

1) Koncepty pro abstrakci, zapouzdření a polymorfismus

třída (class)

- = základní stavební jednotka OOP
- skupina objektů se společnými vlastnostmi a chováním
- { metody = funkce / procedury uvnitř třídy
datové položky (attributes / fields) = proměnné uvnitř třídy

rozhraní (interface)

- definuje „kontrakt“, který třídy mohou implementovat
- dosahuje deklarace metod bez jejich implementace
- třída nějakého rozhraní musí tyto metody implementovat

metody

- = funkce definované uvnitř třídy
- mohou pracovat s datovými položkami a provádět operace
 - instance methods → vyžadují vytvoření instance třídy
 - static methods → mohou volat na třídě bez vytvoření instance

datové položky

- = proměnné nebo konstanty definované uvnitř třídy

dědičnost

- = mechanismus, pomocí kterého může třída zdelebit vlastnosti a chování jiné třídy
 - { jednoduchá
víceúrovní → dědičim z třídy, která sama dědičí z třídy

viditelnost

- private → přístupné jen uvnitř třídy
- public → veřejné
- protected → přístupné pro třídu a její potomky
- (internal → přístupné uvnitř projektu)

polymorfismus

= schopnost různých objektů chovat se jinak při volání stejných metod

př. override tříd při dědičnosti

↳ dynamicky → overriding, řeším při běhu

staticky → přetěžování (stejný název, jiný # parametrů),
řeším při komplikaci

typování

• statické → typ určený při deklaraci, nemění se

• dynamické → type checking až při běhu,

do proměnné můžu vložit cokoliv

jednoduchá dědičnost

- umožňuje sdílet kod a funkčnost mezi třídami

- odvozená třída může použít / rozšířit / upravit

chování základní třídy

(virtual)

• virtuální metody → lze je přepsat pomocí override
→ nevirtuální metody přepsat nelze

vícenásobná dědičnost (v C++ nejde)

= dědím z více tříd najednou

- problem:  2 třídy, které dědí ze stejné třídy
↳ diamant

abstrakce = proces, při kterém se definují klíčové charakteristiky nějakého objektu, oddělující ho od méně podstatných detailů
→ třídy, rozhraní

zapouzdření = princip, který udržuje data a metody bezpečně před vnějším přístupem a zneužitím
→ private, protected

2) Primitivní a objektové typy a jejich reprezentace

číselné typy :

- celá čísla : int = 32-bit

long = 64-bit

short =

byte =

- desetinná čísla : float = 32-bit

double = 64-bit

decimal =

enumerace

- enum = sada pojmenovaných konstant

hodnotové typy

- ukládány na zařízení

- obsahují přímo hodnotu

- efektivnější pro malé a jednoduché datové struktury

- při předání funkci v argumentu jsou kopírovány

př. int, struct

referenční typy

- ukládány na haldě

- reference na paměťovou adresu, kde je uložena hodnota

- výhodné pro větší a složitější struktury

- předávají referenci, tedy funkce je mohou měnit / upravovat

př. class, string, array

• reference :

```
var obj = new MyClass() ← reference na instanci třídy  
var obj2 = obj  
← obj i obj2 ukazují na stejný objekt
```

imutabilní typy

= objekty, jejichž stav nelze po vytvoření změnit
př. String
→ při úpravě Stringu se vytvoří nový objekt

boxing

= proces převodu hodnotového typu na referenciální
př. int i = 123
object o = i

unboxing

= převod zpět na hodnotový typ
př. int j = (int) o

3) Generické typy a funkcionální průkry

generické typy

- umožňují vytvářet třídy / metody / rozhraní / delegáty,

které mohou operovat s různými datovými typy

př. `List < T >`

↳ definice pro generický typ `T`

- při vytváření instance potřeba specifikovat typ

`List < int > = new List < int >();`

delegáty

= typy, které bezpečně zapouzdřují metody

• něco jako jako uložení funkce do proměnné

• použití:

- události a callbacks

- definování metod, které mohou být předány

jako parametry / argumenty funkcí

př. Operation add = $(x, y) \Rightarrow x + y$;

`int res = add(3, 5)`

• můžou ukazovat na více metod

př. Action a = $() \Rightarrow \text{Console.WriteLine("Hello")}$;

`a += () \Rightarrow \text{Console.WriteLine("World")};`

`a(); // vypíše Hello in World`

• generické delegáty: `Func < >` → může vrátit hodnotu

`Action < >` → nic nevraci

• anonymní metody

→ inline definování delegátů

`Operation subtract = delegate (int x, int y) { return x - y; };`

→ zápis lze zkrátit lambda výrazy

lambda funkce

= anonymní metody používané k vytváření kompaktních způsobů implementace funkcí, které lze předat jako argumenty nebo použít v LINQ dotazech

