Anotace

- Objektové programování.

Třídy a objekty

- Co je objekt?

Třídy a objekty

- Co je objekt?
- Co je třída?

Třídy a objekty

- Co je objekt?
- Co je třída?
- Téměř všechno kolem nás je objekt.

Třídy a objekty

- Co je objekt?
- Co je třída?
- Téměř všechno kolem nás je objekt.
- Třída je vlastně typem jednotlivých objektů.

Třídy a objekty

- Co je objekt?
- Co je třída?
- Téměř všechno kolem nás je objekt.
- Třída je vlastně typem jednotlivých objektů.
- Objekty mají různé vlastnosti (atributy) a schopnosti něco dělat (metody).

Definice tříd

- Jelikož objekty jsou instance jednotlivých tříd, nemá smysl definovat jednotlivé objekty, ale definujeme třídy.

Definice tříd

- Jelikož objekty jsou instance jednotlivých tříd, nemá smysl definovat jednotlivé objekty, ale definujeme třídy.
- Řekneme, že prvek má být typu `object`.

Definice tříd

- Jelikož objekty jsou instance jednotlivých tříd, nemá smysl definovat jednotlivé objekty, ale definujeme třídy.
- Řekneme, že prvek má být typu `object`.
- Následně postupujeme jako při definici struktury (record).

Definice tříd

- Jelikož objekty jsou instance jednotlivých tříd, nemá smysl definovat jednotlivé objekty, ale definujeme třídy.
- Řekneme, že prvek má být typu `object`.
- Následně postupujeme jako při definici struktury (record).
- Jenže máme mnohem více možností.

Příklad

Popíšeme, co má v příslušné třídě být:

```
type autom=object
            nosnost,nalozeno:integer;

            procedure naloz(kolik:integer);
            procedure vyloz(kolik:integer);
            procedure dojed_k_hromade;
            procedure dojed_ke_KS;
            procedure stav;
      end;
```

Implementace metod

Atributy jsou v pořádku, ale co metody? Ty musíme definovat:

```
procedure autom.naloz(kolik:integer);
begin if(kolik+nalozeno>nosnost) then
            nalozeno:=nosnost
      else  nalozeno:=nalozeno+kolik;
      writeln('Nakladam, mam nalozeno ',nalozeno);
end;

procedure autom.vyloz(kolik:integer);
begin if(kolik>nalozeno) then
            nalozeno:=0
      else  nalozeno:=nalozeno-kolik;
      writeln('Vykladam, mam nalozeno ',nalozeno);
end;
```

Pokračování

Další metody

```
procedure autom.dojed_k_hromade;
begin
end;
procedure autom.dojed_ke_KS;
begin
end;
procedure autom.stav;
begin
writeln('Mam nosnost ',nosnost,' a nalozeno
',nalozeno,' tun...');
end;
```

Tvorba instance (tedy objektu)

Konec příkladu

```
var liaz:autom;
begin
      liaz.nosnost:=10;
      liaz.nalozeno:=0;
      liaz.naloz(5);
      liaz.stav;
end.
```

Rané poznámky k objektům

- Vypadají na pohled podobně jako struktury (recordy).

Rané poznámky k objektům

- Vypadají na pohled podobně jako struktury (recordy).
- Umí ovšem přiřadit strukturám "funkce", což může přispět k zamezení nekompetentní práce se "strukturou".

Rané poznámky k objektům

- Vypadají na pohled podobně jako struktury (recordy).
- Umí ovšem přiřadit strukturám "funkce", což může přispět k zamezení nekompetentní práce se "strukturou".
- Zatím to ovšem byla jen nevinná dětská hra, objekty toho umějí mnohem víc...

Privátní a veřejné prvky

- Chceme-li řídit přístup k vybraným prvkům, použijeme klíčová slova `public`, `private` a `protected`.

