C#

24. ledna 2017

Obsah

1 20. 1. 2016 **2**
- 1.1 1 . . . 2
  - 1.1.1 A . . . 2
  - 1.1.2 B . . . 2
- 1.2 2 . . . 2
- 1.3 3 . . . 2
  - 1.3.1 A . . . 2
  - 1.3.2 B . . . 2
- 1.4 4 . . . 3
- 1.5 5 . . . 3
- 1.6 6 . . . 3
  - 1.6.1 A . . . 3
  - 1.6.2 B . . . 4
- 1.7 7 . . . 4

2 27. 1. 2016 **4**
- 2.1 2 . . . 4
  - 2.1.1 A . . . 4
  - 2.1.2 B . . . 4
  - 2.1.3 C . . . 4
- 2.2 3 . . . 5
- 2.3 4 . . . 5
  - 2.3.1 A . . . 5
  - 2.3.2 B . . . 5
- 2.4 5 . . . 5
- 2.5 6 . . . 5

3 1. 2. 2016 **6**
- 3.1 2 . . . 6
- 3.2 3 . . . 6
- 3.3 4 . . . 6
- 3.4 5 . . . 6
- 3.5 6 . . . 6

3.6 7 7
3.7 8 7

20. 1. 2016

1

A

Ano, dalo by se to zapsat prakticky identicky, jen bez použití static (ve většině případů), a instance třídy, která má být společná, by se vždy předávala v konstruktoru ostatním třídám, které by si udržovaly na tuto instanci pointer.

B

Naše řešení by mělo výhodu, že bychom pro každou skupinku tříd mohli nadefinovat společnou jednu třídu → dělili bychom to po skupinkách, jak bychom chtěli, neházeli bychom je do stejného pytle.

2

Tato implementace **není** vhodná. Důvody k tomu jsou, že neimplementuje zároveň metodu equals, která je pro rozumné zacházení s touto třídou potřeba (jsou rozhraní, co na to spoléhají).
Problém v této funkci dále je, že bude docházet často ke kolizím, tedy místo konstantního přístupu k položce bude až lineární, protože věk dětí je relativně malé rozmezí (zvláště bereme-li ho jako celé číslo).
Dalším problémem je, že věk rozhodně není *immutable*, takže se nedá vhodně použít například jako klíč v *Dictionary*, protože změní-li se věk, změní se nenávratně informace o tomto dítěti ve slovníku.

3

A

Je mezi nimi rozdíl. `TimeUtilsA.InFiveMinutes` vždy k současné hodnotě času přičte 5 minut. `TimeUtilsB.InFiveMinutes` toto provede jednou při zavolání statického konstruktoru, a potom už tuto funkci bude znovu a znovu vracet.
První varianta je očividně vhodnější.

B

Tím, že vždy vrací hodnotu struktury, nám říkají, že s tím můžeme dělat, co chceme.
To, že ty vlastnosti mají jenom getter znamená, že se tam dá přiřadit hodnota pouze v konstruktoru.
Tímto se zamezilo tomu, že se nám objekt bude "modifikovat pod rukama", a můžeme ho použít jako referenční čas.
To znamená, že vývojáři tento objekt navrhovali jako *immutable*.

4 UPRAVENO

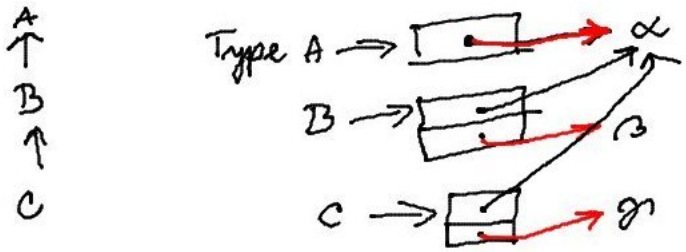
```
1
1
```

Co se zavolá se určuje za překladu, generická metoda tedy bude volat generickou.

5

Verifiable code je kód, u kterého může kompilátor ověřit, že nepoužívá *unsafe* metody jazyka, tedy například Céčkovské pointery, kterými by mohl sahat na adresy, kam nemá. Tedy *unverifiable code* je kód takový, který je používá.

