Simulace

Pokračování

Seznamy

- Seznamy (spojové) a základní datové struktury byly probrány v Pascalu,
- protože v C# jsou už předimplementované:
- `System.Collections.ArrayList` je univerzální seznam.
- Instance této třídy jsou osazeny například metodami:
  - `Add` – přidání prvku.
  - `Remove` – odebrání prvku (jednoho výskytu),
  - `Sort` – setřídění (by default integerů, se stringy jsem měl potíže),
  - `IndexOf` – vyhledání prvku, při nálezu vrací nezáporné číslo, není-li prvek nalezen, vrátí se -1.

Příklad

```
using System.Collections;
ArrayList AL = new ArrayList();
AL.Add("První");
AL.Add(222);
AL.Add(100);
AL.Add(1);
AL.Add(null);
AL.Remove("První");
```

Příklad pokračování

Všechny prvky lze vypsat pomocí `foreach`:

```
Console.WriteLine("Počet: 0", AL.Count );
Console.WriteLine("Prvni prvek: 0", AL[0]);

AL.Sort();

System.Console.Write("Všechny prvky: ");
foreach (object obj in AL)
System.Console.Write("0", obj);
System.Console.WriteLine();
```

Typový seznam a generika

- Často potřebujeme udělat to samé pro různé typy (například spojový seznam).
- Víme ale předem, že prvky budou homogenní (tedy jednoho typu).
- K tomu slouží generika (v C++ šablony). Poznají se podle parametru ve špičatých závorkách:
- `List<int> cisla=new List<int>();`
- Po definici s generikem pracujeme jako s obyčejnou proměnnou.
- `cisla.Add(10);`
- Dnes si ukážeme jen jak generika použít.

Generikum List

- Je nástupcem ArrayListu od C# 2.0, proto se ovládá podobně.
- ```
  List<int> cela_cisla=new List<int>();
  cela_cisla.Add(5);
  cela_cisla.Add(3);
  cela_cisla.Add(1);
  foreach(int i in cela_cisla)
       Console.WriteLine(i);
  Console.WriteLine("Celkem {0}",cela_cisla.Count);
  ```
  ```

Komplexní čísla

... jenže jenže, mně integery nestačí

```
class Komplexni
{    public double Re,Im;
     public Komplexni(double Re,double Im)
     { this.Re=Re; this.Im=Im;}
}
List<Komplexni> s=newList<Komplexni>();
s.Add(new Komplexni(1,0));
s.Add(new Komplexni(0,1));
```

Generická třída List

- je v `System.Collections.Generic`,
- obsahuje řadu metod, například:
- `Add, Contains, Sort, BinarySearch`
- Příklad:

```
List<string> s1 = new List<string>();
s1.Add("abcd");
s1.Add("efgh");
if(s1.Contains("abcd"))
      Console.WriteLine("Je tam!");
```

2. možnost

implementace diskrétní simulace

- Kalendář je seznam událostí `List<Udalost>`,
- jednotlivé procesy jsou postrkovány ovladači událostí,
- v tom případě fronty na čekající procesy nejsou potřeba,
- čekání lze realizovat tak, že událost naplánujeme na nejbližší čas, kdy může nastat,
- musíme dát pozor na race-condition.

Generika I

- v C++ fungují obecnější šablony,
- využijeme jich, pokud chceme vytvořit šablonu nějaké třídy, tedy
- chceme vytvořit více totožných tříd, které se budou lišit datovým typem.
- Příklad využití byl minule v podobě generické třídy `List`.
- Jde o jistou náhražku maker preprocesoru známých z jazyka C.

Generika II

- Při použití generika jsme postupovali stejně jako u obyčejného datového typu, jen jsme na správné místo dali parametr do špičatých závorek.
- Při definici postupujeme podobně a tímto parametrem bude datový typ se všemi důsledky, které z toho plynou,
- tedy můžeme dělat proměnné dotyčného (parametrického) typu.
- `class jmeno <parametry,oddelene,carkami>` `{ definice třídy }`
- Příklad: `public class genericka <T> {public T promenna;}`

Generika příklad

