Dědičnost

- Nezřídka máme speciální případy obecné třídy a chceme, aby se chovaly uniformně.
- Například typ živá hmota má 7 metod (projevů života), každá živá hmota se ale chová úplně jinak.
- Chceme-li implementovat zvěřinec, hodilo by se implementovat typ zvíře a od něj odvodit typy jednotlivych živočišných druhů.
- Definovat "rodiče" je snadné, syna definujeme opět pomocí operátoru dvojtečky.

Příklad

rodič

```
class zvire {
      string jmeno;
      public void VydejZvuk()
      { Console.WriteLine("Nevydam, jsem zvire!");}
      public void NastavJmeno(string jmeno)
      { this.jmeno=jmeno;}
      public void KdoTam()
      { Console.WriteLine(jmeno);}
}
```

Příklad

syny

```
class tygr:zvire
{    public tygr(string jmeno)
     {    NastavJmeno(jmeno);}
     public void VydejZvuk()
     {    Console.Writeline("Vrrrrrr-rrum!");}
}
class slepice:zvire
{   public slepice() //slepice jmena nemaji
    { NastavJmeno("zadne"); }
    public void VydejZvuk()
    {    Console.WriteLine("Ko - ko - ko!");}
}
```

Použití dědičnosti

- Synovská třída zdědí všechno z rodičovské,
- pokud nějakou metodu předefinujeme, je předefinována,
- potřebujeme ale uniformní přístup ke všem synům (jinak by dědičnost byla k ničemu), proto
- do rodiče lze přiřadit syna:
- `zvire matysek=new tygr("Matysek");`
- Můžeme zavolat zděděné metody:
  `matysek.KdoTam();`
- Můžeme zkusit zavolat synovské metody (definované i v rodiči),
- ale spláčeme nad vejdělkem: `matysek.VydejZvuk(); =>`
  `"Nevydam, jsem zvire!"`
- Abychom mohli volat synovské metody, musí být metoda virtuální!

Virtuální metody

- Metody jednotlivých objektů se hledají v prototypu dané třídy,
- proto příklad z minulého slidu nefungoval (jak by měl).
- Virtuální metody jsou reprezentovány odkazy z VMT.
- VMT přísluší taktéž celé třídě, ale...
- každý objekt má ukazatel na tu svou (tedy syn přetypovaný na otce může poznat podle VMT, že je syn).
- Syntax: Modifikátory `virtual` a `override`.
- V rodiči použijeme `virtual`, v potomcích `override`.

Příklad

rodič

```
class zvire {
      string jmeno;
      public virtual void VydejZvuk()
      { Console.WriteLine("Nevydam, jsem zvire!");}
      public void NastavJmeno(string jmeno)
      { this.jmeno=jmeno;}
      public void KdoTam()
      { Console.WriteLine(jmeno);}
}
```

Příklad

syny

```
class tygr:zvire
{    public tygr(string jmeno)
     {    NastavJmeno(jmeno);}
     public override void VydejZvuk()
     {    Console.Writeline("Vrrrrrr-rrum!");}
}
class slepice:zvire
{   public slepice() //slepice jmena nemaji
    { NastavJmeno("zadne"); }
    public override void VydejZvuk()
    {    Console.WriteLine("Ko - ko - ko!");}
}
```

Poznámky

k virtuálním metodám – aneb co když zkoušíme překladač zlobit...

- Pokud neuvedeme `override`, vznikne v synu nová (nevirtuální) metoda (a pochopitelně se nezavolá, jak bychom chtěli).
- Pokud neuvedeme `virtual` (a `override ano`), je z toho error (nelze overridovat nevirtuální metodu).
- Zapomeneme-li oboje, je z toho warning (zakrýváme metodu a není jasné, jestli je to to, co chceme).
- V předchozím bodě se warningu zbavíme modifikátorem `new`.

Čistě virtuální funkce a abstraktní třídy

- Co když má rodičovská třída sloužit jen jako vzor (a nechceme dělat instance – a některé metody nechceme ani definovat)?
- Uděláme abstraktního rodiče (modifikátor `abstract`).
- Příklad: `abstract class zvire...`
  a `public abstract void VydejZvuk();`
- Abstraktní (čistě virtuální) metoda je automaticky virtuální.
- Odlišnost od C++: C++ nemá modifikátor `abstract` a do čistě virtuální funkce se přiřadí `NULL`.

Příklad

rodič

```
abstract class zvire {
      string jmeno;
      public abstract void VydejZvuk();
      public zvire():this("Nemam!"){}
      public void zvire(string jmeno)
      { this.jmeno=jmeno;}
      public void KdoTam()
      { Console.WriteLine(jmeno);}
}
```

Příklad

syny

```
class tygr:zvire
{    public tygr(string jmeno):base(jmeno){}
     public override void VydejZvuk()
     {    Console.Writeline("Vrrrrrr-rrum!");}
}
class slepice:zvire
{   public slepice():base(){} //slepice jmena nemaji
    public override void VydejZvuk()
    {    Console.WriteLine("Ko - ko - ko!");}
}
```

Modifikátory přístupu

- public – veřejné prvky, přistoupit smí každý
- protected – přistoupit smí sama třída a potomci
- private (implicitní) – jen já (současná třída)
- internal – jen současné assembly (česky sestavení – zatím ignorujme)
- protected internal – současné assembly a potomci.
- Je zvykem nepouštět si nikoho k datům (zapouzdřenost),
- k nastavování a čtení se tvoří veřejné metody.
- Ač někdy hraničí s objektovým fanatismem, ve větších projektech je to užitečné!
- Metody get a set si lze nechat téměř vygenerovat:

Příklad

metod get a set

```
class zvire
{    private int pocetNohou;
     public int PocetNohou //pozor, velikost!
     {    get
          {    return pocetNohou;}
          set
          { if((value % 2)==0) pocetNohou=value;}
     }
}
```

Použití

```
matysek=new Tygr("Matysek");
matysek.PocetNohou=4;
matysek.PocetNohou=3;
System.Console.WriteLine(matysek.PocetNohou);
```

Zapečetěné třídy a metody

- Modifikátor `sealed`,
- znamená, že z třídy nelze dědit,
- metodu nelze overrideovat (lze jen udělat new).
- Příklad: `sealed class bezdeti{...}`
- Příklad: `public sealed override void cilova_rovinka(){...}`

Destruktor

- Analogie konstruktoru, volá ho memory-management při odallokování,
- to nastane při garbage-collection (což je těžké odhadnout).
- Destruktor má oproti konstruktoru na začátku vlnku (`~zvire`).
- Nebere argumenty a nic nevrací.
- Tudíž se v C# příliš nepoužívá (v C++ má naopak dobrý smysl).

Struktury

- V jazyce C nebyly objekty, ale byly struktury,
- ovládaly se podobně jako recordy v Pascalu,
- definovaly se pomocí klíčového slova `struct` (syntakticky podobně jako třídy),
- V C# jsou také (ale místo nich se spíše používají objekty).
- Příklad: `struct kompl{public int re,im;};`

Dynamicky alokované proměnné

- V Pascalu jsme vytvořili datový typ,
- volali jsme `new` a `dispose`,
- používali jsme operátor stříšky (dereference) a zavináče (vzetí ukazatele),
- používali jsme operátor tečky.
- V C# ukazatele nejsou, tudíž se nepoužívají operátory vzetí reference a dereference,
- je tu garbage collector, tedy neříkáme ani `dispose`.
- Cokoliv pohodíme, garbage collector dle vlastního uvážení uklidí.

allokace a "konec paměti"

- `null` je ekvivalent pascalského NIL,
- používá se úplně stejně.
- Místo pointerů jsou typy rozdělené na hodnotové a referenční.
- Operátor dereference je implicitní a udělá se vlastně vždy při použití operátoru tečky.
- Alokace: `new` + volání konstruktoru `;` (už jsme si ukazovali minule).

Spojové seznamy

- V Pascalu jsme potřebovali strukturu (record) s ukazatelem (na next).
- V C# použijeme objekty podobným způsobem:
- ```
  class prvek{
      public int hodnota;
      public prvek next;
  }
  ```
- Inicializace: `prvek a=null;`
- Smazání seznamu: `prvek a=null;`
  ```

Univerzální spojový seznam

```
abstract class seznobecny
{    seznobecny next;
     public seznobecny Next
     {    set { next=value;}
          get { return next;}
     }
     public seznobecny()
     {    next=null;}
}
```

Příklad – syn

```
class seznintu:seznobecny
{    public int hodnota;
     public seznintu(int hodnota)
     {    this.hodnota=hodnota; }
}
seznobecny x=new seznintu(10);
while(x.Next!=null)
    x=x.Next;
```

Pole

jsou úplně snadná, ale chovají se úplně jinak než v Pascalu

- Typ pole definujeme operátorem hranatých závorek za jménem typu:
- `int [] poleintu;`
- Pole se definují předem neznámé délky a dynamicky se alokují:
- `int[] pole=new int[10];`
- Pole se indexují od nuly (do délka - 1 (!))
- Pole některých typů lze rovnou inicializovat:
- `int[] pole=new int[3]{1,2,3};` anebo
- `int[] pole=new int{1,2,3};`

Pole II

- Pole námi definovaného typu:
- `seznobecny[] polesez=new seznintu [10];`
- Toto pole se neinicializuje (je plné nullů)!
- `polesez[0]=new seznintu(10);`
- Délka pole – atribut Length:
  `delka=polesez.Length;`
- Při sáhnutí mimo rozsah pole vypadne IndexOutOfRangeException,
- Při použití neinicializovaného odkazu NullReferenceException.

Pole III

- Vícerozměrná pole: `int [,] pole=new int[2,3];`
- Takové pole je obdélníkové a:
  `pole.Rank==2, pole.Length==6`
- Nepravidelné pole (pole polí):
  `int [][] pole=new int [3][];`
- Následně: `pole[0]=new int[3];`
  `pole[1]=new int[2];...`
- Konstrukce `foreach`:
- ```
  int[] pole={1,2,3,4,5};
  foreach (int i in pole)
      Console.WriteLine(i);
  ```
  ```

Řetězce

Mají některé ďábelské vlastnosti

- Proměnné typu `string`,
- jsou to objekty, ale lze je inicializovat implicitně:
- `string retez="kus textu!";`
- Porovnání na rovnost (překvapivě) porovnává obsahy
- (mezi jazyky rodiny C je to ale spíše výjimka).
- Délka (atribut Length): `retez.Length`,
- Lze k němu přistupovat jako k poli,
- ale je read-only! Pro zápis použijte StringBuilder.
- Pole lze rozsekat podle oddělovačů:
  `string[] retez.Split(char[])`
- Příklad: `string retez="1 22 3 14";`
  `string[]cisla=retez.Split({' '});`

Allokace poznámky

- Množství dostupné paměti:
  `System.GC.GetTotalMemory(bool)`
- Explicitní volání Garbage-collectoru: `System.GC.Collect()`

Konec...

... děkuji za pozornost.