

celkové o jazyku
C#

Generické funkce a třídy

↳ umožňují definovat třídy s typovými parametry, ze kterých lze dosadit později:

↳ zástup pro konkrétní datový typ až při použití

↳ lze jen jeden kód použít pro více typů, a lze pro typovou bezpečnost definovat interface, který musí splňovat např.

→
typová bezpečnost

↓
pomocí where $T: \text{IComparable} < T >$

Delegáty

↳ preferenční datový typ, co zapouzdří odkaz na metodu

↳ typově bezpečný ekvivalent ukazatele na funkci

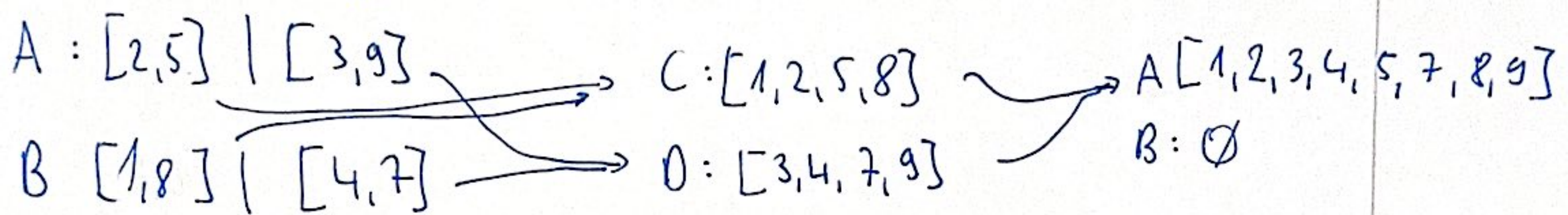
Multicasting

↳ pomocí + nebo += lze nanést více metod

→
Action
↳ delegát, co
vrací void

VNEJŠÍ TRÍDĚNÍ

- pokud je potřeba setřídit větší množství dat než je velikost RAM
- nemění se v počtech instrukcí procesoru, ale v počtech diskových operací
- používáme 2 soubory
 ↗ setříděné údaje v paměti (disku)



Nastane setřídění údajů - ^{skládání} paměti RAM a pak skládání

K-cestné skládání více souborů

- setřídí se do k souborů a načta si buffery aby se případný seek amortizoval

Počet binárních stromů o n prvcích → z dynamického programování

- o 0 prvcích → 1
- o 1 prvek → 1
- o 2 prvky → 2
- o 3 prvky → 5
- o 4 prvky → → 14

- o n prvcích → levý o i uzlech, pravý o n-1-i uzlech

tedy o 4 prvcích, tak $C_3 \cdot C_0 + C_2 C_1 + C_1 \cdot C_2 + C_0 \cdot C_3 = 5 \cdot 1 + 1 \cdot 2 + 2 \cdot 1 + 1 \cdot 5 = 14$

$$\frac{1}{n+1} \cdot \binom{2n}{n}$$

↗ tam probíhá
ženi sekvenční

↗ jinak předpokládám
víc disků nebo pásek

Dynamické programování

↳ rozdělení na menší podúlohy

↳ od rozdělení a ^{z výhledu rozdělil} postup se liší v tom, že někdy lze

rozdělit, a proto zkoušíme všechny možnosti a vybereme nejlepší

↳ memoizace (rekurze s ukládáním mezivýsledků)

↳ bez rekurze = dynamické programování

Minimax a α - β průřezování

MAX $\rightarrow$ snaží se maximalizovat hodnotu uzlu

MIN $\rightarrow$ minimalizovat hodnotu uzlu

α $\rightarrow$ max. zaručená hodnota, co má hráč max. zaručené - kořen, default $-\infty$

β $\rightarrow$ min. zaručená hodnota, pro hráče min. zaručené - hořeni, default $+\infty$

Pokud $\alpha \geq \beta$, tak se další větve neprozkoumávají, protože soupeř by do té větve stejně nepustil

Sealed

tridy $\rightarrow$ netze od nich dále dědit

metody a ostatní stavy $\rightarrow$ přestanou být virtuální, lze jen přepsat pomocí new

