Programovanie v jazyku C#

Dedičnosť a polymorfizmus

prednáška 3
Ing. Ján Magyar, PhD.
ak. rok. 2024/2025 ZS

Motivácia

znovupoužiteľnosť - nechceme viackrát napísať ten istý kód

rozšíriteľnosť - rozširujeme funkcionalitu, zaručená typová konzistencia

Dedičnosť

nové triedy vytvárame použitím už existujúcich tried

nová trieda znovupoužíva, rozširuje alebo modifikuje správanie pôvodnej triedy

pôvodná trieda sa nazýva: **base** (základná), **super** (nadtrieda), **parent** (rodičovská)

odvodená trieda sa nazýva: **derived** (odvodená), **sub** (podtrieda), **child** (dcérska trieda, potomok)

Dedičnosť z pohľadu typov - subtyping

ak B je podtypom A ($B \leq A$), všade kde kód očakáva objekt typu A, môžeme použiť objekt typu B

podtrieda vždy rozširuje nadtriedu, ale zachová jej základné vlastnosti

napr.
```
class Student : Person
Person p = new Student();
Student s = new Person();
```

Typy subtypingu

implementácia rozhraní (neskôr)

dedenie od triedy

- aj rozhranie môže dediť od rozhrania (neskôr)

Dedičnosť v C#

iba od jednej triedy

tranzitívna: ak $D \leq C \leq B \leq A$, tak $D \leq A$

nie všetky členy tried vieme dediť:

- statické konštruktory - inicializuje statické premenné
- konštruktor - na vytvorenie novej inštancie triedy
- finalizér - na uvoľnenie miesta v pamäti po zániku objektu

ostatné členy sú prístupné na základe modifikátorov prístupu

Modifikátory prístupu pri dedení

- `private` - neprístupné v odvodených triedach (okrem vnorených)
- `protected` - prístupné iba v odvodených triedach
- `internal` - prístupné v triedach z rovnakej assembly
- `public` - prístupné aj v odvodených triedach

Praktická dedičnosť v C#

`virtual` - umožňuje prepísanie členov (`public` alebo `protected`)

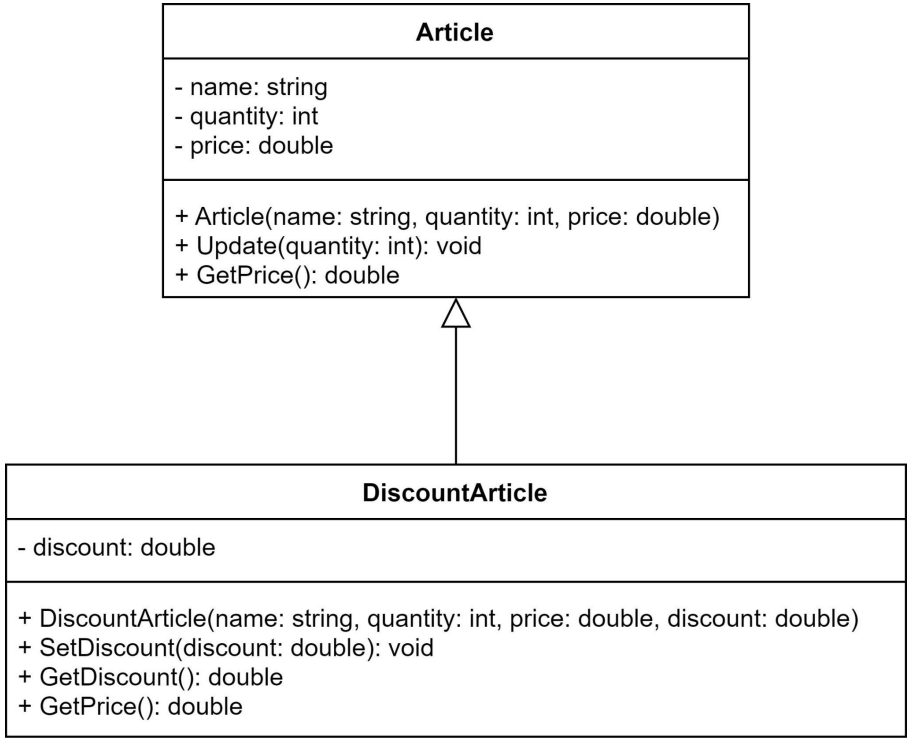
`override` - prepisujeme metódu z nadtriedy

`new` - skryjeme metódy nadtriedy

`abstract` - musia byť prepísané v podtriedach (iba v abstraktných triedach)

`sealed` - od danej triedy nemôžeme ďalej dediť, resp. nemôžeme prepísať danú metódu

