása, a memóriából felszabadíthatjuk a már megszokott **Free** metódus segítségével.

Egy másik megoldás a programszál memóriából való felszabadítására, hogy még az elején (pl. létrehozás után) beállítjuk a **FreeOnTerminate** tulajdonság értékét true-ra. Ilyenkor ha a programszál futása befejeződik, automatikusan felszabadul a memóriából (nem kell meghívunk a Free metódust).

4.2. Szakaszok párhuzamos kirajzolása

Készítsünk alkalmazást, amely véletlen helyű és színű szakaszokat fog kirajzolni egy külön programszál segítségével egy image komponensre. Az így megírt programunk a kirajzolás alatt is fog reagálni az eseményekre, pl. a kirajzolás közben is át tudjuk majd helyezni az alkalmazás ablakát, mivel a rajzolás egy külön, a főprogramszállal párhuzamos szálon fut. 015



Hozzunk létre egy új alkalmazást, melyre helyezzünk el egy Image komponenst és két nyomógombot. Az egyik nyomógommbal fogjuk a mi programszálunkat elindítani, a másikkal felfüggeszteni.

Nézzük meg először a programszálunkat tartalmazó unit forráskódját:

```
unit Unit2;

interface

uses
  Classes, Graphics;

type
  TVonalSzal = class(TThread)
  private
    x1,y1,x2,y2: integer;
    Szin: TColor;
  protected
    procedure Execute; override;
    procedure KepFrissitese;
  end;

implementation

uses Unit1;

procedure TVonalSzal.Execute;
begin
  repeat
    x1 := Random(Form1.Image1.Width);
    y1 := Random(Form1.Image1.Height);
    x2 := Random(Form1.Image1.Width);
    y2 := Random(Form1.Image1.Height);
    Szin := Random($FFFFFF);
    Synchronize(KepFrissitese);
  until Terminated = true;
end;
```

```
procedure TVonalSzal.KepFrissitese;
begin
  Form1.Image1.Canvas.Pen.Color := Szin;
  Form1.Image1.Canvas.MoveTo(x1,y1);
  Form1.Image1.Canvas.LineTo(x2,y2);
end;

end.
```

Bevezettünk néhány privát változót (x1, y1, x2, y2, szin) melyek a kirajzolandó vonal helyét és színét határozzák meg.

Azt, hogy a programszálunk pontosan mit végezzen el, az **Execute** metódusban kell megadnunk. Láthatjuk, hogy itt kigenerálunk véletlen számokat, majd a KepFrissitese eljárás segítségével kirajzoljuk a vonalat a képernyőre. Ez addig fog körbe-körbe futni egy repeat..until ciklus segítségével, amíg nem állítjuk le a programszálat a **Terminate** metódussal (ez a metódus állítja be a **Terminated** tulajdonságot, melyet a ciklusban vizsgálunk).

A komponensre való kirajzolást a fő programszálnak kell elvégeznie, ezért a KepFrissitese eljárást a **Synchronize** metódus segítségével végezzük el.

Az alkalmazásunk komponenseinek eseményeihez tartozó forráskód:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, ... , Unit2;

...
```

```

var
  Form1: TForm1;
  Szal: TVonalSzal;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Szal := TVonalSzal.Create(True);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Enabled := False;
  Button2.Enabled := True;
  Szal.Resume;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Button1.Enabled := True;
  Button2.Enabled := False;
  Szal.Suspend;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  if Szal.Suspended then Szal.Resume;
  Szal.Terminate;
  Szal.Free;
end;

end.

```

A programban lesz egy globális objektumunk (Szal), amely TVonalSzal osztály típusú lesz.

A **Form – OnCreate** eseményében létrehozunk az új programszálat (tehát Szal objektumot). A Create konstruktor true

paramétere azt jelzi, hogy a létrehozás után a programszál felfüggesztett állapotban legyen, tehát ne fusson.

A **Button1 – OnClick** eseményében miután beállítottuk a nyomógombok elérhetőségét, a **Resume** metódus segítségével elindítjuk a programszálat.

A **Button2 – OnClick** eseményéhez tartozó eljárásban ennek a programszálnak a futását a **Suspend** metódus segítségével felfüggesztjük.

A **Form – OnDestroy** eseményében a form memóriából való felszabadítása előtt, ha a programszálunk futása fel volt függesztve, akkor elindítjuk, majd a **Terminate** metódus meghívásával a programszálunk futását (Execute eljárást) befejezzük. Végül a **Free** metódus segítségével felszabadítjuk a programszálnak lefoglalt memóriát.

4.3. Szálak szinkronizálása – várakozás egy másik programszálra

A programszálak használatakor sokszor szükségünk lehet a szálak szinkronizálására – például egy szálban kiszámított eredményre szüksége lehet egy másik programszálnak. Ilyenkor a legegyszerűbb szinkronizálási módszer a várakozás a másik programszál futásának befejezésére. Erre készítünk egy példaprogramot.

Az alkalmazásunk tartalmazzon egy Label komponenst és két nyomógombot. Az első nyomógomb megnyomásakor egy kisebb számítást fogunk elvégezni két programszál segítségével. Az egyik programszál a Label-ben levő számhoz hozzáad 1-et, közben a másik

szál várakozik erre az eredményre, majd beszorozza 2-vel. A második nyomógomb csupán az alkalmazás bezárására fog szolgálni. 016



Miután létrehoztuk az alkalmazást, hozzunk létre egy programszálát új unitban (THozzaadSzal), majd ebbe a unitba beírjuk manuálisan a másik programszálunkat is (TSzor2Szal). Ennek a modulnak a programkódja így néz ki:

```
unit Unit2;

interface

uses
  Classes;

type
  THozzaadSzal = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

  TSzor2Szal = class(TThread)
  private
    { Private declarations }
  protected
```

```
    procedure Execute; override;
    procedure LabelFrissites;
  end;
```

```
implementation

uses Unit1, SysUtils;

var szam: int64;

procedure THozzaadSzal.Execute;
begin
  szam := szam + 1;
end;

procedure TSzor2Szal.Execute;
var
  HozzaadSzal: THozzaadSzal;
begin
  szam := StrToInt(Form1.Label1.Caption);
  HozzaadSzal := THozzaadSzal.Create(false);
  HozzaadSzal.WaitFor;
  HozzaadSzal.Free;
  szam := 2 * szam;
  Synchronize(LabelFrissites);
end;

procedure TSzor2Szal.LabelFrissites;
begin
  Form1.Label1.Caption := IntToStr(szam);
end;

end.
```

Láthatjuk, hogy a TSzor2Szal Execute metódusában miután a Label komponensben szereplő számot átírtuk a „szam” változóba, létrehozunk egy THozzaadSzal típusú objektumot. A **WaitFor** metódus segítségével várakozunk ennek a HozzaadSzal-nak a befejeződésére,

majd ezt a szálat felszabadítjuk és a kapott „szam”-ot beszorozzuk kettővel. Végül az így kapott eredményt kiírjuk a Label komponensbe.

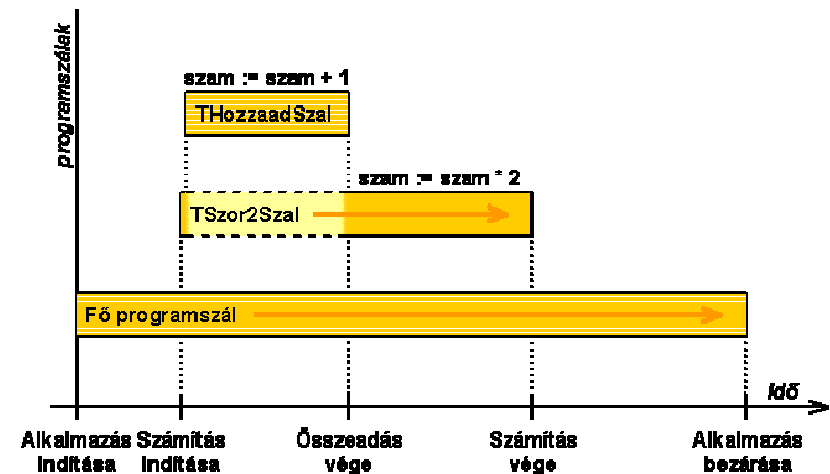
Nézzük most a nyomógombok OnClick eseményeihez tartozó programkódokat:

```
unit Unit1;  
  
interface  
  
uses  
    Windows, Messages, ... , Unit2;  
  
...  
  
var  
    Form1: TForm1;  
    Szor2Szal: TSzor2Szal;  
  
implementation  
  
{$R *.dfm}  
  
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Close;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Szor2Szal := TSzor2Szal.Create(false);  
    Szor2Szal.FreeOnTerminate := true;  
end;  
  
end.
```

A „Szor2Szal” programszál a létrehozása után rögtön elindul (mivel a konstruktor paraméterében „false” értéket adtunk meg).

A létrehozás után beállítottuk a programszál **FreeOnTerminate** tulajdonságát true-ra. Ezzel elértük, hogy a programszál futásának befejezése után automatikusan felszabaduljon a memóriából.

A szálak létrehozását és élettartamát az idő függvényében az alábbi ábra szemlélteti:



4.4. Programszálak prioritása

A programszál létrehozása után megadhatjuk a **Priority** tulajdonság segítségével a programszál futási prioritását. A nagyobb prioritású programszál több processzoridőt kap az operációs rendszertől, így az gyorsabban le tud futni mint a kisebb prioritású. Túl nagy prioritást azonban csak indokolt esetben adjuk egy programszálnak, ugyanis az jóval lelassítja a többi programszál futását.

A prioritás meghatározásánál a következő értékek adhatók meg:

tpIdle	A szál csak akkor indul el, ha a Windows várakozó állapotban van.
tpLowest	A normál prioritásnál két ponttal alacsonyabb.
tpLower	A normál prioritásnál egy ponttal alacsonyabb.
tpNormal	Normál prioritású szál.
tpHigher	A normál prioritásnál egy ponttal magasabb.
tpHighest	A normál prioritásnál két ponttal magasabb.
tpTimeCritical	A legmagasabb prioritású szál.

Készítsünk alkalmazást, melyben megmutatjuk két különböző prioritású programszál párhuzamos működését. Az alkalmazás ablakán két golyót fogunk mozgatni. Mindegyik golyó mozgatását külön szál fogja végezni. Attól függően, hogy a két szálnak milyen a prioritása, az egyik golyó gyorsabban vagy lassabban fog mozogni, mint a másik. 017



A Form-on helyezünk el két Image komponest (golyók) és három Button komponest.

Hozzunk létre egy új programszál a golyók mozgatására. Mindegyik szálnak meg kell jegyeznünk melyik golyót (melyik Image komponest) fogja a szál mozgatni (**FGolyo**) és a golyó (Image komponens) aktuális koordinátáját a Form-hoz viszonyítva (**FXKoord**). Ezek kezdeti értékeinek beállításához készítünk egy konstruktort. Az új konstruktor paramétereként megadjuk az Image komponest és a prioritását a programszálnak.

A programszál forráskódja:

```
unit Unit2;

interface

uses
  Classes, ExtCtrls;

type
  TGolyoSzal = class(TThread)
  private
    FGolyo: TImage;
    FXKoord: integer;
  protected
    procedure Execute; override;
    procedure Kirajzol;
  public
    constructor Create(AGolyo: TImage;
      APrioritas: TThreadPriority);
  end;

implementation

Uses Unit1, Forms;
```



```

constructor TGolyoSzal.Create;
begin
    inherited Create(false);
    FGolyo := AGolyo;
    FXKoord := FGolyo.Left;
    Priority := APrioritas;
end;

procedure TGolyoSzal.Execute;
var
    i: integer;
begin
    repeat
        { bonyolultabb szamitas helyett }
        for i:=0 to 1000000 do
            FXKoord := FXKoord + 1;
            FXKoord := FXKoord - 1000000;
            { ha kimenne a kepernyorol,
              a masik oldalra atrakjuk }
            if FXKoord > Form1.ClientWidth-FGolyo.Width then
                FXKoord := 2;
            { golyo kirajzolasa }
            Synchronize(Kirajzol);
        until Terminated;
    end;

    procedure TGolyoSzal.Kirajzol;
    begin
        { golyo athelyezese }
        FGolyo.Left := FXKoord;
        { tovaabbi esemenyek feldolgozasa }
        Application.ProcessMessages;
    end;

end.

```

A **Create** konstruktorban létrehozzunk az ős konstruktorának meghívásával a programszálat, melyet rögtön futtatunk is (false paraméter). Ezután beállítjuk az FGolyo és FXKoord mezők kezdeti értékeit és a programszál prioritását.

Az **Execute** metódusba tettünk egy ciklust, amely 1000001-szer hozzáad az FXKoord mező értékéhez egyet, majd kivon belőle 1000000-t. Erre azért volt szükség, hogy a programszál végezzen valamilyen hosszab ideig tartó műveletet. Különben a legbonyolultabb művelet a golyó mozgatása lenne, amit a főprogramszál végez el (a Kirajzol metódus szinkronizálásával), így a prioritás nem lenne érzékelhető.

A **Kirajzol** metódus arréb teszi a golyót, majd feldolgozza az alkalmazás további eseményeit. Erre az Application.ProcessMessages parancsra csak azért van szükség, mert ha átállítanánk a prioritást magasra, akkor a golyók mozgatása olyan nagy prioritást kapna, hogy az alkalmazás a többi eseményre nem tudna időben reagálni.

Az alkalmazás eseményeihez tartozó eljárások forráskódja:

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, ... , Unit2;

...

implementation

{$R *.dfm}

var
    g1, g2: TGolyoSzal;

procedure TForm1.FormCreate(Sender: TObject);
begin
    DoubleBuffered := true;
    g1 := TGolyoSzal.Create(Image1, tpLower);
    g2 := TGolyoSzal.Create(Image2, tpLowest);

```

```

end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    if g1.Suspended then g1.Resume;
    if g2.Suspended then g2.Resume;
    g1.Terminate;
    g2.Terminate;
    g1.Free;
    g2.Free;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    g1.Resume;
    g2.Resume;
    Button1.Enabled := false;
    Button2.Enabled := true;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    g1.Suspend;
    g2.Suspend;
    Button1.Enabled := true;
    Button2.Enabled := false;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    Close;
end;

end.

```

Láthatjuk, hogy a két golyó mozgatását végző programszálra két globális változót vezettünk be (**g1**, **g2**). A form létrehozásakor létrehozuk ezt a két programszál is, az első alacsony, a másodikat ennél eggyel alacsonyabb prioritással.

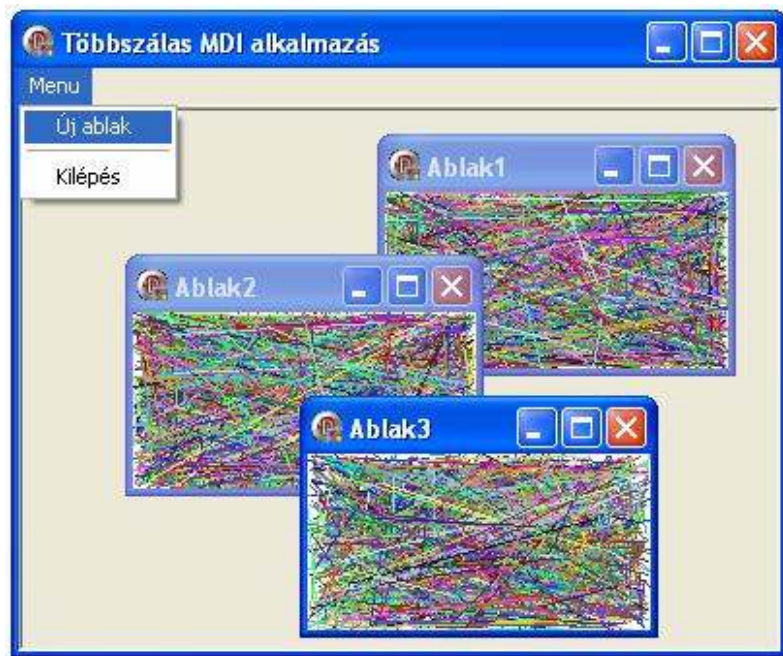
Az egyik gomb szünetelteti (felfüggeszti) mindkét szál futását, a másik elindítja mindkét szálat. A harmadik nyomógomb segítségével kiléphetünk az alkalmazásból.