```
public class seznam <T>
{    public T data;
     public seznam<T> next;
}
...
     seznam<int> x=new seznam<int>();
     x.data=10;
     x.next=new seznam<int>();
     // Tohle by bylo spatne:
     // x.next=new seznam<double>();
```

Komplexní čísla

a operace s nimi, aneb přetížené operátory

- Čísla jsou různá: Přirozená, celá, racionální, reálná, komplexní...
- Na všech ale má smysl definovat sčítání, násobení, odčítání a dělení.
- V počítači máme jen čísla celá a necelá. Co s tím?
- Definujeme si vlastní třídu. Tu ale nepůjde sčítat a násobit. Anebo - ze by?
- Ano: Dotyčné třídě přetížíme operátory (přesněji dodefinujeme je).
- Syntakticky se tváříme, jako bychom definovali běžnou statickou metodu, tato funkce se ale bude divně jmenovat.

Příklad

Opět Gaussova celá čísla

```
class kompl
{    public int re,im;
     public kompl(int re,int im)
     {    this.re=re; this.im=im;}
     public static kompl operator + (kompl a,kompl b)
     {    return new kompl(a.re+b.re,a.im+b.im);}
     public static kompl operator * (kompl a,kompl b)
     {    return new kompl(a.re*b.im-a.im*b.im,
               a.re*b.im+a.im*b.re);
     }
}
```

Příklad pokračování

Abychom mohli třídu `kompl` demonstrovat, předefinujeme jí virtuální metodu `ToString` jako minule:

```
public override string ToString()
{    return ""+re+"+ "+im+"i";}
```

A jedeme:

```
kompl a=new kompl(1,0), b=new kompl(0,1),c;
c=a+b;
Console.WriteLine(c);
Console.WriteLine(a*b);
```

Přetížitelné operátory

Přetížit lze mnoho operátorů, konkrétně:
unární `+`, `-`, `!`, `~`, `++`, `--`
a binární `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`
NELZE například $\&\&, ||, [], (typ)x, +=, -=...$

Když je problém můžeme...

- ukončit program,
- na všech možných místech myslet na všechno možné,
- nehasit, co nás nepálí.
- V Pascalu byly k dispozici první dvě možnosti, tedy buďto pštrosí algoritmus, nebo se starat.
- C# umožňuje všechny tři možnosti, my zatím umíme tu první.

Výjimky II

- Výjimky už známe, objevovaly se, když jsme šlápli vedle a program tím skončil.
- My ovšem můžeme výjimky odchytávat,
- anebo dokonce házet a posílat tím zprávu, že se něco nepovedlo.
- Výjimka postupně propadá programem a ukončuje funkce, které ji nečekaly,
- dokud nevypadneme z programu, nebo nenarazíme na blok, který ji očekával.

Výjimky III

- Syntax a sémantika:
- `try` uvádí blok, ve kterém může nastat výjimka.
- `catch` uvádí ovladač události za `try` blokem.
- `catch` bloků může být více, protože výjimek je mnoho typů (a každou můžeme ošetřovat zvlášť, přesto jsou všechny výjimky potomkem třídy `System.Exception`).
- `finally` uvádí blok, který se má provést v každém případě (ať výjimka přijde nebo ne a ať je výjimka jakákoliv, tedy včetně nečekané).
- `throw` hodí výjimku (ovládá se podobně jako return).

Výjimky příklad

```
void bezpecnedeleni(int a, int b)
{    try{
          Console.WriteLine(a / b);
     }
     catch(System.DivideByZeroException e)
     {    Console.WriteLine("NELZE");}
}
```

Vlastní výjimka

```
class me:System.Exception{}
...
void bezpecnedeleni(int a, int b) {    try{
          if(y==0) throw new me();
          return (x/y);
     }
     catch (System.Exception e)
     {    Console.WriteLine("Prisla vyjimka!"); }
     finally
     {    Console.WriteLine("V kazdem pripade...");}
}
```

Výjimky – poznámky I

- Bloků `catch` může být více za sebou.
- Vykoná se první blok, který popisuje dotyčnou výjimku.
- V C# je třeba definovat ovladače synovské výjimky před rodičovskými:
- `catch(System.Exception e){...}`
  `catch(System.DivideByZeroException e){...}`
  ... tohle ani nezkompilujeme.

Výjimky – poznámky II

- Jak neprogramovat:

```
bool uz=false;
while(!uz)
{    try{ volani_divne_funkce();uz=true;}
     catch(System.Exception e)
     {    Console.WriteLine("Tak znova...");}
}
```

- Výjimky jsou dobrý sluha, ale špatný pán!

Konec

Děkuji za pozornost...