Dedičnosť - príklad



Konštruktor podtriedy

musíme definovať vzhľadom na povinný názov

vždy sa zavolá defaultný konštruktor nadtriedy

ak chceme špecifikovať iný konštruktor, môžeme použiť kľúčové slovo `base`

v C# sa to uvádza po deklarácii konštruktora

```
public Student(string name, int age, int
year) : base(name, age) { }
```

Kľúčové slovo `base`

umožňuje prístup k prepísaným členom nadtriedy

```
public double GetPrice() {
   return base.GetPrice() * (1.0 - this.discount);
}
```

zavolá konštruktor nadtriedy

Výhody dedičnosti

dokážeme zadefinovať nové triedy jednoducho, využitím už implementovanej funkcionality

podpora polymorfizmu

- rôzne pohľady na rovnaký objekt
- inštancie podtried môžeme považovať za inštancie nadtried (s iným rozhraním)

Best practice

definícia nových vlastností a správaní

- nová funkcionalita je dostupná iba v podtriede
- typ objektu musíme vyberať pozorne

prepisovanie existujúceho správania

- vždy sa použije implementácia podľa zavolaného konštruktora
- v rôznych OO jazykoch je umožnené rôznymi spôsobmi

Polymorfizmus

na jeden objekt môžeme pozerať rôznymi spôsobmi

hlavné typy:

- subtyping (počas behu)
- parametrický (počas kompilácie) - generiká

ďalšie typy:

- ad-hoc - preťažovanie metód
- pretypovanie - napr. `int` na `float`

Binding

volanie metódy je v istom momente pridelené (*bound*) istej definícii metódy

zvyčajne počas kompilácie

v prípade polymorfizmu dokážeme urobiť iba počas behu

early a late binding

Early binding

pri parametrickom polymorfizmu

generiká - typ ako parameter

premenné nahradíme konkrétnymi typmi počas kompilácie

napr. zoznamy

Zdanlivý a reálny typ

zdanlivý typ (**apparent**) - podľa deklarácie, nemenný

reálny typ (**actual**) - pre objekt, podtyp zdanlivého typu

typová kontrola sa uskutoční na základe zdanlivého typu

binding sa robí na základe reálneho typu

Late binding

pri subtypingu - použitie jedného typu tam, kde sa očakáva iný typ

pri prepisovaní metód:
- názov metód a ich popis (typ a počet parametrov) musí byť rovnaký
- kompilátor rozhodne počas behu programu, ktorú definíciu zavolá

vyhľadávanie začneme v reálnej triede a postupujeme hore hierarchiou

Výhody dedičnosti

jednoduchá implementácia nových tried

väčšina funkcionality by mala byť zdedená

jednoduchá modifikácia zdedenej implementácie

Nevýhody dedičnosti

porušuje zapuzdrenie

white-box - často vidíme interné detaily nadtriedy

potreba aktualizovať podtriedy ak zmeníme nadtriedu

zdedenú funkcionalitu nedokážeme zmeniť počas behu

Znovupoužiteľnosť pomocou kompozície

druhý spôsob zabezpečenia znovupoužiteľnosti

dnes sa preferuje kompozícia namiesto dedičnosti

dedičnosť stále má svoje opodstatnenie!

Kompozícia

nová funkcionalita sa zabezpečí vytvorením objektu, ktorý sa skladá z ďalších objektov

nová funkcionalita je zabezpečená cez delegovanie komponentom

typy:

- **aggregation** - objekt vlastní druhý objekt, alebo sa skladá z osobitných objektov
- **containment** - ku komponentu vieme pristupovať iba cez kontajner

Výhody kompozície

ku komponentom pristupujeme výlučne cez ich rozhranie

black-box - nepoznáme interné detaily

zapuzdrenie je zabezpečené

menej závislostí (*dependencies*)

každá trieda rieši jednu jedinú úlohu

kompozíciu dokážeme zadefinovať počas behu

Nevýhody kompozície

viac objektov

rozhrania objektov musia byť presne navrhnuté

Coadove pravidlá

použi dedičnosť iba ak:

- podtrieda reprezentuje špeciálny typ a nie rolu
- inštancia podtriedy sa nikdy nestane objektom inej triedy
- podtrieda rozširuje a nie prepisuje úlohy nadtriedy
- podtrieda nerozširuje iba utility triedu
- podtrieda bližšie špecifikuje rolu, tranzakciu alebo zariadenie z aplikačnej domény

Dedičnosť a kompozícia

obe sú cenné spôsoby znovupoužitia kódu

dedičnosť sa používala príliš často v minulosti

návrhy sú viac znovupoužiteľné a jednoduché vďaka kompozícii

komponenty vieme rozširovať cez dedičnosť

všeobecne sa preferuje kompozícia

otázky?