A Form – OnDestroy eseményében, ha a programszálak éppen felfüggesztett állapotban vannak, akkor elindítjuk őket, majd a Terminate metódus segítségével befejeztetjük a futásukat. Végül felszabadítjuk a számukra lefoglalt memóriát a Free metódus meghívásával.

4.5. Többszálú MDI alkalmazás

Az eddig létrehozott alkalmazásaink mindig csak egy, kettő, maximum három programszál használtak. A következő feladatban olyan MDI alkalmazást készítünk, melynek mindegyik gyermek ablaka külön szálat fog használni, így az alkalmazásunk több szálat is használhat majd.

Készítsünk MDI alkalmazást, amelyben minden MDI-gyermek ablakra egy új szál segítségével párhuzamosan fogunk kirajzolni véletlen helyzetű és színű szakaszokat. **019**



Ehhez először is a Delphi-ben hozzunk létre a már megszokott módon egy új alkalmazást (File – New – VCL Form Application - Delphi for Win32). Erre helyezzünk el egy főmenü (MainMenu) komponenst, melynek állítsuk be a szükséges tulajdonságait. Továbbá ne felejtsük el beállítani a Form **FormStyle** tulajdonságát **fsMDIForm** értékre.

Ezek után hozzuk létre a gyermek ablakot. Ehhez hozzunk létre egy új Form-ot (File – New – Form - Delphi for Win32), melyre helyezzünk el egy Image komponenst. Az Image komponens Align tulajdonságát állítsuk be alClient-re. A Form **FormStyle** tulajdonságát ennél a form-nál **fsMDIChild** értékre állítsuk.

Végül hozzunk létre a már megszokott módon egy Thread Object-et egy új unitban (File – New – Others – Delphi Projects – Delphi Files – Thread Object). Az új programszálunk neve legyen **TSzal**.

Fontos, hogy minden egyes gyermek ablak pontosan tudja melyik programszál fog neki dolgozni, és hasonlóan minden egyes szál tudja, melyik Form-ra (pontosabban melyik Image komponensre) fogja kirajzolni a vonalakat. Ehhez a gyermek ablakban (Form-ban) meg fogjuk adni a hozzá tartozó programszálat, és minden programszálban pedig azt az Image komponenst, amelyre a szakaszokat kirajzolja.

Nézzük először a programszálunkat tartalmazó modult (Unit3):

```
unit Unit3;

interface

uses
  Classes, Graphics, ExtCtrls;

type
  TSzal = class(TThread)
  private
    { Private declarations }
    x1,y1,x2,y2: integer;
    szin: TColor;
    img: TImage;
  protected
    procedure Execute; override;
    procedure Kirajzol;
  public
    constructor Create(iimg: TImage);
  end;

implementation

constructor TSzal.Create;
begin
  inherited Create(false);
```

```

    img := iimg;
    priority := tpLowest;
end;

procedure TSzal.Execute;
begin
    repeat
        x1 := random(img.Width);
        y1 := random(img.Height);
        x2 := random(img.Width);
        y2 := random(img.Height);
        szin := random($FFFFFF);
        Synchronize(Kirajzol);
    until Terminated;
end;

procedure TSzal.Kirajzol;
begin
    img.Canvas.Pen.Color := szin;
    img.Canvas.MoveTo(x1,y1);
    img.Canvas.LineTo(x2,y2);
end;

end.

```

A programszál **x1**, **y1**, **x2**, **y2** mezőiben jegyezzük meg a kirajzolandó vonal koordinátáit, a **szin** mezőben a vonal színét és az **img** mezőben azt az image komponenst, melyre a szakaszokat a szál rajzolni fogja. Ez utóbbit a konstruktor segítségével adjuk meg a programszálnak.

Nézzük most meg, hogy néz ki az MDI-gyermek ablakhoz tartozó programkód (Unit2):

```

unit Unit2;

interface

```

```

uses
    Windows, Messages, SysUtils, Variants, Classes,
    Graphics, Controls, Forms, Dialogs, ExtCtrls, unit3;

type
    TForm2 = class(TForm)
        Image1: TImage;
        procedure FormResize(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure FormClose(Sender: TObject; var
            Action: TCloseAction);
    private
        { Private declarations }
        Szal: TSzal;
    public
        { Public declarations }
    end;

implementation

{$R *.dfm}

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

procedure TForm2.FormCreate(Sender: TObject);
begin
    DoubleBuffered := true;
    Szal := TSzal.Create(Image1);
end;

procedure TForm2.FormResize(Sender: TObject);
begin
    Image1.Picture.Bitmap.Width := Image1.Width;
    Image1.Picture.Bitmap.Height := Image1.Height;
end;

end.

```

A form osztályának private deklarációjába ne felejtsük el beírni a programszálunkat (**Szal:TSzal**).

Ezt a programszál-objektumot a form létrehozásakor a **Form – OnCreate** eseményében hozzuk létre.

A **Form – OnClose** eseményében beállítjuk, hogy a form bezárása után felszabaduljon a számára lefoglalt memória, majd leállítjuk és megszüntetjük a formunkhoz tartozó programszálát is.

A **Form - Resize** eseményében csupán a képünk bitmapjának a méretét állítjuk be, hogy a rajzolás a form átméretezésekor a teljes képre történjen.

Végül nézzük meg az MDI Form-hoz tartozó programkódot is (Unit1):

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    Menu1: TMenuItem;
    jablak1: TMenuItem;
    N1: TMenuItem;
    Kilps1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure jablak1Click(Sender: TObject);
    procedure Kilps1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
end;

var
  Form1: TForm1;
  n: integer;

implementation

{$R *.dfm}

procedure TForm1.Kilps1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.jablak1Click(Sender: TObject);
var
  Form2: TForm2;
begin
  Form2 := TForm2.Create(self);
  Form2.Caption := 'Ablak' + IntToStr(n);
  inc(n);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  n := 1;
end;