Privátní a veřejné prvky

- Chceme-li řídit přístup k vybraným prvkům, použijeme klíčová slova `public`, `private` a `protected`.
- `public` – modifikátor říká, že prvek je veřejný, smí k němu přistupovat každý (jako v příkladu).

Privátní a veřejné prvky

- Chceme-li řídit přístup k vybraným prvkům, použijeme klíčová slova `public`, `private` a `protected`.
- `public` – modifikátor říká, že prvek je veřejný, smí k němu přistupovat každý (jako v příkladu).
- `private` – označuje sekci neveřejných prvků. K těm se smí jen zevnitř třídy.

Privátní a veřejné prvky

- Chceme-li řídit přístup k vybraným prvkům, použijeme klíčová slova `public`, `private` a `protected`.
- `public` – modifikátor říká, že prvek je veřejný, smí k němu přistupovat každý (jako v příkladu).
- `private` – označuje sekci neveřejných prvků. K těm se smí jen zevnitř třídy.
- `protected` – pochopíme později, nyní stručně: Z třídy lze odvodit "potomka", tedy třídu specifičtější. Do položek `protected` se smí jen ze současné třídy, nebo z potomka.

Privátní a veřejné prvky

- Chceme-li řídit přístup k vybraným prvkům, použijeme klíčová slova `public`, `private` a `protected`.
- `public` – modifikátor říká, že prvek je veřejný, smí k němu přistupovat každý (jako v příkladu).
- `private` – označuje sekci neveřejných prvků. K těm se smí jen zevnitř třídy.
- `protected` – pochopíme později, nyní stručně: Z třídy lze odvodit "potomka", tedy třídu specifičtější. Do položek `protected` se smí jen ze současné třídy, nebo z potomka.
- Toto je způsob, jak zabránit nekompetentnímu přístupu do objektů. Nepovolujte zasahovat tam, kam to není nutné.

Privátní a veřejné prvky

- Chceme-li řídit přístup k vybraným prvkům, použijeme klíčová slova `public`, `private` a `protected`.
- `public` – modifikátor říká, že prvek je veřejný, smí k němu přistupovat každý (jako v příkladu).
- `private` – označuje sekci neveřejných prvků. K těm se smí jen zevnitř třídy.
- `protected` – pochopíme později, nyní stručně: Z třídy lze odvodit "potomka", tedy třídu specifičtější. Do položek `protected` se smí jen ze současné třídy, nebo z potomka.
- Toto je způsob, jak zabránit nekompetentnímu přístupu do objektů. Nepovolujte zasahovat tam, kam to není nutné.
- Typicky se povoluje přístup k metodám a ne k atributům, pro atributy se v krajních případech zavádějí funkce `get` a `set`.

Příklad

Jak jsme říkali, atributy neveřejné, metody veřejné: type

```
autom=object
      private nosnost,nalozeno:integer;
      public procedure naloz(kolik:integer);
            procedure vyloz(kolik:integer);
            procedure stav;
      end;
```

... jenže ouha! Kód:

```
var liaz:autom;
begin liaz.nosnost:=10;
      liaz.nalozeno:=0;
      liaz.naloz(5);
      liaz.stav;
end.
```

... nebude fungovat!

Opravička problému

Zachráníme to funkcí `init`:

```
type autom=object
      private nosnost,nalozeno:integer;
      public procedure naloz(kolik:integer);
            procedure vyloz(kolik:integer);
            procedure stav;
            procedure init(nosn:integer);
      end;
procedure autom.init(nosn:integer);
begin nosnost:=nosn; nalozeno:=0;
end;
var   liaz:autom;
begin liaz.init(10);
      ...
end.
```

Přirozené zobecnění

Bez dynamické alokace to nebylo ono ani bez objektů...

- V programu nám typicky nepostačí předem daný (pevný) počet objektů.

Přirozené zobecnění

Bez dynamické alokace to nebylo ono ani bez objektů...

- V programu nám typicky nepostačí předem daný (pevný) počet objektů.
- Zpravidla na ně chceme dělat pointery (jako když jsme se učili o spojových seznamech).

