Programovanie v jazyku C#

Generické programovanie

prednáška 7

Ing. Ján Magyar, PhD.

ak. rok. 2023/2024 ZS

Opakovanie

- downcasting
- problémy s downcastingom (pretypovaním)

Riešený problém

- niektoré algoritmy fungujú rovnako bez ohľadu na použité typy
  - triedenie vyžaduje iba možnosť porovnávania dvoch objektov
  - každá údajová štruktúra definuje základnú funkcionalitu, ktorá funguje rovnako pri rôznych konkrétnych typoch prvkov
- efektívna práca s objektmi rôznych typov ale so spoločným rozhraním
- v iných jazykoch riešený cez makrá

Boxing

- hodnotový typ zabalíme do typu `object` (alebo do iného rozhrania)
- hodnota sa ukladá namiesto zásobníka v dynamickej halde
- v zásobníku budeme mať smerník na novovytvorenú premennú
- implicitná operácia
- podpora pre jednotné spracovanie: všetko môže byť `object`

Unboxing

- extrahujeme hodnotu z objektu do hodnotového typu
- explicitná operácia
- dva kroky
  - kontrola, či požadovaná hodnota je správneho typu
  - kopírovanie hodnoty do novej premennej
- ak hodnota je prázdna, vygeneruje sa `NullReferenceException`
- ak hodnota je nesprávneho typu, vygeneruje sa `InvalidCastException`

Boxing a unboxing

- výpočtovo náročné operácie
- pri boxingu musíme vytvoriť nový objekt s alokáciou pamäte (cca 20x)
- pri unboxingu dochádza k pretypovaniu (cca 4x)

Ukážkový problém

- potrebujeme vytvoriť kontajnerový objekt pre objekty rôznych typov
- konkrétna implementácia kontajnera nie je podstatná
- naša implementácia by mala byť všeobecne použiteľná
- ako sme už videli, nemôžeme použiť spoločné rozhranie `Object`

Nedostatky spoločného rozhrania

- použitie polymorfických referencií
- potrebujeme pretypovanie počas behu - môže generovať chyby
- čo ak náhodou pridáme objekt nesprávneho typu?
- downcasting funguje iba ak pretypujeme do pôvodného typu vytvoreného objektu
- riešenie - všeobecná implementácia, pri použití ale špecifikujeme konkrétny typ prvkov

Generické programovanie

- paradigma pre znovupoužiteľné a efektívne riešenie s výhodami typovej kontroly
- **lifting** - podpora abstrakcie pre vytvorenie všeobecnej implementácie algoritmu pre viaceré konkrétne implementácie
- **concepts** - abstrakcie musia spĺňať požiadavky jednotlivých konceptov

Podpora generického programovania v OOP

- abstrakcia a enkapsulácia
- dedenie
- subtype polymorfizmus
- šablóny a generiká - triedy, rozhrania a metódy s typovými parametrami

Typové parametre v triedach

- parametrický polymorfizmus
- generické triedy špecifikujú všeobecné správanie, ktoré bude fungovať s ľubovoľným dosadeným typom - nemôžeme sa spoliehať na špecifickú funkcionalitu typu
- generické triedy sú definované
  - generickými atribútmi - nie je špecifikovaný konkrétny typ
  - generickými metódami - nie je špecifikovaný presný návratový typ
- pri vytvorení inštancie určíme konkrétny typ