Rozdělení dětí do skupin

6 dětí $\rightarrow$ rozdělení do 3 skupin

Podle rozdělení do skupin:

$$\begin{aligned} 4, 1, 1 &\rightarrow \binom{6}{4} \binom{1}{1} \binom{1}{1} = 15 \\ 3, 2, 1 &\rightarrow \binom{6}{3} \binom{3}{2} \binom{1}{1} = 60 \\ 2, 2, 2 &\rightarrow \binom{6}{2} \binom{4}{2} \binom{2}{2} = 15 \end{aligned} \quad \left. \vphantom{\begin{aligned} 4, 1, 1 \\ 3, 2, 1 \\ 2, 2, 2 \end{aligned}} \right\} 90$$

$P_{D,S}$ - počet způsobů rozdělení D dětí do S skupin

$$P_{0,1} = 1$$

$$P_{0,0} = 1$$

$\nwarrow$ přidat do jiné skupiny

$\swarrow$ založit vlastní skupinu

$$P_{D,S} = P_{D-1,S} \cdot S + P_{D-1,S-1}$$

počet přijde $\rightarrow$
další dítě

Q-dy byla vytvořena nová skupina,
tak se $D-1$ dětí musí rozdělit do
 $S-1$ skupin

Mrkav a petržel

$\hookrightarrow$ počet posloupností k , kde nemusí být 2 mrkve vedle sebe

$\swarrow$ počet, když poslední je petržel

$$Z_k = M_k + P_k$$

$\nwarrow$ celkem $\nwarrow$ počet, když poslední je mrkev

$$M_1 = P_1 = 1$$

$$M_k = P_{k-1}$$

$$P_k = M_{k-1} + P_{k-1}$$

$\nwarrow$ pro konci mrkvi

$\nwarrow$ pro konci petrželi

Násobení matic

$\rightarrow$ nejméně násobení k vynásobení matic
rozměrů, projde všechny rozdělení na součin
dvou matic a vybere optimální

to, co používají ostatní
části programu

Žapouzdření

- odděluje rozhraní a implementaci

- většinou jsou veřejné pouze - metody

- vlastnosti

- výhody: objekt je vždy v konzistentním stavu (udržováním invariantů)

- properties

↳ get, set

→ pro uživatele se chová jako datová složka,

lze specifikovat jen read-only nebo write-only nebo
obojí

OOP

- objektově orientované programování

- Filosofie OOP a dekompozice

- striktní oddělení dvou rovin:

1. Rozhraní - jak lze s objektem komunikovat

2. Implementace - jak to funguje uvnitř

- Objekt sdružuje data i kód

- Třída (class) je abstraktní předpis, Instance (Objekt) je konkrétně alokovaná v paměti

Ochrana přístupu

- private → přístup jen uvnitř téže třídy (v potomcích nejsou přímo přístupné)

- protected → přístup jen dané třídě a jejím potomkům

- internal → omezené na assembly, (tedy DLL, exe po zkompilování jeden nebo více kódů)

- public → neomezený přístup

VMT (virtual method table)

↳ VMT

proto může starší
třída zavolat metodu
novější třídy

- Každá třída má tabulku virtuálních metod, kterou vytvoří překladač

- VMT obsahuje adresy virtuálních metod

dezaduje
ho konstruktor

- každý objekt obsahuje odkaz na VMT

- občejná metoda (early-binding)

↳ o jejich volání se rozhoduje při překladači

všechny
metody v Pythonu

- virtuální metoda (late-binding)

↳ o jejich volání se rozhoduje až při volání

↳ proto se udržují pointery ve VMT (protože není známo
jestli bude volána funkce přímka nebo potlouka)

statické členy a třídy

↳ vázано na samotný typ, ne instanci

↳ statické členy (metody, data) jsou alokovány pro
statickou třídu (ne pro instanci - nelze vytvořit)

můžou
být i v nestatické
třídě

↳ statická třída → má jen statické členy, nelze
vytvářet instance a nelze z ní dědit

→ implicitně sealed

Dekompozice

↳ rozklad na funkce, objekty, moduly

Funkční

↳ Top-down → co má která část dělat?

↳ části programu jsou rozděleny podle funkce a cíle

↳ Architektura je křehká na změny → je potřeba měnit hodně kódu

Objektová

↳ Bottom-Up → z čeho se musí systém skládat?