Přirozené zobecnění

Bez dynamické alokace to nebylo ono ani bez objektů...

- V programu nám typicky nepostačí předem daný (pevný) počet objektů.
- Zpravidla na ně chceme dělat pointery (jako když jsme se učili o spojových seznamech).
- Stačí, aby auto dojelo do fronty u hromady s pískem...

Přirozené zobecnění

Bez dynamické alokace to nebylo ono ani bez objektů...

- V programu nám typicky nepostačí předem daný (pevný) počet objektů.
- Zpravidla na ně chceme dělat pointery (jako když jsme se učili o spojových seznamech).
- Stačí, aby auto dojelo do fronty u hromady s pískem...
- Postupujeme úplně stejně, jako když jsme se učili o pointerech (potažmo spojových seznamech):

Přirozené zobecnění

Bez dynamické alokace to nebylo ono ani bez objektů...

- V programu nám typicky nepostačí předem daný (pevný) počet objektů.
- Zpravidla na ně chceme dělat pointery (jako když jsme se učili o spojových seznamech).
- Stačí, aby auto dojelo do fronty u hromady s pískem...
- Postupujeme úplně stejně, jako když jsme se učili o pointerech (potažmo spojových seznamech):
-
```
type automobil=^autom;
     autom=object;
     ...
     end;
var liaz:automobil;
...
```

Dynamicky alokované objekty

Pracuje se s nimi úplně přirozeně:

```
var liaz:automobil;
begin
      liaz:=new(automobil);
      liaz^.init;
      liaz^.naloz(5);
      ...
      dispose(liaz);
end.
```

Jenže typicky při naalokování (nebo odalokování) chceme udělat plno věcí (zinicializovat nebo uklidit). K tomu byl navržen tzv. *konstruktor*, resp. *destruktor.*

Konstruktory a destruktory

- Definujeme je podobně jako metody, ale uvedeme je klíčovým slovem `constructor` resp. `destructor`.

Konstruktory a destruktory

- Definujeme je podobně jako metody, ale uvedeme je klíčovým slovem `constructor` resp. `destructor`.
- K jejich volání pak dochází při volání `new` resp. `dispose`.

Konstruktory a destruktory

- Definujeme je podobně jako metody, ale uvedeme je klíčovým slovem `constructor` resp. `destructor`.
- K jejich volání pak dochází při volání `new` resp. `dispose`.
- Těmto funkcím jako druhý parametr předáme explicitní volání konstruktoru (resp. destruktoru).

Konstruktory a destruktory

- Definujeme je podobně jako metody, ale uvedeme je klíčovým slovem `constructor` resp. `destructor`.
- K jejich volání pak dochází při volání `new` resp. `dispose`.
- Těmto funkcím jako druhý parametr předáme explicitní volání konstruktoru (resp. destruktoru).
- Konstruktorů i destruktorů může být kolik chce a mohou se jmenovat vpodstatě jak chtějí.

Příklad konstruktory a destruktory

Stále auta

```
type automobil=^autom;
      autom=object;
      ...
      constructor init;
      constructor init(nosn:integer);
      destructor done;
      end;
constructor init;
begin nosnost:=10;
      nalozeno:=0;
      writeln('Zavolan konstruktor bez parametru!');
end;
```

Příklad – pokračování

Konstruktory a destruktory

```
constructor autom.init(nosn:integer);
begin
      nosnost:=nosn;
      nalozeno:=0;
      writeln('Vytvoreno auto s nosnosti ',nosnost);
end;
destructor autom.done;
begin
      writeln('Auto jede do srotu!');
end;
```

Konstruktory a destruktory

Použití – tedy volání

```
var liaz,tatra:automobil;
begin
      liaz:=new(automobil,init);
      liaz^.naloz(5);
      liaz^.stav;
      dispose(liaz,done);
      tatra:=new(automobil,init(15));
      dispose(tatra,done);
      {Tatra nezná bratra!}
end.
```

- Objektů by zjevně bylo možno použít k implementaci spojového seznamu.
- Funkce tento seznam obhospodařující by bylo možno udržovat jako metody.
- S konstruktory odpadá nutnost vyplňovat údaje ve struktuře vždycky jeden po druhé.
- Abychom mohli udělat spojový seznam samostatně implementovaný (bez globálních funkcí), můžeme ho udělat s hlavou (a volat metody nějakého reprezentanta tohoto spojového seznamu).

