7. Objektově Orientované Programování

1. Úvod
2. Obalení
3. Dědičnost
4. Polymorfismus
5. Statické a virtuální metody
6. Dynamické objekty

7.1 Úvod

Objektově orientované programování (dále jen OOP) představuje v dnešní době novou metodu vytváření programů oproti klasické metodě strukturovaného programování. Metody OOP napodobují vzhled a chování objektu z reálného světa s možností velké abstrakce. Přínosem OOP je také větší strukturovanost a modularita vytvářeného programu.

Objektově orientovaný přístup tvorby programu je charakterizován třemi základními vlastnostmi:

7.2 Obalení

Obalení (zapouzdření) – encapsulation: je realizováno novým datovým typem – **objekt**, který vzniknul kombinací standardního datového typu záznam (record) a datového typu procedura (funkce). Obsahuje tedy kromě datových položek také řídící struktury – metody. Oním obalením tedy mužeme rozumět obalení datových položek řídícími strukturami – metodami, které zajišťují přístup k datovým položkám. Dobrý programátor by měl připravit dostatečný počet metod, aby uživatel neměl potřebu přímo přistupovat k datovým položkám.
Příklad:

```
type
  MujNovyObjekt = object
                    X, Y, Z : integer;        - datová položka
                    Velikost : real;          - datová položka
                    procedure
ZjistiPozici;                                 - metoda
                    function
ZjistiVelikost;                               - metoda
                  end;
```

Jak vidíme z výpisu části programu, je datový typ objekt označen vyhrazeným slovem **object** a jeho syntaxe je obdobná jako syntaxe datového typu záznam.

V předchozím výpisu byla provedena nepřesnost vůči definici obalení. Uživatel totiž nedisponuje žádnou metodou, která by vložila počáteční hodnoty do proměnných X, Y a Z.

Musíme tedy takovou metodu dodefinovat, aby definice předchozího objektu byla kompletní a odpovídala definici obalení. Výsedek bude vypadat následovně:

```
type
  MujNovyObjekt = object
                    X, Y, Z : integer;
                    Velikost : real;
                    procedure Init(Xi, Yi, Zi : integer);
                    procedure ZjistiPozici;
                    function ZjistiVelikost;
                  end;
```

Takto upravená definice objektu již vyhovuje definici obalení.

Nabízí se zde ještě jeden způsob, jak nepovolit v případech, kdy je to nezbytně nutné, přístup přímo k datovým položkám. Můžeme to provést tím způsobem, že všechny datové položky, které chceme takto upravit, přesuneme do části **private**. Ilustruje to následující příklad:

```
type
  MujNovyObjekt = object
                    procedure Init(Xi, Yi, Zi : integer);
                    procedure ZjistiPozici;
                    function ZjistiVelikost;
                  private
                    X, Y, Z : integer;
                    Velikost : real;
                  end;
```

Metody, které jsme použili v předchozí definici objektu, zde uvádějí pouze své hlavičky. Jejich deklarace následuje až za definicí objektu. V této deklaraci je kromě identifikátoru metody uveden ještě identifikátor objektu, který je oddělen od identifikátoru metody tečkou. Ukážeme si to na deklaraci metody *Init*, která má svou hlavičku v předchozí definici objektu *MujNovyObjekt*.

```
procedure MujNovyObjekt.Init(Xi, Yi, Zi : integer);
begin
  X := Xi;
  Y := Yi;
  Z := Zi;
end;
```

V těle metody jsou všechny datové položky objektu přímo přístupné bez předpony jména objektu. Chceme tím říci to, že použitím identifikátoru *MujNovyObjekt* v hlavičce metody jasně víme, kterého objektu jsou dané datové položky. Pokud by mělo dojít ke konfliktu, který může nastat, pokud se v metodě současně objeví datové položky stejných jmen dvou různých objektů, můžeme pro odlišení použít automaticky předávaný identifikátor *Self*, což je ukazatel na daný objekt, kterému datové složky přísluší. Zápisem *Self.X* nebo *Self.Y* tedy jednoznačně pojmenujeme a odlišíme datovou položku od jiné datové položky jiného objektu.