end.
```

Ebben a modulban használtunk egy globális **n** változót, amely csupán arra szolgál, hogy a létrehozott gyermek ablak feliratába be tudjuk írni, hányadik ablak jött éppen létre.

Az MDI-gyermek ablakot az alkalmazásunk megfelelő menüpontját kiválasztva hozzuk létre a **Create** konstruktor segítségével. Az új ablak tulajdonosának az MDI Form-ot (**self**) állítjuk be.

5. OLE technológia

Az OLE (Object Linking and Embedding = objektum csatolása és beszúrása) egy érdekes technológia, melyet a Microsoft fejlesztett ki. Ennek segítségével megjeleníthetünk, sőt szerkeszthetünk bitmap-okat a programunkban az nélkül, hogy ezt külön be kéne programoznunk.

Valójában ezt a technológiát használjuk ki akkor is, amikor például a MS Word-be írt dokumentumba egy Paintban rajzolt képet vagy egy MS Excel-ben elkészített táblázatot rakunk be. Ha az ilyen objektumra duplán rákattintunk, akkor az megnyílik szerkeszthető formában a Paint, ill. MS Excel programban.

Az OLE technológia a munkájához ún. **OLE szervereket** használ. Ne gondoljunk itt most külön szerver számítógépekre! Az OLE szerverek a saját gépünkben futó alkalmazások (egyszerűen megfogalmazva), melyek más alkalmazások (nevezzük ezeket OLE klienseknek) részére felkínálják a szolgáltatásaikat és működésüket.

Az alkalmazás (OLE kliens) kihasználhatja valamelyik, a számítógépen elérhető OLE szerveret olyan tevékenységek elvégzésére, melyre saját maga nem képes. Ha van például MS Word szerverünk, amely felkínálja „DOC formátumú dokumentumok megjelenítését”, akkor a mi alkalmazásunkból ezt a szerveret meghívhatjuk és így a saját alkalmazásunkban Word dokumentumot jeleníthetünk meg az nélkül, hogy a programunkban ezt saját magunk implementáltuk volna.

Az **OLE** tehát egy olyan mechanizmus, amely lehetőséget ad az alkalmazásunkban elhelyezni olyan objektumot, amely egy másik alkalmazásban van definiálva.

Jelenleg az OLE 2-es verziója használt a gyakorlatban. Ez lehetőséget ad tehát más alkalmazásban definiált objektum

megjelenítésére a saját programunkban. Ha ezt az objektumot a programunkban szerkeszteni szeretnénk (dupla kattintással aktiválva), akkor vagy egy új ablakban, vagy a programunk belsejében nyílik meg az objektumhoz tartozó alkalmazás (pl. MS Word).

Az OLE-val való munkánk során megkülönböztetünk:

- **Beszúrást (Embedding)** – az objektum fizikailag is az alkalmazásba kerül. Ha például van a merevlemezünkön egy „teszt.doc”, melyet az alkalmazásba beszúrunk, akkor ez a fájl fizikailag át lesz másolva a helyéről az alkalmazásba.
- **Csatolást (Linking)** – az alkalmazásba nem lesz beszúrva fizikailag az objektum, csak egy hivatkozás rá. Az állomány tehát meg fog jelenni az alkalmazásunkban, de fizikailag dolgozni a merevlemezen levő állománnyal fogunk.

5.1. A Delphi és az OLE

Az OLE-vel való munkánk alapkomponense a System palettán található **OleContainer**. Ez a komponens jelképezi az OLE szervereket a Delphi-ben létrehozott alkalmazásban.

Az OleContainer lehetőséget ad bármilyen OLE objektum kiválasztására és elhelyezésére vagy csatolására (linkelésére) az alkalmazásunkban. Az objektumok száma, melyek közül választhatunk, a számítógépen telepített programoktól függ. Ezért az alábbi mintafeladatok közül nem biztos, hogy mindegyik fog működni a számítógépünkön. Ha ugyanis a számítógépen nincs például MS Word,

akkor nem tudunk MS Word dokumentumot megjeleníteni az alkalmazásunkban.

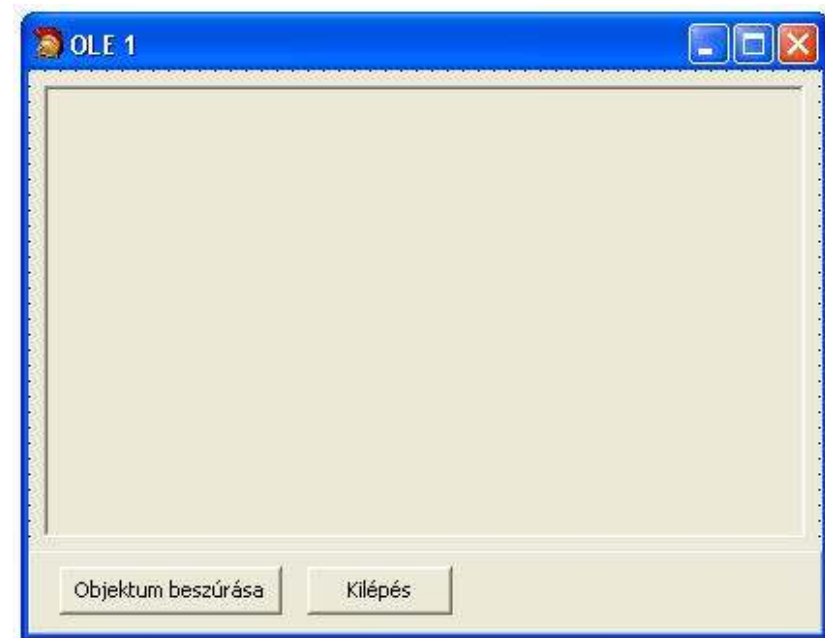
Az **OleContainer** komponenssel való munkánk során mindenekelőtt el fogunk helyezni a formon egy ilyen komponenst, melyben a kiválasztott objektum fog megjelenni. Ez után meghívhatjuk az **InsertObjectDialog** metódust, amely egy dialógusablakot nyit meg, melyben a felhasználó kiválaszthatja a számítógépen levő összes elérhető OLE objektum közül, hogy milyent akar elhelyezni.

Ha az objektumot a programból, tehát nem dialógusablak segítségével szeretnénk beszúrni vagy linkelni, akkor a beszúráshoz az OleContainer komponens **CreateObject**, **CreateObjectFromFile** vagy **CreateObjectFromInfo** metódusai közül választhatunk, csatoláshoz pedig a **CreateLinkToFile** metódust használhatjuk.

5.2. Első OLE-t használó alkalmazásunk

Egy egyszerű példa segítségével megmutatjuk, hogyan lehet az alkalmazásunkban megjeleníteni Word, Excel, Paint, stb. dokumentumokat. **020**

A formra helyezzünk el egy **Panel** komponenst, melynek Align tulajdonságát állítsuk be alBottom-ra. Erre a Panel-ra tegyünk rá két **Button** komponenst, egyet az objektum beszúráására, egyet pedig a programból való kilépésre. Végül a Panel fölé tegyünk be egy **OleContainer** komponenst. Méretezzük át úgy, hogy kitöltse a fennmaradó részt, majd állítsuk be mind a négy Anchors tulajdonságát true-ra. Ezzel az alkalmazásunk tervezésével megvagyunk.



Írjuk meg az egyes eseményekhez tartozó programkódot:

...

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    OleContainer1.InsertObjectDialog;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    { az OLE objektum merete az ablak
      atmeretezeskor változzon }
    OleContainer1.Anchors :=
        [akLeft, akTop, akRight, akBottom];
    { az OLE objektum ne 3D, hanem egyszeru
      feher 2D hatteren legyen }
    OleContainer1.Ctl3D := false;
```

```

end;

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

end.

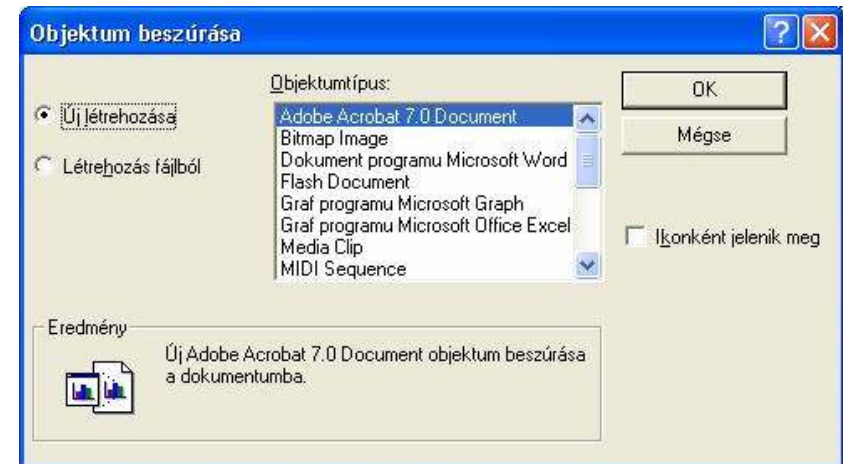
```

A **Form – OnCreate** eseményében beállítottuk az OleContainer Anchors tulajdonságát. Ha ezt már megtettük az alkalmazás tervezésekor, akkor itt nem szükséges. Ezek után beállítottuk az OleContainer Ctl3D tulajdonságát false-ra. Ez sem feltétlenül szükséges, csak a kontainer külalakját változtatja meg. Természetesen ezt a tulajdonságot is beállíthattuk volna tervezési időben.

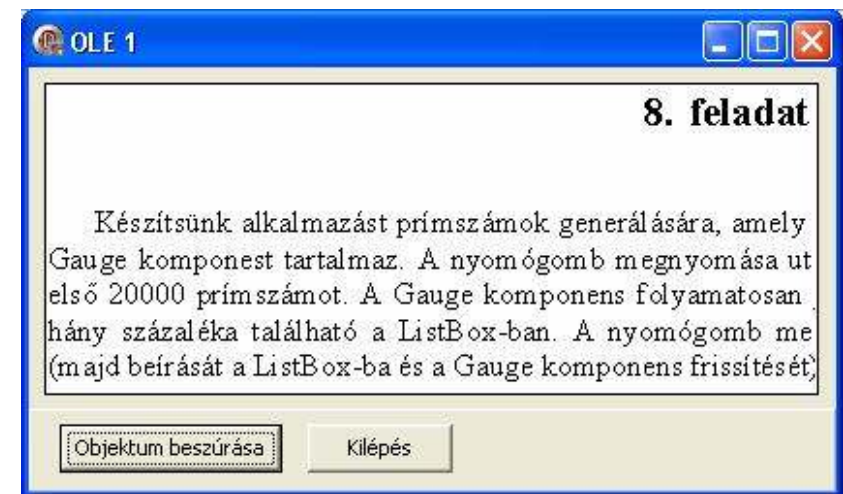
Az alkalmazásunk leglényegesebb része az “Objektum beszúrása” gomb megnyomásakor történik. Ennek a gombnak az OnClick eseményében az OleContainer **InsertObjectDialog** metódusának segítségével megnyitunk egy dialógusablakot, melyben a felhasználó kiválaszthatja, hogy milyen objektumot és milyen formában (beszúrva, csatolva) szeretne az alkalmazásunk OLE kontainerében megjeleníteni.