Dědičnost

Změna příkladu!

- Vraťme se k příkladu s knihovnou. Máme tiskoviny různých typů.

Dědičnost

Změna příkladu!

- Vraťme se k příkladu s knihovnou. Máme tiskoviny různých typů.
- Společné mají jen to, že se dávají do knihovny.

Dědičnost

Změna příkladu!

- Vraťme se k příkladu s knihovnou. Máme tiskoviny různých typů.
- Společné mají jen to, že se dávají do knihovny.
- Může se jednat o knihu, časopis nebo noviny.

Dědičnost

Změna příkladu!

- Vraťme se k příkladu s knihovnou. Máme tiskoviny různých typů.
- Společné mají jen to, že se dávají do knihovny.
- Může se jednat o knihu, časopis nebo noviny.
- Jednotlivé typy definujeme jako potomka typu `tiskovina`.

Dědičnost

Změna příkladu!

- Vraťme se k příkladu s knihovnou. Máme tiskoviny různých typů.
- Společné mají jen to, že se dávají do knihovny.
- Může se jednat o knihu, časopis nebo noviny.
- Jednotlivé typy definujeme jako potomka typu `tiskovina`.
- Syntakticky: Za klíčové slovo `object` do závorky uvedeme rodiče.

Dědičnost

Změna příkladu!

- Vraťme se k příkladu s knihovnou. Máme tiskoviny různých typů.
- Společné mají jen to, že se dávají do knihovny.
- Může se jednat o knihu, časopis nebo noviny.
- Jednotlivé typy definujeme jako potomka typu `tiskovina`.
- Syntakticky: Za klíčové slovo `object` do závorky uvedeme rodiče.
- Semanticky: Dojde ke zdědění všeho, čím rodič disponoval.

Příklad

```
type ptisk=^tiskovina;
tiskovina=object
      nazev:string;
      pocet_stran:integer;
      procedure zasun_do_knihovny;
      procedure vytahni_z_knihovny;
      ptisk next;
end;
kniha=object(tiskovina)
      autor:string;
end;
casopis=object(tiskovina)
      sefredaktor:string;
      barevnost:boolean;
end;
```

Příklad – pokračování

...

```
noviny=object(tiskovina)
      sefredaktor:string;
      objem_reklamy:real;
end;
```

- Třídy `kniha`, `casopis` i noviny zdědí společné údaje,

Příklad – pokračování

...

```
noviny=object(tiskovina)
        sefredaktor:string;
        objem_reklamy:real;
end;
```

- Třídy `kniha`, `casopis` i noviny zdědí společné údaje,
- stejně jako metody `pridej_do_knihovny` a `vytahni_z_knihovny`

Příklad – pokračování

...

```
noviny=object(tiskovina)
        sefredaktor:string;
        objem_reklamy:real;
end;
```

- Třídy `kniha`, `casopis` i noviny zdědí společné údaje,
- stejně jako metody `pridej_do_knihovny` a `vytahni_z_knihovny`
- a dokonce i ukazatel na `next`.

Příklad – pokračování

...

```
noviny=object(tiskovina)
        sefredaktor:string;
        objem_reklamy:real;
end;
```

- Třídy `kniha`, `casopis` i noviny zdědí společné údaje,
- stejně jako metody `pridej_do_knihovny` a `vytahni_z_knihovny`
- a dokonce i ukazatel na `next`.
- Příklad nasvědčuje, že pro objekty neplatí až tak striktní typová kontrola, jak známe a tedy že je možné na synovský objekt si ukázat jako na rodiče.