↳ autonomní objekty, co si vyměňují informace

↳ zapouzdření

↳ celkové věci změněm

Modulární dekompozice a metriky kvality

↳ rozklad na knihovny, jmenové prostory (namespace)

Kvalita

1) co nepřetř single responsibility

2) decoupling → nízká vazba mezi moduly

Diskrétní simulace

↳ spojitá simulace → smyčka a v každém kroku se zeptá, zda se něco změnilo

↳ diskrétní simulace → řízení událostmi, skáče od jedné k další

Architektura diskrétní simulace

1. Kalendar událostí

↳ kontejner s všemi naplánovanými událostmi

↳ používá se min. halda a vybírá se s nejnížším časem

2. Simulační jádro

```
while (!kalendar. Je Prázdný()) {
```

```
    Událost u = kalendar. Vyber Nejmenší();
```

```
    aktuální Cas = u. Cas;
```

```
    u. Zpracuj (tento System);
```

```
}
```

→ • změni stav systému
• naplánuje události

3. Hierarchie událostí

↳ Open / Closed Principle

↳ abstraktní třída událost s vlastností Cas a ^(abstraktní) metodou zpracuj

↳ od události dělí Príchod Zákazníku, Konec Obsluhy, ...

Objektová dekompozice - SOLID

S = Single responsibility principle

↳ jeden objekt by měl mít na starosti jednu věc

O = Open / Closed Principle

↖ jako ochrání

↳ třída by měla být otevřena na rozšiřování, ale uzavřena ve smyslu, že se nebude měnit rozhraní

L = Liskov Substitution Principle

↳ objekty by měly být možné nahradit stejnými potomky bez porušení funkčnosti programu

I = Interface Segregation Principle

↳ klient by neměl být nucen záviset na velkém rozhraní, co nepoužije

↳ Lepší je mít více menších rozhraní, co jsou úzce zaměřené

D = Dependency Inversion Principle

↳ konkrétnější musí záviset na abstraktnějším

↳ vyšší moduly by neměly záviset na nižších modulech

↑ Dependency Injection

↳ např. Eshop má použít PosliZpramu() přes INotifikator
Eshop nemá použít PosliMail() nebo PosliSMS() přes nějakou třídu, co si vytvoří přes new, ale mělo by to fungovat naopak.

abstrakce →
nesmí záviset
na detailech

Vlákna

- umožňuje provádět více instrukcí zároveň současně
- Environment. Processon Count
- problém se synchronizací

race - condition

↳ správnost výsledků závisí na časování a pořadí

Thread-safe

↳ např. Random ^{Next()} není bezpečná, nelze aby z ní četlo více vláken, potřeba použít - Shared()

zámk

↳ určitou část kódu může provádět jen jedno vlákno
↳ zamezení přispívání referenční objekt

deadlock

↳ pokud vlákno B a A drží zámek sobě navzájem a program zamrzne

Typy proměnných

hodnotové → int, byte, short, long, int ..., double, float,
→ bool
→ char
→ struct
→ enum, System.Value Tuple

referenční → class, List<T>, Random, ...

→ pole int[], string[]

→ string (System.String)

