Co semestr dal

čili co se jinam nehodilo

- Delegáty,
- thready (a zamykání),
- hashování,
- string vs. StringBuilder.

Delegáty

významný rozdíl oproti Javě

- Potřebujeme předat funkci jako parametr (například při třídění porovnavač).
- V Javě nebyly, proto bylo nutné předat objekt vybavený správnou metodou.
- Jsou realizovány ukazately (pointery), ač jinak jsou pointery v C# dovedně zamaskované.
- Lze je použít i k definici anonymních funkcí.
- Jde tedy o datový typ reprezentující funkci (kterou lze ve vhodné chvíli zavolat).

Delegáty

příklady

```
public delegate int porovnej(object co, object scim);
// definujeme datovy typ reprezentujici funkci
public static int porovnej_cisla(object co, object
scim)
{   int a=(int)co,b=(int)scim;
    if(a>b)  return 1;
    else     if(a<b)  return -1;
             else   return 0;
}
public void setrid(object[]co, object[]scim,porovnej
por)
{   // zde implementujeme treba mergesort
    // a porovnavat budeme funkci por.
}
```

Delegáty

další příklad

Minule definovaný datový typ můžeme použít k definici nějakého ovladače (jako atribut do třídy přidáme proměnnou, kterou pak půjde volat: `porovnej porovnavadlo=porovnej_cisla;`

```
...
porovnavadlo(a,b);
...
```

porovnavadlo bude pracovat podle toho, jakou funkci do něj přiřadíme. Může tedy dělat různé věci (podle toho, co zrovna potřebujeme). Delegát může ovladače i řetězit:

```
porovnavadlo+=porovnej_cisla;// porovnej 2x za sebou.
```

nebo lze ovladač odebrat:

```
porovnavadlo-=porovnej_cisla;
```

Delegáty

potřetí

- Delegáty lze použít k definici anonymních funkcí, tedy funkcí, které nechceme pojmenovat.
- Abychom anonymní funkci dokázali použít, přiřadíme ji do delegáta:

```
static void Main(string[]args)
{  porovnej p=delegate(object co,object scim)
   {return 1;};
}
```

– porovnání, kde první prvek je vždy větší sice nedefinuje uspořádání, ale to nám při programování nevadí. :-)

Interfacy

zvané "styčno", "rozhraní" a námi provokatéry "meziksicht"

- Nahrazují chybějící násobnou dědičnost.
- Umožňují vynutit implementaci nějaké funkce.
- Vypadají jako abstraktní třída se všemi metodami abstraktními.
- Chceme-li je implementovat, syntakticky postupujmeme jako při dědičnosti.

Interfacy

příklad

```
interface tiskarna
{   void tiskni(string co);
    void znic_papir();
}
class jehlickova_tiskarna:tiskarna
{   public void tiskni(string co)
    {    Console.WriteLine("Rachtam O",co);     }
    public void znic_papir()
    {   Console.WriteLine("Tiskovou hlavou sapu\
                papir!");
    }
}
```

Každá správná tiskárna umí tisknout a zničit papír. I ta jehličková.

Interfacy

příklad

```
interface tiskarna
{   void tiskni(string co);
    void znic_papir();
}
class neexistujici_tiskarna:tiskarna
{   public void tiskni(string co)
    {    Console.WriteLine("Rachtam 0",co);     }
}
```

Tohle nezkompilujeme, protože tiskárna, která neumí zničit papír, neexistuje! :-)

Thready

vyřeší některé vaše problémy

- Dva procesy vedle sebe v systému si umíme představit.
- Thready vypadají podobně – ale sdílejí paměť. Patří tedy jednomu procesu.
- Thread funguje podobně jako program, ale ne nutně s metodou `Main`, tedy řekneme, která funkce má proběhnout v novém vlákně.
- Vlákna použijeme například u paralelizovatelných problémů, máme-li vícejádrový systém.
- Anebo se hodí při práci s formulářovými a androidími aplikacemi, chceme-li spočítat něco, co trvá.

Thready

ve formulářových a androidích aplikacích

- Formulář se nepřekreslí, dokud ovladač nedoběhne.
- My ovšem potřebujeme pracovat kontinuálně (a překreslovat).
- $\Rightarrow$ vejce a slepice problém.
- Vyřešíme tak, že výpočet realizujeme ve zvláštním vláknu, zatímco to hlavní skončí.
- Formulář se překreslí. Pozaďové vlákno naopak nesmí vypisovat (na formulář). Má-li tedy spočteno, pošle popředovému vláknu zprávu.
- Na Androidu popředové vlákno nesmí dělat trvající věci (například číst ze sítě).