Příklad – pokračování

...

```
noviny=object(tiskovina)
        sefredaktor:string;
        objem_reklamy:real;
end;
```

- Třídy `kniha`, `casopis` i noviny zdědí společné údaje,
- stejně jako metody `pridej_do_knihovny` a `vytahni_z_knihovny`
- a dokonce i ukazatel na `next`.
- V konjunkci s tím, že je možné při dědění metody předefinovávat začíná pravá objektová legrace. Začne být zajímavé zjišťovat, které metody se kdy zavolají a jak se dobýt k těm správným.

Příklad – pokračování

...

```
noviny=object(tiskovina)
       sefredaktor:string;
       objem_reklamy:real;
end;
```

- Třídy `kniha`, `casopis` i noviny zdědí společné údaje,
- stejně jako metody `pridej_do_knihovny` a `vytahni_z_knihovny`
- a dokonce i ukazatel na `next`.
- K tomu použijeme tzv. *virtuální metody.*

Zapouzdření, polymorfismus, dědičnost

- Zapouzdření (encapsulation) odkazuje k tomu, že atributy "žijí" v dotyčném objektu a metody pracují nad daty příslušného objektu. Někdy se tak mluví i o tom, že objekt je zvenku určitým způsobem chráněný.

Zapouzdření, polymorfismus, dědičnost

- Zapouzdření (encapsulation) odkazuje k tomu, že atributy "žijí" v dotyčném objektu a metody pracují nad daty příslušného objektu. Někdy se tak mluví i o tom, že objekt je zvenku určitým způsobem chráněný.
- Polymorfismus – synovské třídy se mohou chovat určitým způsobem odlišně, než rodičovské třídy. Můžeme metody předefinovávat a překrývat.

Zapouzdření, polymorfismus, dědičnost

- Zapouzdření (encapsulation) odkazuje k tomu, že atributy "žijí" v dotyčném objektu a metody pracují nad daty příslušného objektu. Někdy se tak mluví i o tom, že objekt je zvenku určitým způsobem chráněný.
- Polymorfismus – synovské třídy se mohou chovat určitým způsobem odlišně, než rodičovské třídy. Můžeme metody předefinovávat a překrývat.
- Dědičnost nám umožňuje definovat třídu jako šablonu určující, jak mají být definovány společné rysy synovských tříd. To ovšem není všechno.

Zapouzdření, polymorfismus, dědičnost

- Zapouzdření (encapsulation) odkazuje k tomu, že atributy "žijí" v dotyčném objektu a metody pracují nad daty příslušného objektu. Někdy se tak mluví i o tom, že objekt je zvenku určitým způsobem chráněný.
- Polymorfismus – synovské třídy se mohou chovat určitým způsobem odlišně, než rodičovské třídy. Můžeme metody předefinovávat a překrývat.
- Dědičnost nám umožňuje definovat třídu jako šablonu určující, jak mají být definovány společné rysy synovských tříd. To ovšem není všechno.
- Pointerem na rodičovský typ je možno ukázat na syna. A nyní, co z toho všechno plyne?

Rodič ukazuje na syna

```
type  zvire=^zv;
      zv=object
            vek:integer;
            procedure kdotam;
            next:zvire;
      end;
      pes=^p;
      p=object(zv)
            procedure hlidej;
      end;
      kocka=^k;
      k=object(zv)
            procedure chyt_drzou_mys;
      end;
```

```
procedure zv.kdotam;
begin
      writeln('Ja ''sem ale zvire!');
end;

procedure p.hlidej;
begin
      writeln('Haf, baf!');
end;

procedure k.sezer_drzou_mys;
begin
      writeln('Kocka kolotocka dela ham!');
end;
```

Hlavní program

```
var zverinec:zvire;
begin
      zverinec:=new(zvire);
      zverinec^.next:=new(kocka);
      zverinec^.kdotam;
      zverinec^.next^.kdotam;
end.
```

Využití dědičnosti

- Dědičnost můžeme použít jako šablonu na jednotlivé (synovské) třídy.
- Jde tedy o něco jako variantní record z minuta!
- Jenže k tomu místo definování nových tříd budeme chtít předefinovávat staré metody,...
- ... což kupodivu lze.