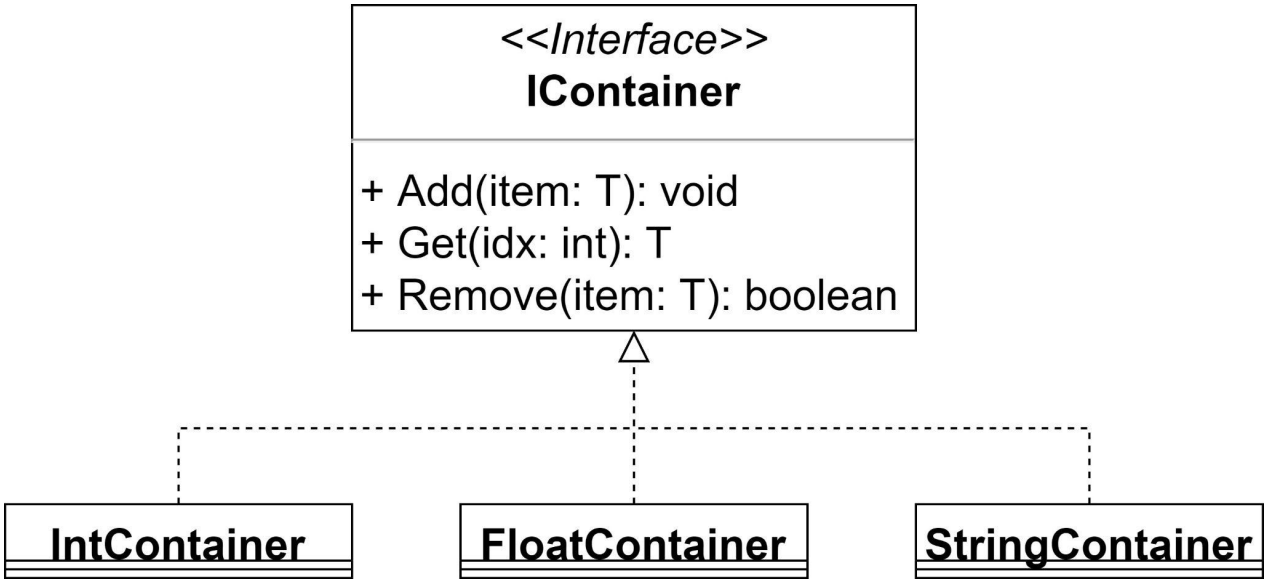
Ukážka - kontajner

- `GenericList` je generická trieda s typovým parametrom T
- nepotrebujeme použiť pretypovanie, keďže pri konkrétnom použití už pracujeme s konkrétnym typom
- keby sme chceli použiť nesprávny typ, chyba sa vygeneruje počas kompilácie - výhoda statickej kontroly

Generické rozhrania

- najmä pre podporu jednotného rozhrania pre špecifické kontajnerové triedy
- nepotrebujeme boxing a unboxing
- podpora viacnásobných ohraničení cez viacnásobnú implementáciu

Generické rozhrania - ukážka



Generické metódy

- pre ľubovoľnú metódu
- zápis podobný ako pri triedach a rozhraniach
- volania rovnaké ako pri negenerických metódach
- v generických triedach môžeme limitovať prístup k metódam iba pre určité dátové typy
- môžu byť aj statické

Wildcardy pri typových parametroch

- v Jave môžeme typové parametre zapísať nasledovne:
  - `?` - ľubovoľný typ (any)
  - `? extends T` - podtypy typu T
  - `? super T` - nadtypy typu T
- C# takú možnosť neponúka - čiastočná podpora cez generické triedy, generické rozhrania a boxing/unboxing
  - `where T : System.IComparable<T>`

Ohraničenia

`where T: struct` - hodnotový typ

`where T: class` - referenčný typ

`where T: class?` - referenčný typ (aj nullovateľný)

`where T: notnull` - nenullovateľný typ (hodnotový alebo referenčný)

`where T: unmanaged` - nenullovateľný nemanažovaný typ

`where T: IFoo` - typ implementujúci rozhranie `IFoo`

`where T: Foo` - typ je podtriedou `Foo`

`where T: new()` - typ musí mať bezparametrový konštruktor

`where T1: T2` - `T1` je podtypom generického typu `T2`

Subtyping v generickom programovaní

- ak máme triedu B, ktorá je podtypom triedy A
- bude zoznam `List<B>` podtypom zoznamu `List<A>`?
- `List<B>` **≤** `List<T> where T: A`

Výhody generického programovania

- vyššia efektivita (nepotrebujeme pretypovanie a boxing)
- znovupoužitie kódu
- typová kontrola
- akurát určené pre kontajnery a kolekcie
- zvyčajne nepotrebujeme vytvoriť vlastné generické triedy

Generické triedy v C#

- definované v mennom priestore `System.Collections.Generic`
  - `Dictionary<TKey, TValue>`
  - `HashSet<T>`
  - `LinkedList<T>`
  - `Queue<T>`
  - `Stack<T>`

otázky?