Thready

příklad

```
using System.Threading;//jinak zajímavě dopadneme
string co;
public void tiskni()
{   for(int i=0;i<co.Length;i++)
    {   Thread.Sleep(5000);
        Console.Write(co[i]);
}   }...
co="dlouhy text, co se bude dlouho tisknout";
Thread t=new Thread(tiskni);// zde je delegát!
t.Start();
```

Abychom nečekali do konce tisku, vytvoříme vlákno, které funkci tiskni zavolá. Definujeme proměnnou thread reprezentující, konstruktoru předáme funkci k zavolání a thread rozběhneme.

Thready

poznámky

- Třída Thread umožňuje programu počkat stanovenou dobu. Přesněji zbrzdí současné vlákno.
- Thready sdílejí paměť, takže je třeba dát pozor, aby někdo nezapsal do proměnné `co`, zatímco funkce `tiskni` běží.
- Teorie vyučována na principech počítačů, základech překladačů,... – synchronizační primitiva, zamykání (zámek, semafor,...), deadlock (problém uváznutí a Coffmannovy podmínky), race condition,...
- V C# funguje zamykání pomocí `lock(co){zamceny_blok();}` .
- Jako parametr lze předat objekt. Objekt lze zamknout jednou. Pokusí-li se další proces získat zámek, bude čekat, až vlákno držící zámek tento odemkne.

Asynchronní volání

a čekání na jeho výsledek

- Funkce, které trvají, můžeme zavolat asynchronně a čekat na jejich výsledek.
- Klíčovým slovem `await` před zavoláním funkce oznámíme, že se nemá čekat, až funkce doběhne (kód může pokračovat dál).
- Klíčovým slovem `async` při definici funkce oznámíme, že tuto funkci má smysl volat asynchronně.
- Asynchronně volanou funkci také můžeme rozvrhnout jako `Task`.

Asynchronní volání

příklad

```
string co="dlouhy text k tisku";
Task<bool>t=tiskni();
bool vysl=await t;
Task<bool>u=uvar_kafe_precti_noviny();
bool docteno=await u;
if(vysl&&docteno)
    Console.WriteLine("To se mame!");
```

Funkce `tiskni` je nám známa tím, že trvá dlouho. U té druhé funkce si totéž umíme představit. Proto je spustíme asynchronně.

Async

příklad

```
async void pekelne_dlouha_funkce()
{...}
```

Asynchronní funkci definujeme snadno. Smysl to ale má jen tehdy, když se v ní objeví `await`, protože až tam dojde k přerušení výpočtu (a vyvětvení vlákna, na které má smysl čekat).

StringBuilder

a jeho využití

- `string` je pěkný datový typ, který se dá využít jako pole charů, ale jen pro čtení.
- Chceme-li do něčeho podobného i zapisovat, použijeme `StringBuilder`.
- Do StringBuilderu lze zapisovat a lze z něj nechat vytvořit string. Ale string to není.

Hashování

bude jen velmi stručně zmíněno

- Znáte z ostatních kurzů (Algoritmizace, ADS I,...).
- Máme pár prvků z velkého univerza (například jména všech studentů).
- Chceme je uložit do pole, ale stringem nelze indexovat, proto definujeme hashovací funkci.
- Navrhneme-li ji správně, ušetříme množství času.

Hashování

umí být perfektní, anebo je to také pěkný prevít

- Ďábel se skrývá v detailu, tedy jak navrhnout hashovací funkci.
- Třeba jako součet ascii-hodnot znaků jména. Ale je taková funkce dobrá?
- Perfektní (bezkolizní) nejspíše nebude. Kolize lze řešit buďto založením spojového seznamu v každé hashovací přihrádce.
- Jinou možností je kukaččí hashování, tedy prvek dáme do jiné přihrádky (než kam by patřil).
- Při kukaččím hashování je téměř neimplementovatelný delete!
- Nesprávnou přihrádku lze určit mnoha způsoby (následující, správná+hodnota další hashovací funkce),...
- Hashování bylo významně zkoumáno (Knuth či Mehlhorn, Sanders: Algorithms and Data Structures – správný díl).

Konec

zavíráme a jdeme domů – počkat, vždyť už jsme doma

Děkuji za pozornost. Dotazy? Mailem!