7.3 DĚDIČNOST

Dědičnost – inheritance: umožňuje vytvářet nové objekty jako potomky již existujících

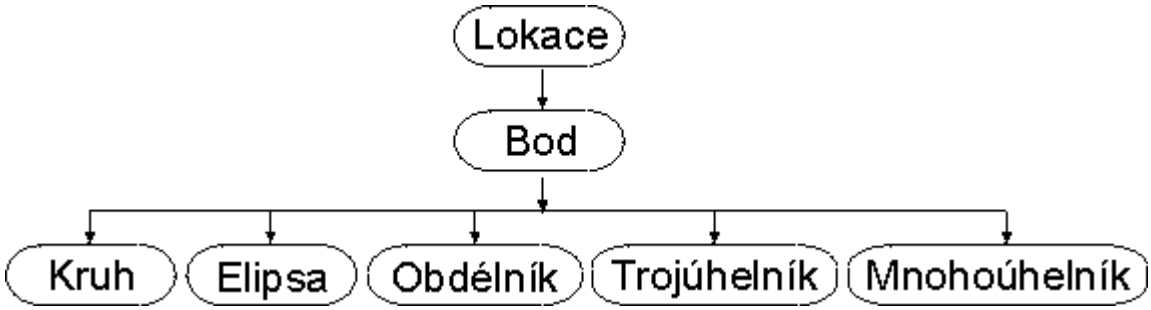
objektů – předků, přebírat od nich datové položky a metody a modifikovat je či upřesňovat. Takto vzniká stromová struktura objektů. Demonstrovat si ji můžeme na následujícím příkladu. Nadefinujeme si nový objekt Lokace, který obsahuje pouze proměnné pro uložení aktuální pozice. Poté následuje definice dalšího objektu, a to objektu Bod, který je potomkem objektu Lokace a přidává navíc ještě vlastnost viditelnosti objektu. Poté již následují definice obrazců, které dědí vlastnosti objektu Bod, neboť se z bodů skládají nebo jsou jimi tvořeny.

```
type Lokace = object
                X : integer;
                Y : integer;
              end;

      Bod = object(Lokace)
              Stav : boolean;
            end;

     Kruh = object(Bod)
              Polomer : integer;
              Barva : word;
              procedure Init(Xi, Yi, Polomer : integer);
              procedure Zobraz;
              procedure Smaz;
            end;
```

atd. Celou hierarchii dědičnosti si můžeme ukázat pomocí diagramu:



Jeden objekt - předek může mít mnoho potomků, avšak jeden potomek může mít pouze jednoho předka.

7.4 POLYMORFISMUS

Polymorfismus (mnohotvarost) - polymorphism: je úzce spjat s předchozí vlastností OOP, a to s dědičností.

Polymorfismus je vlastnost OOP, která umožňuje pojmenovat metodu jedním jménem a tato metoda může být společná pro různé objekty ve stromové hierarchii, i když pro každý objekt v této hierarchii se bude chovat různě. Jinak řečeno: polymorfismus umožňuje, aby každý objekt ve stromové hierarchii objektů (viz obrázek výše) volal metodu se stejným jménem. Přitom může mít každý objekt tuto metodu jinak definovanou. Výsledkem je, že pro každý objekt ve stromové hierarchii bude mít volání metody jinou odezvu, avšak pro každý objekt vždy tu správnou.