↳ (parametry) \Rightarrow výraz
↳ (parametry) $\Rightarrow \{ \text{příkazy} \}$

Func<int, int, int> add = $(x, y) \Rightarrow x + y$;
 $\underbrace{\quad}_{\text{in}} \quad \downarrow \text{out}$

Action<string> greet = $(x) \Rightarrow \text{Console.WriteLine}(\text{"Hello "} + x)$;
 \uparrow
jenom in

var filtered = numbers.Where($n \Rightarrow n > 5$);
List.ForEach(item \Rightarrow Console.WriteLine(n));

\rightarrow stručnost a čitelnost kódu

\rightarrow možnost přístupování k proměnným v oboru,
ve kterém byly definovány

4) Manipulace se zdroji a mechanismy pro ošetření chyb

using

- používáno k automatickému spravování životního cyklu zdrojů
 - ↳ zejména pro objekty, které implementují rozhraní `IDisposable`
 - metoda `Dispose` = uvolnění zdrojů
 - volá se automaticky při ukončení bloku `using`
 - zabrání je úniku paměti apod.

v podstatě

```
try {}  
catch {}  
finally { Dispose(); }
```

ale čitelnější v kódu

- pomáhá předcházet chybám spojeným s manuálním uvolňováním zdrojů
- lze používat `using` pro více objektů zároveň

```
using (var file = new StreamWriter("file.txt"))  
using (var sql = new SqlConnection(connectionString)) {  
    // práce se souborem a databází }
```

try-catch

→ odchytávání výjimek

try {} ↘ může chybět

catch (Exception e) {}

↳ jakákoli výjimka

- výjimky lze specifikovat:

try {} ↗ specifická reakce na I/O výjimku

catch (IOException e) {}

catch (Exception e) {}

- propagace chyboucího stavu → vyvolání výjimky

if (condition Fails) {

 throw new InvalidOperationException ("Nesplněna podm.")

}

- vlastní výjimky

public class MyException : Exception {

 public MyException (string message) : base (message) {}

}

try {

 // Kód, který může vyvolat výjimku

}

catch {

 // Zpracování výjimky

}

finally {

 // Kód, který se vykona' vždy

}

5) Životní cyklus objektů a správa paměti

alokace paměti

◦ statická alokace

- alokace proměnných a objektů na statické paměti
- alokace/dealokace při načítání/ukončení programu

static int variable = 10;

◦ alokace na zásobníku

- automatická alokace a dealokace
- proměnné alokované při vstupu do bloku
a dealokované při jeho opuštění

◦ alokace na haldě

- dynamická alokace objektů
- objekty na haldě existují, dokud nejsou
odstraněny garbage collectorem

konstruktor

→ slouží k inicializaci objektů

→ automaticky volaný při vytváření nové instance třídy

s parametry : public MyClass (int id) {}

bezparametrické : public MyClass () {}

↳ pokud není explicitně definován žádoucí konstruktor, vytváří se automaticky

volání zděděných konstruktorů → : base (parametry)

inicializační blok → kód mimo konstruktor, který je součástí inicializace

např. inicializace u deklarace proměnných

destruktor

- volá se při odstranění objektu garbage collectorem
- slouží k uvolnění zdrojů

`~MyClass()`

finalizator

- často se používá jako synonymum k destruktoru
- metoda executed garbage collectorem, když mže objekt
- implicitně definovaný, nelze ho volat / vytvořit explicitně (naopak od destruktoru ho nelze dležit ani customizovat)

alternativa: rozhraní `IDisposable` + blok `using`
→ metoda `Dispose()`

- explicitní uvolňování zdrojů je v C# neobvykle
 - spravuje to garbage collectorALE pro neřízené zdroje (např. soubory) je vhodné používat `Dispose`, `using`

reference counting

- může ho využívat COM Interop pro správu COM objektů

garbage collector

- průběžně mže nedosažitelné objekty (0 referencí)
- jednou za čas mže smyčky mimo aktuální kód (např. 2 třídy co na sebe ukazují nauzájem)

6) Vlákna a podpora synchronizace

vlákna = základní jednotky umožňující synchronizaci

C#: třída Thread

předává se: delegát nebo metoda, která se má vykonať

→ obvykle void metody ale lze i přijímat parametry přes lambda výrazy nebo anonymní metody

Thread t = new Thread(new ThreadStart(MethodName))
popř. () => MethodWithParam("Hello"))

void ThreadFunction()