A csatolás természetesen csak akkor lehetséges, ha nem új objektumot hozunk létre, hanem egy létező fájlból jelenítjük meg az objektumot.

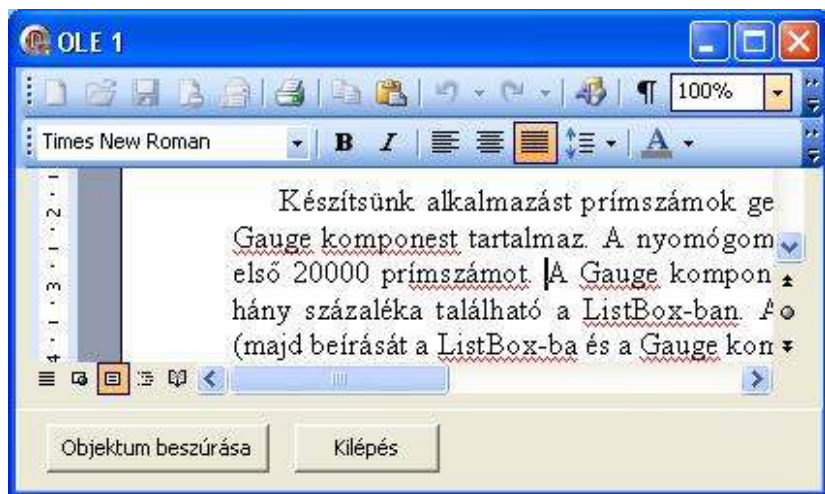
Továbbá ebben a dialógusablakban megadhatjuk azt is, hogy az objektumot eredeti formában (tartalmát) akarjuk megjeleníteni, vagy csak az ikonját szeretnénk megjeleníteni az OLE kontainerben.



Az alkalmazásunk elindítása után próbáljunk meg beszúrni pl. egy Word dokumentumot az alkalmazásunkba a fenti dialógusablak segítségével. Láthatjuk, hogy ez a dokumentum megjelent az alkalmazásunk OLE kontainerében.



Az OLE azonban ennél többre is képes. Ha duplán rákattintunk a dokumentumunkra, akkor azt szerkeszthetjük is – az alkalmazásunk MS Word-é válik.



Menü megjelenítése

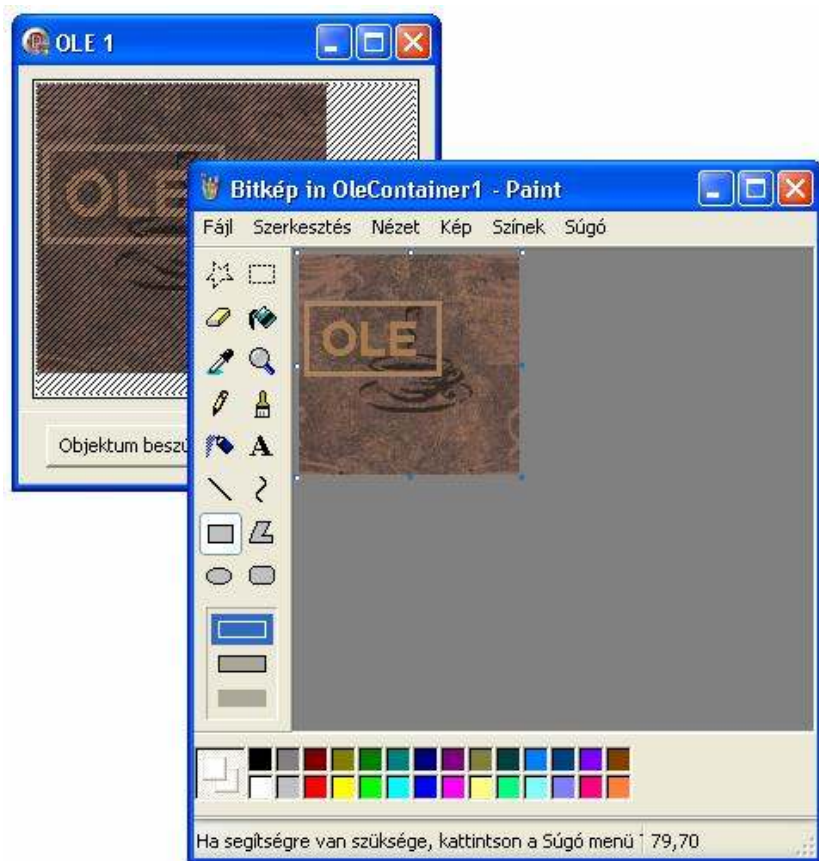
Láthattuk, hogy hiányzik a MS Word főmenüje a dokumentum szerkesztésekor. Ezt nagyon egyszerűen megoldhatjuk. Elég, ha az alkalmazásunkra elhelyezünk egy **MainMenu** komponenst. Így ebben a komponensben automatikusan megjelenik majd a MS Word menüje az OLE objektum aktiválásakor.

Aktiválás új ablakban

Megfigyelhettük azt is, hogy bármilyen dokumentumot (Word, Paint, ...) helyezünk el az OLE kontainerben, aktiváláskor (szerkesztéskor) mindig a Microsoft Word, Paint, ... az alkalmazásunk ablakában jelenik meg. Ezen is tudok változtatni. Elég, ha az OleContainer **AllowInPlace** tulajdonságát beállítjuk false-ra és a dokumentumunkat máris új ablakban szerkeszthetjük majd. Egészítsük ki tehát az előző alkalmazásunk Form – OnCreate eseményéhez tartozó eljárást ennek a tulajdonságnak a beállításával:

```
...
procedure TForm1.FormCreate(Sender: TObject);
begin
    { az OLE objektum merete az ablak
      atmeretezeskor változzon }
    OleContainer1.Anchors :=
        [akLeft, akTop, akRight, akBottom];
    { az OLE objektum ne 3D, hanem egyszeru
      fehér 2D hatteren legyen }
    OleContainer1.Ctl3D := false;
    { duplakattintással az OLE objektumra
      a szerkesztes ne a form-on, hanem
      uj ablakban nyiljon meg }
    OleContainer1.AllowInPlace := false;
end;
```

Most indítsuk el az alkalmazást, majd töltsünk be egy képet. Utána kattintsunk duplán erre a képre a szerkesztéséhez. Láthatjuk, hogy most már külön ablakban szerkeszthetjük a képet, és bármilyen változtatás azonnal megjelenik az alkalmazásunkban látható képen is.



5.3. Az OleContainer tulajdonságai

Az OleContainer-nek sok saját tulajdonsága van, ezek közül felsorolunk egy pár gyakrabban használt tulajdonságot:

Az **AllowInPlace** tulajdonság segítségével beállíthatjuk, ahogy azt már az előző programunkban is tettük, hogy az objektum szerkesztését az alkalmazásunk OLE kontainerén belül (true) vagy külön ablakban (false) szeretnénk elvégezni. Ha a szerkesztést új

ablakban végezzük el, akkor külön ablakban megnyílik az objektumhoz tartozó alkalmazás, melyben az objektumon történő változtatások azonnal megjelennek az alkalmazásunk ablakában (OleContainer-ben) is.

