2 24. března – Diskrétní simulace podruhé, Collections, Generics, Výjimky

Stále jsme motivováni diskrétní simulací. Řešení s kalendářem událostí reprezentovaným obousměrným cyklickým spojovým seznamem brzy začne být neefektivní. Je proto vhodné použít jiné datové struktury, například haldy (přednesené na Algoritmizaci, které umějí přesně to, co kalendář událostí potřebuje).

Nebo můžeme použít již předpřipravená řešení (která bychom ale měli umět v případě potřeby také sami implementovat). Předpřipravená řešení reprezentují:

2.1 Collections

Archaičtější z dostupných prostředků. Implementuje datové struktury nad třídou `object`, která je společným rodičem všech tříd. Tyto struktury se nacházejí v `System.Collections`. Například `System.Collections.ArrayList` je generický (univerzální) seznam. Disponuje metodami `Add, Remove, Sort, IndexOf,...`, podrobnosti a příklady jsou na slidu.

V C# lze použít konstrukci podobající se Pythonskému `for`-cyklu. Jmenuje se `foreach:`

```
foreach(cyklici\_promenna in prohledavany\_objekt)
        telo\_dle\_obvyklych\_pravidel();
```

Příklad je ve slidech.

2.2 Generické datové typy – použití

Nástupci Collectionů jsou generické datové typy. Datový typ parametrizujeme (dalším datovým typem). Například seznam parametrizujeme typem dat, která do něj chceme ukládat. Při využití (tedy definici proměnných) generických typů postupujeme podobně jako při běžné definici proměnných, jen u typu musíme říct, čím je parametrizován. Tyto parametry dáváme do špičatých závorek. Jeden z generických datových typů se jmenuje `List` a je parametrizován typem dat, které do něj budeme ukládat:

```
List <int> cisla=new List<int>();
```

Takto vytvoříme seznam integerů.

Generický seznam je taktéž vybaven metodami a atributy `https://docs.microsoft.com/cs-cz/dotnet/api/system.collections.generic.list-1?view=netframework-4.8#properties` – pro nás mohou být užitečné například metody `Add, Clear, Contains, IndexOf, Remove, Sort, ToArray.`

Generický datový typ může být parametrizován i více typy. V takovém případě je (ve špičatých závorkách) oddělujeme čárkou.

2.3 Využití v diskrétní simulaci

Generický typ `List` lze použít k implementaci kalendáře událostí.
Výhoda: Plno věcí máme předpřipravených.
Nevýhoda: Nevíme jak fungují.

Taktéž můžeme v diskrétní simulaci využít vlastní generické třídy a metody – například na práci s různými typy událostí či procesů.

2.4 Definice generických tříd a metod

Generické datové typy jsou variantou šablon známých z C++. Šablony jsou poněkud obecnější například proto, že mohou být parametrizovány čímkoliv. Generické datové typy a metody mohou být parametrizovány jen datovým typem (nedošlo-li v C# opět ke změně).

Rozdíl mezi využitím obyčejného datového typu a generického datového typu (tedy definice proměnné příslušného typu) poměrně dobře ilustruje i rozdíl v definici běžné třídy či metody a tříd či metod generických. Opět jen dáme na správné místo špičaté závorky. Do nich tentokrát dáme jméno, které bude v dotyčném prvku zastupovat tento generický typ.

Jako příklad si implementujeme generická komplexní čísla. Ta můžeme tvořit nad různými datovými typy. Třeba nad typem int jako tzv. Gaussova celá čísla (aby senám neztrácela přesnost). Nebo nad doublem (aby čísla nemusela být jen celá). Anebo je můžeme tvořit nad třídou reprezentující racionální čísla (ve formátu čitatel, jmenovatel, aby je šlo dělit bez ztráty přesnosti):

```
class kompl<T>
{    T re,im;
      public T Re {
          get { return re; }
          set { re=value; }
     }
     public override string ToString()
     {    return Convert.ToString(re)+" + "+Convert.ToString(im)+"i ";
     ...
}

static void Main(string[]args)
{   kompl<int> a=new kompl<int>(),b=new kompl<int>();
    kompl<double> c=new kompl<double>();
    a.Re=1;
    Console.WriteLine(a.Re);
}
```

Takto definujeme generickou třídu `kompl` vybavenou reálnou a imaginární částí, wrapperem kolem atributu `re`. Samostatně dodejte wrapper kolem `im`.