Práce s objekty v programu je stejná jako s datovým typem záznam. To znamená, že pokud budeme přistupovat k datovým položkám či k metodám objektu, použijeme tečkovou konvenci nebo příkaz **with**. Ukážeme si to na následujícím výpisu.

```
var MNO : MujNovyObjekt;
   i : integer;

begin
  i := 0;
  MNO.Init(i, i+1, i+2);
  with MNO do
  begin
    inc(i);
    ZjistiVelikost;
  end;
```

7.5 STATICKÉ A VIRTUÁLNÍ METODY

Všechny metody, které jsme si uváděli výše, jsou *statické metody*. Mužeme je přirovnat ke statickým proměnným. Překladač jim vyhradí místo v paměti již v době překladu a také řeší všechny vztahy mezi metodami a objekty již v době překladu. Těmto vztahům se říká **early binding** - *brzká vazba*. Díky těmto vlastnostem jsou statické metody velmi rychlé při jejich vykonávání a jejich kód je velice efektivní.

Avšak právě díky statickým metodám může dojít k problému, který si ukážeme na následujícím příkladě.

```
program Kresleni;

type Lokace = object
                    X : integer;
                    Y : integer;
                  end;

        Bod = object(Lokace)
                  Stav : boolean;
                  procedure Init;
                  procedure Zobraz;
                  procedure Smaz;
                end;

       Kruh = object(Bod)
                  Polomer : integer;
                  Barva : word;
                  procedure Init;
                  procedure Zobraz;
                  procedure Smaz;
                end;

var B : Bod;
    K : Kruh;

procedure Vykresli(Co : Bod);
begin
```

```
  Co.Zobraz;
end;

BEGIN
  Vykresli(K);
END.
```

Trošku jsme, vzhledem k nastínění problému, upravili definice objektů *Bod* a *Kruh*. V hlavním programu požadujeme, aby procedura *Vykresli* vykreslila objekt, který je určen jako parametr této procedury. Program je po syntaktické stránce správný, poněvadž za formální parametr *Co* typu *Bod* můžeme dosadit skutečný parametr *K* typu *Kruh*. Problém nastává až v tomto místě. Protože metoda *Zobraz* je statická metoda, překladač určuje její adresu již v době překladu. Překladač tedy při překladu zjistí, že formální parametr *Co* je typu *Bod* a dosadí tedy při překladu volání metody *Zobraz* adresu metody *Bod.Zobraz*. I když bude skutečným parametrem *Kruh*, bude volána metoda objeku *Bod*.

Tento problém řeší až *virtuální metody*. Virtuální metody nabízejí pro řešení těchto situací možnosti vytvářet vazby mezi objekty a jejich metodami až v době výpočtu. Těmto vazbám se říká **late binding** - *pozdní vazba*. Tyto metody jsou pomalejší při vykonávání než metody statické.

Virtuální metody řeší pojem zvaný polymorfismus. Díky těmto metodám můžeme pojmenovat jedním jménem metodu, která je stejná pro všechny objekty ve stromové hierarchii, avšak pro každý objekt má tato metoda jinou implementaci.

A nyní již konkrétněji. Virtuální metoda se označuje v objektu direktivou **virtual**. Adresa této metody je uložena v **tabulce virtuálních metod** (dále jen VMT - z anglického Virtual method table). Potřebujeme-li vyvolat nějakou metodu, její adresa je nalezena právě v této tabulce.

VMT musí být naplněna ještě předtím, než se s objektem začne vůbec pracovat. K naplnění tabulky VMT slouží speciální metoda, zvaná **konstruktor**. V definici objektu se používá místo vyhrazeného slova **procedure** slovo **constructor**. Konstruktor inicializuje mechanismus virtuálních metod. Konstruktor se musí vyskytovat v definici každého objektu, který využívá alespoň jednu virtuální metodu. Než je jakákoliv virtuální metoda daného objektu použita, je nutné spustit metodu konstruktoru. Pokud bychom to neudělali, došlo by k zablokování programu. Metoda, definovaná v objektu jako virtuální, musí být definovaná i ve všech potomcích tohoto objektu jako virtuální.