6




```
-9
```

A

Každá třída obsahuje pointer na Type, který ukazuje na VMT. Každá zděděná třída kopíruje VMT rodiče, a pak s ním něco dělá.
Override metody řekne: "mám lepší implementaci, než byla v předkovi". Tedy pointer z mé VMT bude ukazovat na mou implementaci metody na loader heap.
Nenapíšu-li toto (nebo napíšu-li u nové virtuální metody "new"), tak tím pouze ke zkopírované VMT z rodiče přidávám svůj další záznam.
Vytvořil jsem metodu C, a přiřadil jsem ji do pointeru na třídu A, jež je předkem C. Tedy řekl jsem, že na C se budu dívat jako na třídu A.
Při zavolání funkce se tedy ve třídě A podívám, kolikátá položka ve VMT odpovídá metodě `m`. Té odpovídá první metoda v pořadí. Přestože ukazuji na typ C, virtuální volání na první položku tabulky zavolá implementaci α.

B

Ano, lišilo.
Chování virtual a override vysvětleno výše. Klíčovým slovíčkem override říkáme, že máme lepší implementaci. Když toto override nepoužijeme, tak pouze přidáváme novou implementaci. Zvyšuje to přehlednost kódu.
Zavolala by se implementace γ.

7

```
a is 5
```

Při překladu dojde k určení proměnné x, bude to tedy string.
Za běhu dojde na zavolání funkce `string.Format()`, která dosadí za proměnnou *a* číslo 5. Toto dosazení je napevno (a navíc string je *immutable*), takže se nedá dále změnit.
Nakonec dojde k zavolání funkce Console.WriteLine() tak jak ji známe.

27. 1. 2016

2

A

Tato část není vhodně navržená.
Nechytáme případné výjimky, takže nám program může spadnout na neošetřenou výjimku.
Dále načítáme celý soubor do proměnné typu string. Ta nemusí být schopná tak velký element pojmout, protože v C# je limit na maximální velikost i na large object heap.
Jako pozitivum bych vzal jednoduchost kódu.

B

Ano, je to rozdíl.
Tím, že jsme do něj přiřadili null, jsme označili data za neplatná, tedy může se na ně na soh pustít garbage collector.
Byl-li tento objekt na large object heap, pak se jeho dealokací uvolní spousta paměti. Kdyby se tak nestalo, tak protože nevíme přesně, co se děje v této funkci, tak by mohla přetéct.
Nebyl-li na large object heap, pak se může v libovolný čas zcollectovat. Protože na něj nevede už žádná reference, určitě se v příštím garbage collectoru zcollectuje. Kdybychom tedy v metodě `CountWord()` alokovali nějaké objekty na haldě, v příští kolekci nebudeme muset "přemapovávat"adresy až tolika objektů na haldě.

C

```
string[] Split(params char[] delimiters);
```

Z tohoto příkladu to plyne, ve skutečnsoti to může vypadat jinak.

3

Je to možné.
Klíčové slovíčko u hodnotových typů znamená pouze inicializaci objektu, tedy proměnná *b* se inicializuje na defaultní hodnotu, tedy false.

4

A

Tato metoda nemá setter, tedy lze inicializovat pouze v konstruktoru instance. Vlastnosti by se měli používat jen pokud mají stejný access time jako normální datové položky. Protože tuto hodnotu inicializujeme pouze v konstruktoru a pak ji už jen vracíme, odpovídá této myšlence, a je lepší ji použít.

B

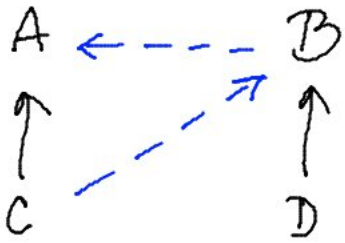
Typ ulong narozdíl od long není CLS compliant, tedy neumí s ním pracovat ostatní jazyky .NETu, stejně jako některé knihovny. Long je 64-bitové znaménkové číslo, takže by mělo snad pro všechny soubory být dostatečně velké, a tedy stačí, tedy není důvod použít něco, s čím se nám bude hůře pracovat.

5

```
111111
```

Při zavolání konstruktoru X dojde k vypsání "1". Konstruktor voláme při vytváření instance X. To děláme v cyklu, který má 3 iterace, v každé iteraci 2 krát.
Při provádění `X[] a` se volá konstruktor pole, ale ne samotného X, proto se to na vypisování neprojeví.