// kód, který vlákno vykonává

}

Thread t = new Thread(ThreadFunction);

t.Start() → spuštění

t.Join() → zablokuje volající vlákno dokud cílové nedokončí činnost

t.Interrupt() → přerušení vlákna

t.Abort() → ukončení

t.Priority → lze nastavit prioritu (např. ThreadPriority.Highest)

t.State → lze zjistit stav vlákna

race-conditions

→ více vláken přistupuje ke sdíleným datům ve stejný čas
a alespoň 1 z nich provádí zápis

→ vede k nesprávnému a nepředvídatelnému chování programu

```
int sharedResource;  
void Increment() { sharedResource++; }
```

lock:

```
object lockObj = new object();  
void SafeIncrement() {  
    lock (lockObject) {  
        sharedResource++;  
    }  
}
```

Monitor = lock s rozšířenými možnostmi

```
Monitor.Enter (lockObject);  
try { sharedResources++; }  
finally { Monitor.Exit (lockObject); }
```

Mutex, Semaphore

7) Implementace základních prvků objektových jazyků

tabulka virtuálních metod

→ obsahuje adresy přepsaných metod
pro dynamické užívání za běhu

viz. předchozí kapitoly + zkusební zadání

8) Nativní a interpretovaný běh, řízení překladu a sestavení programu

- kód v C# → mezikód Microsoft Intermediate Language (MSIL)
- MSIL = nezávislý na platformě
 - připravený pro JIT komplikaci
 - často označován bytecode
 - sada instrukcí, které rozumí Common Language Runtime (CLR) v .NET Frameworku

př. .method public hidebysig static void Main() cil managed

{

.entrypoint

// Code size 11 (0xb)

.maxstack 1

IL_0000: ldstr "Hello World"

IL_0005: call void [mscorlib] System.Console::Write(string)

IL_000a: ret

}

- CLR v .NET → interpret jazyka

→ virtuální stroj .NET Frameworku, který spravuje

běh C# programů

→ poskytuje Správu paměti a výjimek, bezpečnostní kontrola atd.

→ JIT komplikace

just-in-time překlad (JIT)

- překlada MSIL kód do strojového kódu až když je kód potřebný k výkonaři
- umožňuje optimalizace pro konkrétní hardware a aktuální runtime podmínky

ahead-of-time překlad (AOT)

- překlad kódu před spuštěním aplikace
- lepší celkový výkon i doba spuštění (není potřeba čekat)
ALE větší spustitelné soubory

proces sestavení programu

→ komplikace zdrojového kódu do mezijsyka
a následné generování spustitelného souboru

C#:

- komplikace do mezijsyka pomocí C# kompilátoru csc.exe
- mezijsyk je za běhu překládán JIT kompilátorem do nativního strojového kódu

Program.cs $\xrightarrow{\text{csc Program.cs}}$ Program.exe

oddělený překlad

= jednotlivé části zdrojového kódu lze kompilovat
nezávisle na sobě

→ používá se pro velké projekty rozdělené do více souborů
nebo modulů

C#:

- jednotlivé třídy → .exe nebo .dll (knihovna)

linkování

- = spojuje komplikované soubory do 1 spustitelného souboru
- v C++ se obvykle provádí automaticky

staticky linkované knihovny

- ve fázi linkování se dle linkera celá knihovna a zabuduje se napěno do našeho zdrojáku
- v C++ se nepoužívá (ale dle se to nejak naruší vlastnosti)

⊕ zabrání konfliktům verzí závislostí

jednoduchá distribuce (vše v 1 souboru)

⊖ větší spustitelný soubor

aktualizace knihovny vyžaduje rekomplikaci celého programu

dynamicky linkované knihovny

→ celá se přeloží předem (během spuštění nebo provozu)

→ zdrojový kód ji může za běhu využívat

- nepřímý odkaz, zavede se do paměti

C++: .dll soubory

Assembly Load ("Knihovna.dll") → načtení .dll za běhu

⊕ menší velikost spustitelných souborů

možnost aktualizace knihovny bez nutnosti rekomplikace aplikace

⊖ možné problémy s verzemi

potřeba správně spravovat načítání knihoven (např. cesty, verze)

běhové prostředí procesu

- = Soubor všech zdrojů a stavů, které proces potřebuje pro svůj běh

komponenty:

- paměťový prostor
 - obsahuje kód programu, data a zásobník
- procesorový čas
 - rozdělení CPU času mezi procesy
- systémové prostředky
 - soubory, síťové spojení, atd.

C#:

- .NET runtime spravuje paměť, garbage collection a bezpečnost
- přístup k systémovým prostředkům přes .NET API
(např. System.IO pro práci se soubory)

Vazba na operační systém

→ procesy interagují s OS přes systémová volání

C# : System.Diagnostics.Process → pro práci s procesy

Sandboxing = izolace kódu pro zabránění neautorizovaného přístupu k systémovým prostředkům