Předefinovávání metod

Minulý příklad

```
type  zvire=^zv;
      zv=object
            vek:integer;
            procedure kdotam;
            next:zvire;
      end;
      pes=^p;
      p=object(zv)
            procedure hlidej;
      end;
      kocka=^k;
      k=object(zv)
            procedure chyt_drzou_mys;
      end;
```

Předefinovávání metod

A hle, metody se jmenují stejně!

```
type  zvire=^zv;
      zv=object
            vek:integer;
            procedure kdotam;
            next:zvire;
      end;
      pes=^p;
      p=object(zv)
            procedure kdotam;
      end;
      kocka=^k;
      k=object(zv)
            procedure kdotam;
      end;
```

Ale co udělá program?

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.

Ale co udělá program?

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!

Ale co udělá program?

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!
- `zv:=new(kocka);`
  v pořádku, přiřazujeme syna do rodiče.

Ale co udělá program?

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!
- `zv:=new(kocka);`
  v pořádku, přiřazujeme syna do rodiče.
- `zv^.kdotam;`
  Ouha:
  Já 'sem ale zvíře!
  Proč?

Ale co udělá program?

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!
- `zv:=new(kocka);`
  v pořádku, přiřazujeme syna do rodiče.
- `zv^.kdotam;`
  Ouha:
  Já 'sem ale zvíře!
  Proč?
- Protože jednotlivé metody se hledají v prototypu (tedy v popisu příslušné třídy, v tomto případě `zvire`).
  Jak z toho?

Virtuální funkce

- Virtuální metody se nehledají v prototypu, nýbrž každý objekt má **tabulku virtuálních metod** (alias VMT), ve které jsou pointery na jednotlivé virtuální funkce.

Virtuální funkce

- Virtuální metody se nehledají v prototypu, nýbrž každý objekt má **tabulku virtuálních metod** (alias VMT), ve které jsou pointery na jednotlivé virtuální funkce.
- Funkci `kdotam` tedy uděláme virtuální (zejména ve třídě `zvire`, pak už ale bude virtuální i v synovských třídách).

Předefinovávání metod

A hle, metody jsou virtuální!

```
type  zvire=^zv;
      zv=object
            vek:integer;
            procedure kdotam; virtual;
            next:zvire;
      end;
      pes=^p;
      p=object(zv)
            procedure kdotam; virtual;
      end;
      kocka=^k;
      k=object(zv)
            procedure kdotam; virtual;
      end;
```

Volání virtuálních metod

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.

Volání virtuálních metod

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!

Volání virtuálních metod

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!
- `zv:=new(kocka);`
  v pořádku, přiřazujeme syna do rodiče.

Volání virtuálních metod

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!
- `zv:=new(kocka);`
  v pořádku, přiřazujeme syna do rodiče.
- `zv^.kdotam;`
  'Kocka kolotocka'

Volání virtuálních metod

- `var zv:zvire; zv:=new(zvire);`
  Vytvoříme nové zvíře.
- `zv^.kdotam;`
  Já 'sem ale zvíře!
- `zv:=new(kocka);`
  v pořádku, přiřazujeme syna do rodiče.
- `zv^.kdotam;`
  'Kocka kolotocka'
- Virtuální metoda se hledá ve VMT, kde je adresa `kdotam` od typu `kocka`.

Poznámky

- Pokud neřekneme, že metoda má být v synovských metodách virtuální, překladač to buďto sám pochopí, nebo nepřeloží (GPC vydedukuje). Při divočejším předefinovávání virtuálních metod nevirtuálními můžeme mít problémy.