Upravíme-li tedy náš předchozí jednoduchý program pomocí virtuálních metod, bude již pracovat správně a metoda Zobraz v proceduře Vykresli bude volána ze správného objektu.

```
program Kresleni;

type Lokace = object
                  X : integer;
                  Y : integer;
              end;

        Bod = object(Lokace)
                  Stav : boolean;
                  constructor Init;
                  procedure Zobraz; virtual;
```

```
                    procedure Smaz;
                  end;

        Kruh = object(Bod)
                    Polomer : integer;
                    Barva : word;
                    constructor Init;
                    procedure Zobraz; virtual;
                    procedure Smaz;
                  end;

var B : Bod;
    K : Kruh;

procedure Vykresli(Co : Bod);
begin
  Co.Zobraz;
end;

BEGIN
  K.Init;
  Vykresli(K);
END.
```

7.6 DYNAMICKÉ OBJEKTY

Stejně tak jako proměnné mohou být statické či dynamické, tak i instance objektů mohou být statické (jak jsme si ukázali výše) i dynamické. Abychom se však nezmýlili v pojmech. Slovo statické se v tomto případě nevztahuje na metody objektu. Statické objekty mohou obsahovat jak statické, tak i virtuální metody. Stejně i dynamické metody mohou obsahovat statické i virtuální metody.

Dynamickým metodám je při jejich vzniku vymezen paměťový prostor haldy a odkazujeme se na ně pomocí ukazatele. Při práci s instancí dynamického objektu musíme stejně jako při práci s dynamickou proměnnou přidělit této instanci volnou paměť pomocí procedury *New*. Stejně tak pokud instance objektu obsahuje byť jedinou virtuální metodu, musíme nejprve zinicializovat VMT pomocí metody *constructor*. Metodu, definovanou v daném objektu, spouštíme v programu tak, že za jménem instance objektu následuje operátor nepřímé adresace **^** s tečkou a dále pak jméno volané metody. Na konci programu musíme paměť přidělenou objektu opět uvolnit. Toho docílíme pomocí nám již známé procedury *Dispose*.

```
type Kruh = object(Bod)
                Polomer : integer;
                Barva : word;
                constructor Init;
                procedure Zobraz; virtual;
                procedure Smaz;
              end;

var K : ^Kruh;

BEGIN
  New(K);
```

```
  K.Init;
   ...
  K^.Zobraz;
   ...
  Dispose(K);
END.
```

Pomocí procedury *Dispose* jsme zrušili instanci dynamického objektu z paměti. Někdy však může nastat situace, kdy objekt uvnitř sebe obsahuje jiné dynamické struktury, které při rušení objektu z haldy je třeba zrušit také. Takovou strukturou, která ruší z haldy všechny dynamické struktury objektu včetně instance objektu samého, je metoda **destruktor**. V definici objektu se používá vyhrazené slovo **destructor**.

```
type Kruh = object(Bod)
                Polomer : integer;
                 Barva : word;
                constructor Init;
                 destructor Done; virtual;
                procedure Zobraz; virtual;
                 procedure Smaz;
               end;
```

Destruktor mohou dědit všichni potomci daného objektu. Je vhodné definovat tuto matodu jako virtuální, může se takto zajistit správné uvolnění paměti přidělené instanci dynamického objektu.

U polymorfních objektů, tedy u takových objektů, kdy předek i potomek vlastní metody stejných názvů, ale jiných implementací (viz úvodní kapitola o polamorfismu), se může stát, že polymorfní objekt může být přiřazen instanci typu předka, avšak obě instance v paměti zaujímají různý prostor. Procedura *Dispose* neví, kolik paměti se má uvolnit. Proto tuto proceduru rozšiřujeme o metodu destruktoru, který přesně ví, která paměť se má uvolnit, protože si to destruktor dokáže zjistit z VMT. Až poté lze uvolnit paměť po instanci polymorfního dynamického objektu. Rozšířená procedura pak vypadá takto:

```
Dispose(K, Done);
```

Zpět Obsah Vpřed