Taktéž overridujeme metodu `ToString`. Tato metoda je velice důležitá, proto k ní nyní uděláme odbočku. Jistě jste si lámali hlavu, jak to, že metoda `WriteLine` dovede vypsat téměř cokoliv. Je to proto, že jedna z jejích definic operuje s parametrem typu `object`. Toto je společný rodič všech tříd. V této třídě je definována metoda `ToString`, která má zajistit konverzi do stringu. Tuto metodu metoda `WriteLine` vyvolá, aby vynutila konverzi objektového argumentu do stringu. S touto metodou tedy budeme moci v příkladu do funkce Main (až uděláme wrapper kolem proměnné `im`) nechat číslo vypsat takto:
`Console.WriteLine(a);`

K příkladu se ještě vrátíme při odbočce k `interfacům`. Budeme totiž chtít umět tato čísla porovnávat. Teď byly ale slíbené generické třídy (které právě byly) a generické metody (o kterých dosud nepadlo slovo).

2.5 Generické metody

Taktéž jen na správné místo při definici (a volání) dáme špičaté závorky s parametrem označujícím typ:

```
public void genericky_vypis<T,U,V>(T prvni,U druhy,V treti)
{    Console.WriteLine(prvni);
     Console.WriteLine(druhy);
     Console.WriteLine(treti);
}
```

Tato funkce přijímá tři parametry a vypíše je. Můžeme ji tedy zavolat například takto:
`genericky_vypis<int,string,int>(1,"nazdar",2);`

V tomto příkladu konečně poprvé vidíme, jak se zachází s více typovými parametry. Funkce přijímá tři argumenty každý potenciálně jiného typu. Tyto argumenty vidíme ve špičatých závorkách oddělené čárkami.

2.6 Poznámky a odbočka k interfacům

Vyrobíme-li v C++ šablonu, překladač sleduje, s jakými parametry ji používáme a pro různé parametry vyrobí různé třídy. Generická třída v C# je však přeložena jen jako jedna třída. Jaké parametry jí předáme, se řeší až za běhu. To sice vypadá jako nepodstatná drobnost, ale důležitý rozdíl přichází ve chvíli, kdy s dotyčnými typy netriviálně pracujeme (například je porovnáváme, což bude dělat náš příklad). Překladač C++ může ověřit, zda porovnáváme pouze typy, pro které je porovnání definováno. Překladač C# to však ověřit nemůže, ale chce. Proto, kdybychom se pokusili definovat ve výše uvedeném příkladu s komplexními čísly funkci čísla porovnávající (například bychom chtěli přetížit operátory porovnání), nepůjde to. Dokud neslíbíme, že dotyčné typy musejí být porovnatelné. To zajistíme pomocí *interfaců*.

Interfacy (od nom. sg. "interface") alias rozhraní
`https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/interfaces/`

nahrazují v C# chybějící násobnou dědičnost. Ideově bychom je mohli popsat jako čistě abstraktní třídy (ale třídy to nejsou, protože od nich například nelze dědit). Můžeme u nich pouze deklarovat, o jaké metody se opírají. Třída, která prohlásí, že tento interface implementuje, pak musí implementovat všechny v tomto interfacu deklarované metody. Výhodou interfaců je, že jedna třída může implementovat neomezený počet interfaců (jelikož od nich se dědit nedá, nehrozí vznik diamantového problému) Budeme-li například chtít říct, že každá správná tiskárna umí tisknout, ale každá tiskárna bude tisknout jinak (a nechceme-li to dělat abstraktní třídou), můžeme definovat interface, který toto řekne:

```
interface tiskarna
{   void tiskni(string co);
}
```

Budeme-li pak chtít udělat jehličkovou tiskárnu (u které už víme, že při tisku střílí jehličkami), provedeme to takto:

```
class jehlickova\_tiskarna:tiskarna
{   public void tiskni(string co)
    {   Console.WriteLine("Tisknu jehlickami {0}",co);}
}
```

Některé interfacy jsou předpřipravené. Například `IComparable`. Tento interface stanoví, že dotyčný datový typ lze porovnávat. Porovnatelná jsou například čísla. V tomto interfacu se však nevynucují relační operátory, ale metoda `CompareTo` poměřující své údaje s údaji objektu zadaného jako parametr. Tato metoda vrací 1, je-li hodnota objektu, kterému voláme metodu `CompareTo` větší (nežli u toho, který dáváme jako parametr, -1 pokud menší a 0, pokud jsou si hodnoty rovné. Chceme-li výše uvedenou třídu `kompl` obohatit o porovnání, modifikujeme její významné části asi takto:

```
class kompl<T> where T:IComparable<T>
{    ...
     public static operator < (kompl<T> a, kompl<T> b)
     {    if(a.re.CompareTo(b.re)==-1)
               return true;
          if(a.re.CompareTo(b.re)==0 \&\& a.im.CompareTo(b.im)==-1)
               return true;
          return false;
     }
     // Zde je nutno analogicky definovat operator > !
}
...
static void Main(string[]args)
{    ...
```

```
    Console.WriteLine(a<b);
    ...
}
```