Poznámky

- Pokud neřekneme, že metoda má být v synovských metodách virtuální, překladač to buďto sám pochopí, nebo nepřeloží (GPC vydedukuje). Při divočejším předefinovávání virtuálních metod nevirtuálními můžeme mít problémy.
- Pokud má synovská třída (na kterou si ukazujeme pomocí rodičovského typu) destruktor, je potřeba, aby tento byl virtuální!

Funkce `typeof`

- Tím, že můžeme přiřadit syna do rodiče nám vzniká nepořádek. Občas chceme vědět, na jaký typ si ukazujeme.

Funkce `typeof`

- Tím, že můžeme přiřadit syna do rodiče nám vzniká nepořádek. Občas chceme vědět, na jaký typ si ukazujeme.
- K tomu je funkce `typeof`, které předáme strukturu nebo jméno třídy:

Funkce `typeof`

- Tím, že můžeme přiřadit syna do rodiče nám vzniká nepořádek. Občas chceme vědět, na jaký typ si ukazujeme.
- K tomu je funkce `typeof`, které předáme strukturu nebo jméno třídy:
- `if typeof(zv^)=typeof(k) then writeln('Je to kocka!');`

Funkce `typeof`

- Tím, že můžeme přiřadit syna do rodiče nám vzniká nepořádek. Občas chceme vědět, na jaký typ si ukazujeme.
- K tomu je funkce `typeof`, které předáme strukturu nebo jméno třídy:
- `if typeof(zv^)=typeof(k) then writeln('Je to kocka!');`
- Takto můžeme například implementovat spojový seznam s hlavou pomocí objektů, kde hlavu si označíme speciálním typem.

Funkce `typeof`

- Tím, že můžeme přiřadit syna do rodiče nám vzniká nepořádek. Občas chceme vědět, na jaký typ si ukazujeme.
- K tomu je funkce `typeof`, které předáme strukturu nebo jméno třídy:
- `if typeof(zv^)=typeof(k) then writeln('Je to kocka!');`
- Takto můžeme například implementovat spojový seznam s hlavou pomocí objektů, kde hlavu si označíme speciálním typem.
- Příklad lze najít na stránkách kolegy Kryla.

Čistě virtuální funkce, abstraktní třídy

- Jde o metody, které nechceme definovat (společný předek slouží jen jako šablona).

Čistě virtuální funkce, abstraktní třídy

- Jde o metody, které nechceme definovat (společný předek slouží jen jako šablona).
- Čistě virtuální funkce je virtuální funkce, kterou nechceme definovat v rodičovském typu, ale chceme, aby byla ve všech synovských typech překryta místní virtuální funkcí.

Čistě virtuální funkce, abstraktní třídy

- Jde o metody, které nechceme definovat (společný předek slouží jen jako šablona).
- Čistě virtuální funkce je virtuální funkce, kterou nechceme definovat v rodičovském typu, ale chceme, aby byla ve všech synovských typech překryta místní virtuální funkcí.
- V Pascalu stojí na dobrovolnosti (funkce definujeme jako RunError).

Čistě virtuální funkce, abstraktní třídy

- Jde o metody, které nechceme definovat (společný předek slouží jen jako šablona).
- Čistě virtuální funkce je virtuální funkce, kterou nechceme definovat v rodičovském typu, ale chceme, aby byla ve všech synovských typech překryta místní virtuální funkcí.
- V Pascalu stojí na dobrovolnosti (funkce definujeme jako RunError).
- Zpravidla (v moderních objektových jazycích) je syntaktická podpora a například v C++ nelze vůbec definovat proměnnou takového typu.