A **SizeMode** tulajdonság segítségével megadhatjuk, hogyan jelenjen meg az objektumunk az OLE kontainerben. Lehetséges értékek:

- **smClip** – alapértelmezett érték – az objektum eredeti méretben jelenik meg, az objektum azon része, amely nem fér bele a komponensbe nem látható.
- **smCenter** – az objektum eredeti méretben jelenik meg, de a komponensben középre lesz igazítva, tehát az előzőhöz képest az objektumnak nem a bal felső része, hanem a közepe lesz látható.
- **smScale** – megváltoztatja az objektum méretét az oldalak arányát betartva úgy, hogy a komponensbe beleférjen.
- **smStretch** – hasonlóan az előzőhöz megváltoztatja az objektum méretét úgy, hogy a komponensbe beleférje, de az oldalak arányát nem tartja be (széthúzza az objektumot az egész komponensre).
- **smAutoSize** – az objektumot eredeti méretében jeleníti meg és a komponens méretét állítja át úgy, hogy az egész objektum beleférjen.

Az OleContainer **State** tulajdonságának kiolvasásával megtudhatjuk, hogy az OLE objektum éppen milyen állapotban van. Lehetséges értékek:

- **osEmpty** – az OLE kontainer üres, nincs benne semmilyen objektum.
- **osLoaded** – az OLE kontainerben van megjelenítve objektum, de nem aktív – a hozzá tartozó OLE szerver nem fut.
- **osRunning** – az OLE kontainerben van megjelenített objektum és ennek OLE szervere fut.
- **osOpen** – az OLE objektum a saját alkalmazásának ablakában (új ablakban) van megjelenítve.
- **osInPlaceActive** – az OLE objektum helyben van aktiválva, de még nincs összekapcsolva az összes menü és eszköztár. Amint a teljes aktiválás befejeződik, az értéke osUIActive-ra változik.
- **osUIActive** – az OLE objektum helyben lett aktiválva, jelenleg is aktív és az össze menü és eszköztár össze lett kapcsolva.

5.4. Kulcsszavak lekérdezése és végrehajtása

Ha meg szeretnénk tudni, milyen kulcsszavakat támogat az OLE kontainerben levő objektum, használhatjuk az **ObjectVerbs** tulajdonságot. Ezeket a kulcsszavakat mint szöveget kapjuk meg, így tartalmazhatnak & jelet is, melyek a gyors billentyűelérést jelölik.

Az alkalmazásunk tartalmazzon egy ListBox komponenst, melyben megjelenítjük az összes objektum által támogatott kulcsszót. Továbbá alkalmazásunkon legyen egy Panel komponens is, melyre elhelyezünk három nyomógombot: egyet az objektum beolvasására,

egyet a kulcsszavak megjelenítésére és egyet a listából kiválasztott kulcsszó végrehajtására. És természetesen végül alkalmazásunk tartalmazni fog egy OleContainer komponenst is. 021

Az alkalmazás tervezésénél először a panelt helyezzük el a formon, ennek Align tulajdonságát állítsuk be alBottom-ra. Erre a panelre helyezzük el a nyomógombokat.



Az objektum beszurása után a "Kulcsszavak" nyomógomb megnyomásával megjelenítjük a ListBox-ban a kulcsszavakat. Ezek közül valamelyiket kiválaszthatjuk, majd a "Parancs végrehajtása" nyomógomb segítségével végrehajthatjuk.

Az alkalmazás nyomógombjaihoz tartozó események programkódja:

...

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    OleContainer1.InsertObjectDialog;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    ListBox1.Items.Assign(OleContainer1.ObjectVerbs);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    OleContainer1.DoVerb(ListBox1.ItemIndex);
end;

end.

```

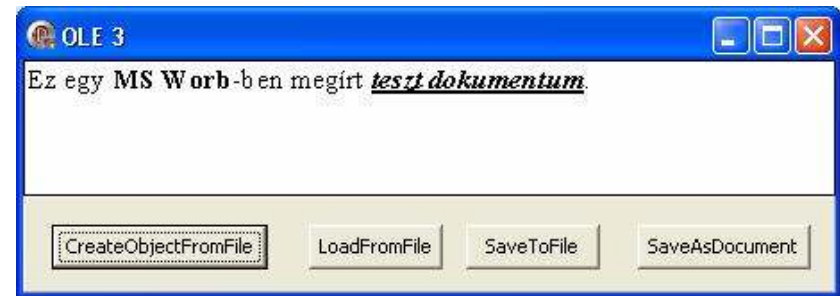
Láthatjuk, hogy a kulcsszavakat az **ObjectVerb** tulajdonság segítségével kérdeztük le.

A kiválasztott kulcsszó végrehajtását a **DoVerb** metódussal végeztük el, melynek paramétereként a kulcsszó sorszámát adtuk meg.

5.5. OLE objektum beolvasása és mentése

A következő alkalmazásunk egy Panel-t, rajta négy nyomógombot és egy OleContainert fog tartalmazni. Az egyes nyomógombok segítségével fogjuk szemléltetni a dokumentum beolvasását, OLE objektum beolvasását fájlból, OLE objektum mentését és a dokumentum mentését fájlba. **022**

A programban az egyszerűség kedvéért mindig a **“teszt.doc”** fájlt fogjuk beolvasni, illetve ilyen nevű állományba fogunk menteni.



Az egyes nyomógombokhoz tartozó események:

```

...

procedure TForm1.Button1Click(Sender: TObject);
begin
    OleContainer1.CreateObjectFromFile(
        ExpandFileName('teszt.doc'), false);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    OleContainer1.LoadFromFile('teszt.doc');
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    OleContainer1.SaveToFile('teszt.doc');
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    OleContainer1.SaveAsDocument('teszt.doc');
end;