6



Překladač určí na zavolání funkci, co nejvíce odpovídá parametrům. Může použít nejvýše jednu uživatelsky definovanou konverzi a jednu implicitní konverzi.
Dojde k zavolání metody s parametrem A, protože na funkci A postačí jedna implicitní konverze, zatímco na ostatní implicitní konverze ani neexistují.

1. 2. 2016

2

Dispose se volá ve finalizerovi. Nejspíš v StreamWriteru je implementovaný bez flushování věcí, co zbylo v bufferu, zatímco ve FileStreamu je to na implementováno i s tím.
Dalo by se to spravit, aby StreamWriter ve finalizeru před uzavření file deskriptoru flushnul buffer,

a až potom to zavřít. Jako uživatel to ale nemůžeme udělat, protože napíšeme-li k tomu nějaký finalizer my, tak protože se do finalizační fronty vkládají elementy vesměs v náhodném pořadí, nevíme, jestli se nejdříve zavře file deskriptor a potom se budeme snažit flushovat, nebo naopak. Museli by to tedy opravit vývojáři MS.

3

Interfacová tabulka u funkce C se za překladnu musí rozhodnout, kterou funkci volat. Bude chtít volat funkci s parametrem object, která se jmenuje `m`.
Program skončí zavolá funkci B, protože ta překrývá objekty funkce A, a překladač se rozhodl, že nejvíc sedí.

4

Zavolá se funkce a parametrem double, protože dojde k implicitní konverzi z int na double.

5

Postupuje se:

1. Nalezení obsluhy výjimky
2. Probublávání k obsluze

```
System.Exception.ArgumentNullException
System.Exception.DivideByZeroException
```

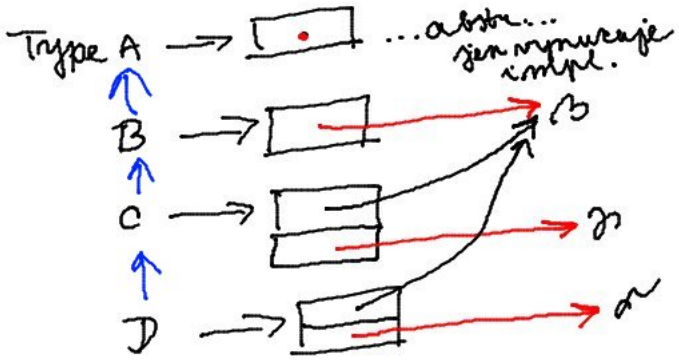
6

Struct je ValueType, class je ReferenceType.
Struct se tedy základně ukládá celý na zásobník, zatímco u class se ukládá při konstrukci na zásobník jen pointer, kde zbytek třídy se posílá na haldu.
Při zavolání `S s = new S()` dojde k zavolání konstruktoru na zásobníku, zatímco při zavolání `C c = new C()` dojde k naalokování prostoru na haldě a k zavolání konstruktoru.
Struct by se mělo používat, dává-li sémanticky smysl, aby se k ní přistupovalo jako k hodnotovému typu (př. zlomek, komplexní číslo, vektor). Přístup ke struktuře bude rychlý, nebude se muset používat garbage collector. Dále platí, že struktura by neměla být příliš velká, protože by mohlo dojít při alokování velkých věcí na zásobníku brzy ke stack overflow.

7

`Checked` znamená, že dojde-li k aritmetickému přetečení, vyhodí se odpovídající výjimka (kterou v tomto příkladě necatchujeme).
Hodnota `0x7FFFFFFF` je maximální hodnota int. Ve funkci po přičtení jedničky dojde jistě k aritmetickému přetečení, takže to vyhodí odpovídající výjimku, která není zachycená, takže se vypíše na obrazovku a program skončí. `Console.Write()` v programu se tedy nevypíše.

8



Vezmeme `a`, což je poitner na A typu D, zkonvertujeme ho nejprve na C, pak na B. Konverze z C na B je jistě validní, otázka je, zdali může selhat konverze a na C. Protože a je ve skutečnosti typu D, nemůže, takže dojde v pořádku ke konverzi z D na C, a potom na B. Budeme mít tedy pointer typu B, ze kterého budeme volat metodu `m()`. Při výběru nejvhodnější funkce pro zavolání se tedy určí první záznam VMT, a zavolá se na prvku typu D. Tedy vypíše se:

```
8.B
```