Čistě virtuální funkce, abstraktní třídy

- Jde o metody, které nechceme definovat (společný předek slouží jen jako šablona).
- Čistě virtuální funkce je virtuální funkce, kterou nechceme definovat v rodičovském typu, ale chceme, aby byla ve všech synovských typech překryta místní virtuální funkcí.
- V Pascalu stojí na dobrovolnosti (funkce definujeme jako RunError).
- Zpravidla (v moderních objektových jazycích) je syntaktická podpora a například v C++ nelze vůbec definovat proměnnou takového typu.
- Třída obsahující aspoň jednu čistě virtuální funkci se nazývá *abstraktní.*

Příklad čistě virt. metody

- Například každá živá hmota dýchá. Je ale otázka, jestli dýchá kyslík nebo oxid uhličitý a jestli dýchá žábrami nebo ústy (kyslík ze vzduchu nebo ten rozpuštěný ve vodě).

Příklad čistě virt. metody

- Například každá živá hmota dýchá. Je ale otázka, jestli dýchá kyslík nebo oxid uhličitý a jestli dýchá žábrami nebo ústy (kyslík ze vzduchu nebo ten rozpuštěný ve vodě).
- Typu `ziva_hmota` tak definujeme metodu `dychej`, která bude čistě virtuální a bude muset být ve všech synovských třídách překryta.

Objekt `self`, operátor vzetí pointeru

- Říkali jsme si o konstruktorech a destruktorech. Konstruktor často udělal plno užitečné práce, například...

Objekt `self`, operátor vzetí pointeru

- Říkali jsme si o konstruktorech a destruktorech. Konstruktor často udělal plno užitečné práce, například...
- přidal prvek do spojového seznamu (obousměrného). Jenže jak to udělal?

Objekt `self`, operátor vzetí pointeru

- Říkali jsme si o konstruktorech a destruktorech. Konstruktor často udělal plno užitečné práce, například...
- přidal prvek do spojového seznamu (obousměrného). Jenže jak to udělal?
- ```
  type spojak=^sp; sp=object
       hod:longint; prev,next:spojak;
  ...
       constructor init(h:longint;p,n:spojak);
       begin  hod:=h; next:=n; prev:=p;
       end;
  ```
  Stačí to takhle?
  ```

Objekt `self`, operátor vzetí pointeru

- Říkali jsme si o konstruktorech a destruktorech. Konstruktor často udělal plno užitečné práce, například...
- přidal prvek do spojového seznamu (obousměrného). Jenže jak to udělal?
- ```
  type spojak=^sp; sp=object
       hod:longint; prev,next:spojak;
  ...
       constructor init(h:longint;p,n:spojak);
       begin  hod:=h; next:=n; prev:=p;
       end;
  ```
  Stačí to takhle?
- Správně, následník a předchůdce neukazuje na nás. Jak to zalepit?
  ```

- Pomocí objektu `self`, který odkazuje k současnému objektu.

- Pomocí objektu `self`, který odkazuje k současnému objektu.
- A pomocí operátoru vzetí pointeru `@`.

- Pomocí objektu `self`, který odkazuje k současnému objektu.
- A pomocí operátoru vzetí pointeru `@`.
- Takhle: `next^.prev:=@self;`
  `prev^.next:=@self;`

- Pomocí objektu `self`, který odkazuje k současnému objektu.
- A pomocí operátoru vzetí pointeru `@`.
- Takhle: `next^.prev:=@self;`
  `prev^.next:=@self;`
- Takto tedy můžeme v případě potřeby odkázat k současnému objektu.

- Pomocí objektu `self`, který odkazuje k současnému objektu.
- A pomocí operátoru vzetí pointeru `@`.
- Takhle: `next^.prev:=@self;`
  `prev^.next:=@self;`
- Takto tedy můžeme v případě potřeby odkázat k současnému objektu.
- Samozřejmě operátorem vzetí pointeru můžeme vytvořit pointer na cokoliv (podobně jako operátorem stříšky můžeme jakýkoliv (negenerický) pointer dereferencovat).

Konec

...děkuji za pozornost...

Otázky?