...

```

Az első nyomógombnál a **CreateObjectFromFile** segítségével beolvassuk a Word dokumentumot a kontainerbe. Ha a *teszt.doc* állományban valóban Word dokumentum van, akkor ez hiba nélkül végrehajtódik. A paraméterben az `ExpandFileName` függvényt azért kellett használnunk, mivel ennek a módszernek az állomány nevét a teljes útvonallal együtt kell átadnunk. A módszer második paramétere azt adja meg, hogy az objektum csak ikon formájában jelenjen meg (true) vagy látható legyen a tartalma (false).

Ha a *teszt.doc* állományban nem Word dokumentum van, hanem OLE objektum, akkor azt csak a **LoadFromFile** módszer segítségével tudjuk beolvasni.

Az OLE kontainerben megjelenített objektumot a **SaveToFile** segítségével menthetjük el úgy, mint Word típusú OLE objektum.

A **SaveAsDokument** módszerrel az objektumot elmenthetjük mint Word dokumentum.

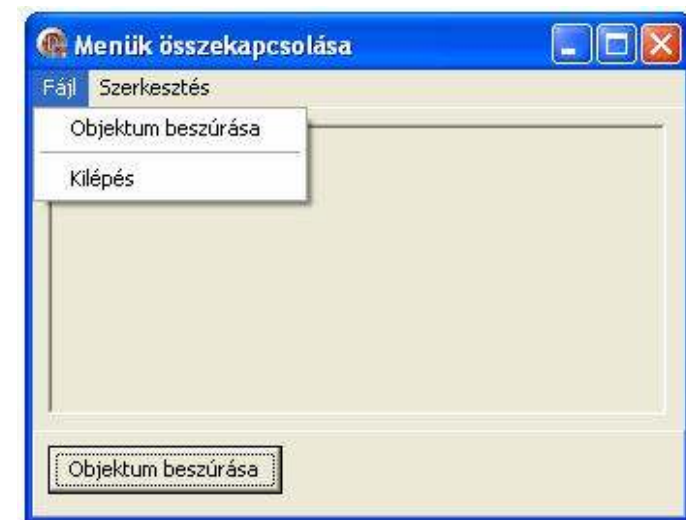
Lehet hogy egy kicsit furcsának tűnik, miért van ilyen nagy különbség a hagyományos állomány (Word dokumentum, bitmap, ...) között és az OLE objektumot tartalmazó állomány formátuma között, amikor az első példánkban minden gond nélkül be tudtunk olvasni dokumentumot az OLE kontainerbe. Ez azért volt, mivel ott a beolvasást az `InsertObjectDialog` módszer segítségével hajtottuk végre, amely automatikusan elvégezte a dokumentum átalakítását OLE objektummá.

5.6. Menük összekapcsolása

Az első példánkban láthattuk, hogy ha elhelyezünk egy MainMenu komponenst a form-on, akkor az objektum szerkesztésekor

ebben megjelennek az OLE szerverhez tartozó menüpontok. De mi van akkor, ha az alkalmazásunkban is szeretnénk saját menüt kialakítani? Erre nézzünk most egy mintafeladatot. 024

Hozzunk létre egy új alkalmazást, melyre helyezzünk el egy Panel komponens, arra egy nyomógombot, amely az objektum beolvasására fog szolgálni. Továbbá tegyünk a formra egy OleContainer és egy MainMenu komponenst. Állítsuk be a komponensek tulajdonságait, majd alakítsuk ki az ábrán látható menüszerkezetet a MainMenu komponensnél.



Ha most elindítjuk a programunkat és beszúrunk valamilyen objektumot, majd duplán rákattintunk a szerkesztéshez, akkor a mi menünk is megmarad és az objektum OLE szerverének menüje is elérhető lesz.



Észrevehetjük, hogy ez azért nem mindig a legjobb megoldás, ugyanis most például két Szerkesztés menüpontunk van – a mienk és az OLE szerveré is. Ebben az esetben talán jobb lenne, ha a mi menünknek a Szerkesztés menüpontja nem lenne látható.

Ezzel a problémával a Delphi-ben könnyen boldogulhatunk. A menüpontoknak ugyanis van **GroupIndex** tulajdonságuk, melyet eddig a menüpontok logikai csoportokba sorolására használtunk.

Most ennek a tulajdonságnak egy további felhasználásával ismerkedhetünk meg. A **GroupIndex** segítségével azt is befolyásolhatjuk, hogy a menük összekapcsolásakor melyik menüpontunk legyen látható és melyik nem:

- Az alkalmazásunk főmenüjének azon menüpontjai, melyeknek GroupIndex értéke 0, 2, 4, ... mindig láthatóak lesznek és nem lesznek felülírva az OLE szerver menüjével (ez a helyzet állt elő az előző példában is, ahol a GroupIndex értéke mindegyik menüpontnál 0 volt).
- Az alkalmazásunk főmenüjének azon menüpontjai, melyeknek GroupIndex értéke 1, 3, 5, ..., az OLE szerver menüjével felül lesznek írva, tehát amikor az OLE szerver menüje látható lesz (objektum szerkesztésekor), akkor az alkalmazásunknak ezen menüpontjai nem lesznek megjelenítve.

Próbáljuk meg most átállítani az alkalmazásunk MainMenu komponensében a Szerkesztés menüpont GroupIndex tulajdonságát 1-re. Ha így elindítjuk az alkalmazásunkat, akkor az objektum szerkesztésénél a mi Szerkesztés menünk már nem lesz látható.



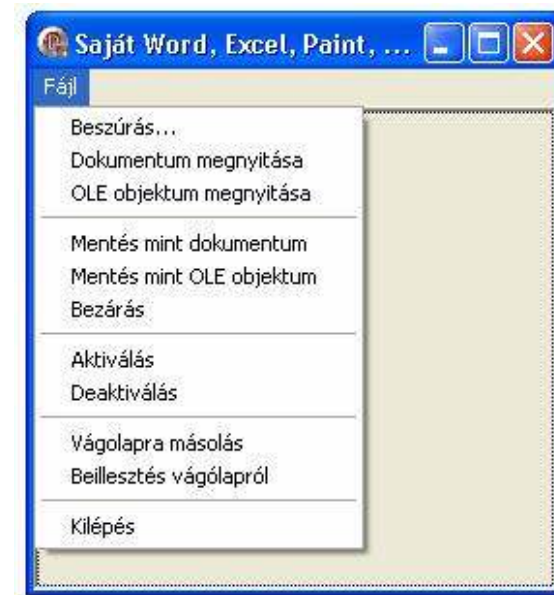
Hasonlóan a menükhöz, az eszköztárak is összekapcsolhatók. Alapértelmezett beállításoknál az OLE server eszköztára felülírja az alkalmazásunk eszköztárát. Ha ezt el szeretnénk kerülni, állítsuk be az alkalmazásunk eszköztárának **Locked** tulajdonságát True értékre.

5.7. Saját Word, Excel, Paint, ...

Készítsünk az OLE-t felhasználva egy alkalmazást, amelyben Word, Excel, Paint, stb. dokumentumokat tudunk megnyitni és szerkeszteni, majd elmenteni is. 023

Az alkalmazásunk indításakor egy **OleContainer**-t és egy **MainMenu**-t fog tartalmazni. Ennek a MainMenu komponensnek csak

egy főmenüpontja lesz, a File. A többi menüpontot a beolvasott objektumhoz tartozó OLE server fogja berakni.



Az alkalmazásra helyezzünk el még egy **OpenDialog** és egy **SaveDialog** komponenst a fájlok beolvasásához és mentéséhez (pontosabban a beolvasandó és elmentendő állomány nevének és helyének meghatározásához).

A tervezéskor állítsuk be az OleContainer Align tulajdonságát alClient-ra, hogy az OLE kontainer az egész formon jelenjen meg. Továbbá ne felejtsük el megadni az OleContainer SizeMode tulajdonságát smScale-ra. Ezzel elérjük, hogy az egész dokumentum látható lesz a kontainerben (arányosan lekicsinyítve vagy felnagyítva a kontainer méretéhez).

A menüpontok OnClick eseményeihez tartozó eljárások:

```
...

procedure TForm1.Beszrs1Click(Sender: TObject);
begin
    OleContainer1.InsertObjectDialog;
end;

procedure TForm1.DokumentummegnyitsalClick(Sender:
                                           TObject);
begin
    if OpenFileDialog1.Execute then
        OleContainer1.CreateObjectFromFile(
            OpenFileDialog1.FileName, false);
end;

procedure TForm1.OLEobjektummegnyitsalClick(Sender:
                                           TObject);
begin
    if OpenFileDialog1.Execute then
        OleContainer1.LoadFromFile(OpenDialog1.FileName);
end;

procedure TForm1.Mentsmintdokumentum1Click(Sender:
                                           TObject);
begin
    if (OleContainer1.State <> osEmpty) and
        (SaveDialog1.Execute) then
        OleContainer1.SaveAsDocument(
            SaveDialog1.FileName);
end;

procedure TForm1.MentsmintOLEobjektum1Click(Sender:
                                           TObject);
begin
    if (OleContainer1.State <> osEmpty) and
        (SaveDialog1.Execute) then
        OleContainer1.SaveToFile(
            SaveDialog1.FileName);
end;

procedure TForm1.Bezrs1Click(Sender: TObject);
```

```
begin
    OleContainer1.DestroyObject;
end;

procedure TForm1.Aktivls1Click(Sender: TObject);
begin
    if OleContainer1.State <> osEmpty then
        OleContainer1.DoVerb(ovShow);
end;

procedure TForm1.Deaktivls1Click(Sender: TObject);
begin
    if OleContainer1.State <> osEmpty then
        OleContainer1.DoVerb(ovHide);
end;

procedure TForm1.Vgolapramsols1Click(Sender:
                                       TObject);
begin
    if OleContainer1.State <> osEmpty then
        OleContainer1.Copy;
end;

procedure TForm1.Beillesztsvglapr11Click(Sender:
                                           TObject);
begin
    if OleContainer1.CanPaste then
        OleContainer1.Paste;
end;

procedure TForm1.Kilps1Click(Sender: TObject);
begin
    Close;
end;