2.7 Výjimky

Již ze zimy víme, že když se výpočet nepovede, může funkce buďto spadnout (což není moc chytré), nebo vrátit specifickou hodnotu (což také není vhodné, protože to omezuje obor hodnot dotyčné funkce), anebo může vyvolat výjimku. V Pythonu se výjimky ovládaly pomocí klíčových slov `try, except, finally` a `raise`. V C# se analogicky používají klíčová slova `try, catch, finally` a `throw`. Syntax je do jisté míry podobná.

Příklad (podíl dvou čísel):

```
static void Main(string[]args)
{   int a=Convert.ToInt32(Console.ReadLine()),
        b=Convert.ToInt32(Console.ReadLine());
    try{
        Console.WriteLine(a/b);
    }
    catch(DivideByZeroException e)
    { Console.WriteLine("NELZE");}
}
```

V příkladu vidíme, že po načtení dvou celých čísel tato zkusíme vydělit. Při dělení může dojít k chybě (dělení nulou). Chyby výpočtu jsou v C# reprezentovány výjimkami. Dělení nulou je reprezentováno výjimkou typu `DivideByZeroException`.

Pravidla pro práci s výjimkami jsou tedy tato: Klíčové slovo `try` uvádí blok, který zkusíme vykonat. Klíčové slovo `catch` následuje za `try` blokem. Za tímto slovem řekneme, o výjimku jakého typu se má jednat a pod jakým jménem ji chceme zpracovávat (jako při běžné definici proměnných - proměnná výjimku reprezentující je objekt a tedy má atributy a metody). Následuje blok, který se má vykonat, pokud nastala výjimka dotyčného typu. Klíčové slovo `finally` uvádí blok finalizátoru. Tento blok se vykoná, ať výjimka nastala, nebo ne. To může být vhodné, potřebujeme-li zavřít, co jsme si otevřeli (a co za nás nikdo nezavře). Bez finalizátoru, kdyby nastala výjimka, by dotyčný zdroj zůstal viset otevřený, což by bylo nepříjemné především u dlouhohrajících procesů.

Klíčové slovo `throw` hodí výjimku. Syntakticky se ovládá stejně jako třeba klíčové slovo `return`, tedz n8sleduje popis házené výjimky.
Například: `throw new Exception("Spadlo ti to!");`

V C# lze jako výjimku hodit instanci jakékoliv třídy, která zdědila od třídy `Exception`. Třída `Exception` je společným praotcem všech reprezentantů výjimek. Chceme-li tedy zachytit jakoukoliv výjimku, stačí do popisu příkazu `catch` uvést jako typ `Exception`. Jak vidíme, vše je krásně metodické a na typech reprezentujících výjimky vzniká struktura daná dědičností.

Za `try`-blokem může následovat více `catch`-bloků. V takovém případě se postupně hledá první blok, jehož popis typu výjimky odpovídá. Tedy takový, kde jde o tentýž typ, nebo kde `catch` blok chytá jakéhokoliv rodiče hozeného typu (opět jde o Liskovové princip, kdy místo rodiče můžeme použít jakéhokoliv syna).

Z popisu obsluhy výjimky plyne mrzutá vlastnost: Co se stane, dáme-li `catch`-blok obsluhující potomkovský typ za blok obsluhující rodičovský typ? V tom případě bychom se k bloku obsluhujícího potomka nikdy nedostali. Aby se toto omylem nestalo, překladač kontroluje, aby bloky obsluhující potomky byly před bloky obsluhujícími rodiče (opak skončí chybou při kompilaci).

Není-li za současným `try`-blokem odpovídající `catch`-blok, výpočet (po případném provedení finalizátoru) vypadne ze současné funkce a propadá dále, dokud se neocitne v `try`-bloku, za kterým se dotyčná výjimka ošetřuje. Takový blok jistě existuje, v nejhorším případě je v zavaděči našeho programu. Poslední ovladač výjimky (který chytá typ `Exception`) je ten, který vám dosud hlásil chyby při běhu programu.

Šíření výjimek rozhodně neoplývá rychlostí. Proto tak k výjimkám přistupujte a používejte je jen výjimečně. Podrobnosti sdělí Pavel Ježek na pokročilé přednášce.