→ pomocí operátoru → delegáty a rozhraní System.Tuple

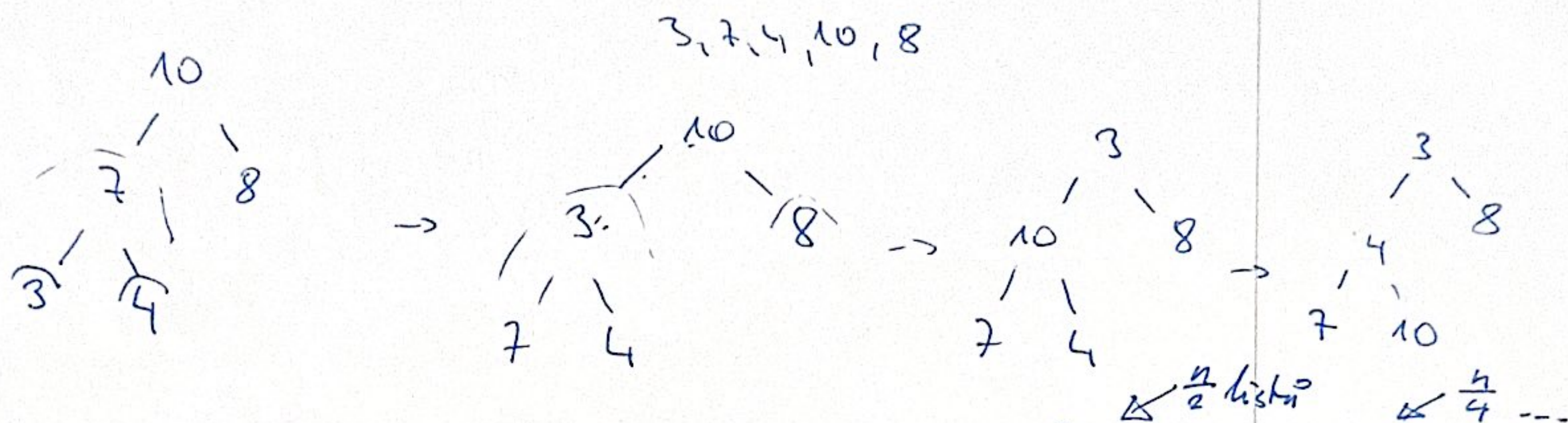
→ kopíruje se hodnoty

→ pouze odkaz na dynamicky alokovanou paměť

Halda v lineárním čase

- máme úplný binární strom z pole, pro listy je pochůška kudy minimálně splněna

- při postupujícím listu výš a



- $T(n) \rightarrow$ protože heapify (problém dle) končíme pro listy 0, 1, 10 další úroveň 1, tedy počet listů klesá exponenciálně, ale počet vrcholů stoupá lineárně

Optimální uspořádání matic

$$A_1: 10 \times 100$$

$$A_2: 100 \times 5$$

$$A_3: 5 \times 50$$

$a \times b$ a $b \times c$ vyžaduje $a \cdot b \cdot c$ operací

$$1.) (A_1 A_2) A_3$$

$$\rightarrow 10 \cdot 100 \cdot 5 = 5000 \quad (A_1 A_2) \quad 10 \times 5$$

$$10 \cdot 5 \cdot 50 = 2500 \quad (A_1 A_2) A_3$$

$$\left. \begin{array}{l} 5000 \\ 2500 \end{array} \right\} 5000 + 2500 = 7500$$

$$2.) A_1 (A_2 A_3)$$

$$\rightarrow A_2 A_3 = 100 \cdot 5 \cdot 50 = 25000$$

$$A_1 A_2 A_3 = 100 \cdot 50 \cdot 10 = 50000$$

$$\left. \begin{array}{l} 25000 \\ 50000 \end{array} \right\} 75000$$

$$2 \times 8$$

$$8 \times 4$$

$$4 \times 5$$

$$5 \times 2$$

menší množství

$$A_1(A_2 A_3)$$

$$(A_1 A_2) A_3$$

vyřeší se horší

$$(A_2 A_3) A_1$$

$$A_2(A_3 A_1)$$

tedy se vyřeší horší

$$\text{přes } (A_1 A_2)(A_3 A_4)$$

$$\text{přes } (A_1 A_2 A_3)(A_4)$$

$$(A_1 A_2 A_3 A_4)$$

Problém diamantu



- třída A definuje metodu, kterou
třída B a C přepíše, D dědí od
B a C. Kterou metodu má použít?

→
C# nepodporuje násobné dědění, aby se tomto
problému vyhnul

→ lze namísto mít více interface (kontraktů) → ten jen říká,
co musí daná třída implementovat

virtual

↳ výchozí implementace metody, lze přepsat pomocí override

↳ nemusí se přepsat

late-binding
~ VMT

abstract

↳ lze implementovat jen v abstraktní třídě

↳ vlastně nemá implementaci a každý potomek musí
použít override a přepsat ji

nelze vytvořit instanci
abstraktní třídy

Dědičnost

- is-a vztah

- class B: A → kdykoliv chci použít objekt typu A, tak můžu použít B, protože má jen vlastnosti navíc

- abstrakce se objekt se složky jak z A, tak z B

- při použití override se přepíše ukazatel v VMT ← tabulce virtuálních metod

- konstruktory

↳ potomek nemůže vytvořit objekt sám o sobě, ale musí předem zavolat konstruktor předka, pak mezitříd a nakonec potomka

↳ pomoci : base (parametry)

stinění (new) / přepsání (override)

↳ override (dynamická vazba)

↳ přírodní metoda v předku musí být abstract nebo virtual

↳ polymorfismus

↳ pokud zavolám metodu přes referenci na předka a objekt je typu potomek, tak se zavolá metoda potomka

↳ new (statická vazba)

↳ pokud zavolám metodu přes referenci předka, ale objekt je typu potomek, tak se zavolá metoda předka
↳ pokud přes referenci potomka, tak metoda potomka

List<Zvire>, v něm
Pes, Kočka, Ptak a
.DejHlas() zareaguje
vždy jinak

není
polymorfismus →