end.
```

Az alkalmazásunkat még lehetne tovább szépíteni, el lehetne játszani a menüpontok Enabled tulajdonságaival, hogy ne lehessen például aktiválni az objektumot, ha a kontainer üres.

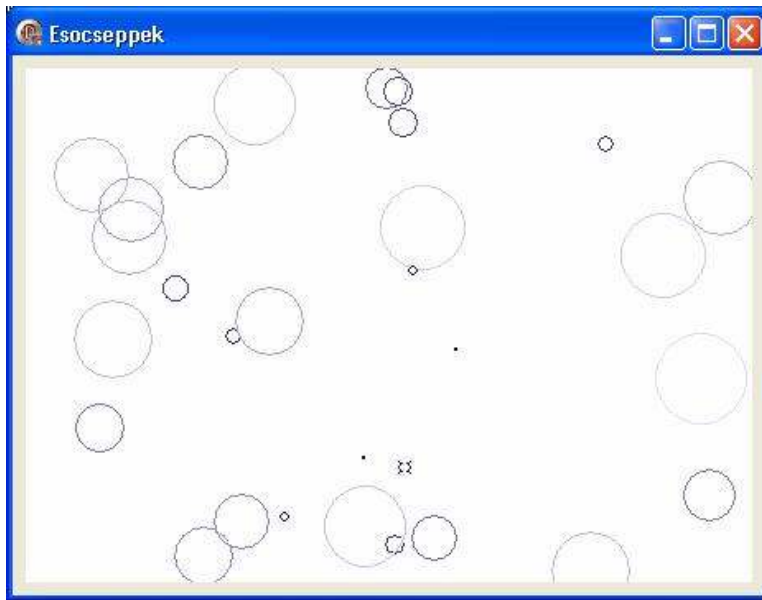
6. OLE Automation

XXXXXX

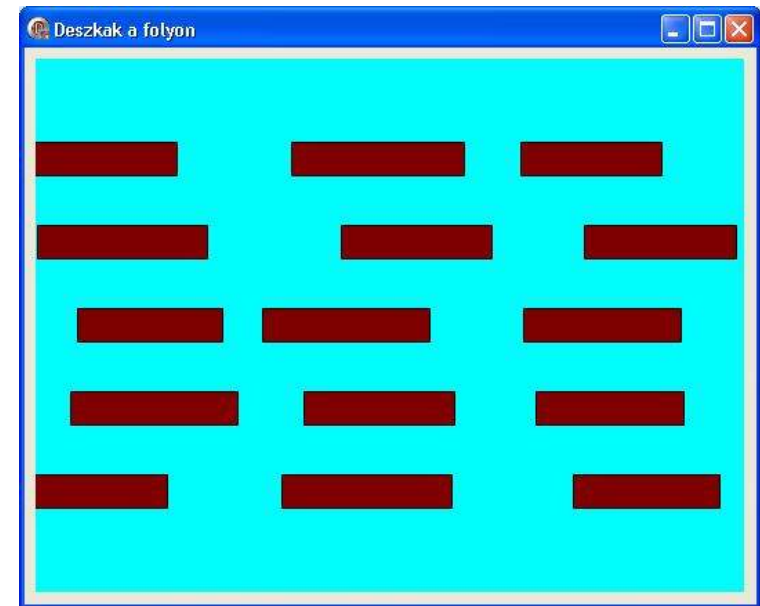
Gyakorlatok:

1. Készítsünk egy **TCsepp** osztályt, amely egy esőcseppet (kört) rajzol ki a megadott Image komponensre. A TCsepp osztálynak írjuk meg a **konstruktorát**, mely csak egy paramétert tartalmazzon – annak az Image (*ExtCtrls unitban található*) komponensnek a nevét, ahová az esőcseppet ki akarjuk rajzolni. A konstruktor generáljon ki egy véletlenszerű koordinátát ezen a képen (Image-en) és egy véletlen sugarat (0-tól 29-ig). Majd rajzolja ki az esőcseppet erre a koordinátára. Az osztály tartalmazzon még egy **kirajzol** és egy **letöröl** eljárást, amely kirajzolja a kört az objektumhoz tartozó koordinátára az objektumhoz tartozó sugárral. A kirajzolást bsClear (*graphics unitban található*) ecsetstílussal és RGB(sugár*8, sugár*8, 100+sugár*5) (*windows unitban található*) kék színárnyalatú körvonallal végezzük. Az osztálynak legyen még egy **növekszik** metódusa, amely letörli az esőcseppet, növeli a sugarát, majd megnézi hogy a sugár nem érte-e el a 30 pixelt. Ha elérte, akkor beállítja 0-ra és új koordinátákat generál ki az objektumnak. Végül kirajzolja az esőcseppet.

Ezt az osztályt felhasználva készítsünk egy programot, amely megjelenít 30 esőcseppet és 25 század-másodpercenként növeli azok nagyságát (amíg nem érik el a 30-at, utána egy másik helyen 0 sugárral jelennek meg). 004

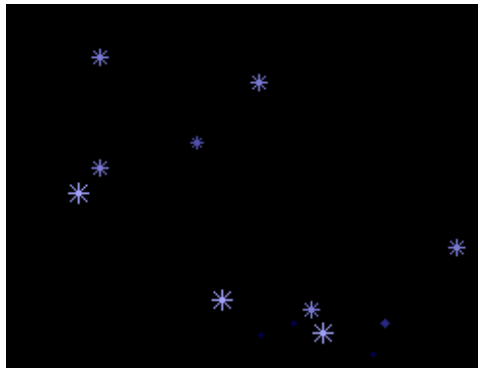


2. Készítsünk egy **TDeszka** osztályt, majd ennek segítségével azt az alkalmazást, melyben véletlen hosszúságú deszkák úszkálnak a vízben. Az első sor balra, a második jobbra, a harmadik megint balra, a negyedik megint jobbra, az ötödik sor pedig ismét balra ússzon. Ügyeljünk arra, hogy két egymás melletti deszka között mindig legyen egy kis hely. Az egyik szélén kiúszó deszkák a másik szélén ússzanak be (ne hirtelen jelenjenek meg). **003**



3. Készítsünk **TCsillag** osztályt, amely kirajzol egy véletlen méretű (pl. 1-től 5-ig) csillagot. Az osztálynak legyen egy olyan metódusa, melynek meghívásával a csillag eggyel nagyobb méretű és világosabb kék színű lesz. Ha eléri az 5-ös méretet, akkor tűnjön el és egy új, véletlen helyen jelenjen meg 1-es mérettel és sötétkék színnel.

A TCsillag osztály segítségével készítsünk egy képernyővédőt, amely teljes képernyőn kirajzol 100 csillagot, majd ezek méretét és fényességét a megírt metódus segítségével növeli (ha valamelyik csillag eléri a maximum méretét, akkor egy másik helyen jelenjen meg kis méretben). Az alkalmazás bármelyik billentyű megnyomásával fejeződjön be. **013**

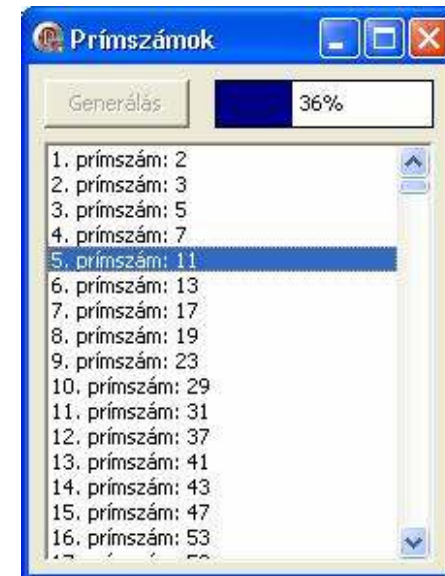


4. Készítsünk egy dinamikus csatolású könyvtárat (DLL-t), amely tartalmaz egy „prímszám” függvényt. Ez a függvény a paraméterében megadott egész számról döntse el, hogy az prímszám-e és ettől függően adjon vissza igaz vagy hamis értéket.

A megírt DLL-t felhasználva készítsük el az alábbi alkalmazást, amely egy nyomógomb megnyomásakor megvizsgálja, hogy az Edit komponensben megadott szám prímszám-e. 014



5. Készítsünk alkalmazást prímszámok generálására, amely egy nyomógombot, egy ListBox-ot és egy Gauge komponest tartalmaz. A nyomógomb megnyomása után a program a ListBox-ba generálja ki az első 20000 prímszámot. A Gauge komponens folyamatosan jelezzze, hogy a 20000 prímszámnak eddig hány százaléka található a ListBox-ban. A nyomógomb megnyomása után a prímszámok generálását (majd beírását a ListBox-ba és a Gauge komponens frissítését) egy külön programszámban végezzük el! Próbáljuk úgy megírni az algoritmust, hogy az a számok generálását minél hamarabb elvégezze. 018



6. xxxxxx

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: **Programozzuk 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