Objektově orientované programování

v C#

Petr Voborník

Tento materiál vznikl v rámci realizace projektu
Strategický rozvoj Univerzity Hradec Králové, reg. č. CZ.02.2.69/0.0/0.0/16_015/0002427

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

verze z 21.10.2022

Obsah

Seznam obrázků

Seznam Tabulek

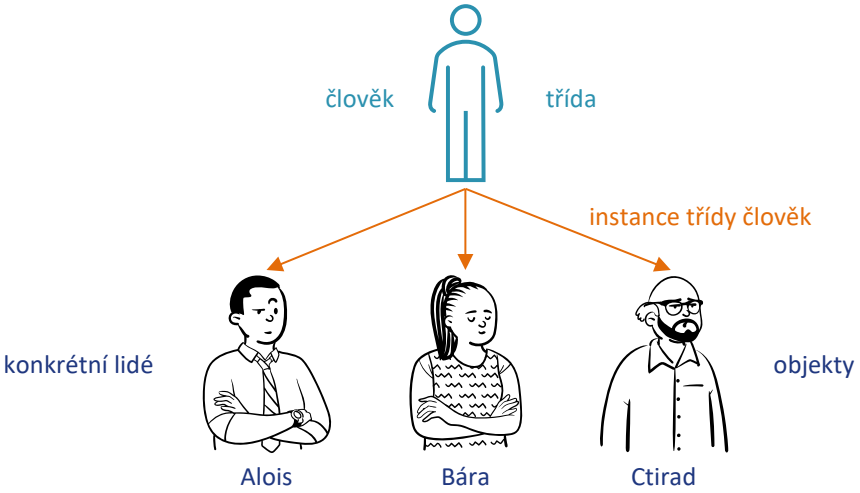
Seznam kódů

1 Třídy

1.1 Základní pojmy

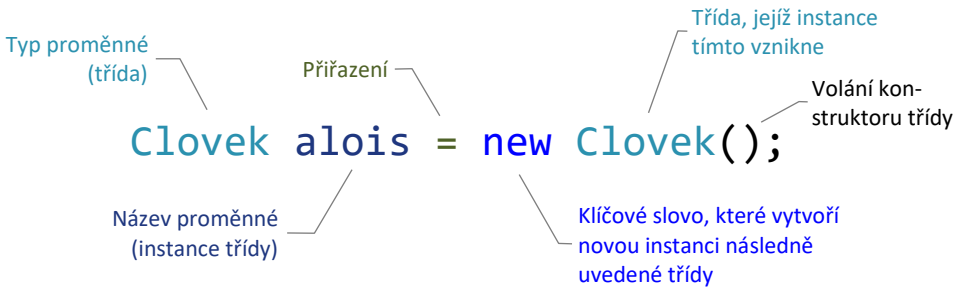
1.1.1 Třída, objekt, instance

Jedny z nejčastěji používaných pojmů jsou **třída**, **objekt** nebo též **instance**. Třída (*class*) je v podstatě šablona nějaké druhově podobné skupiny reálných objektů (*object*), která mapuje jejich vlastnosti (např. jméno, výška, věk, barva očí…), jimž však neurčuje konkrétní hodnoty. Jde tedy o obecný pojem, jako je např. člověk, auto, pes nebo třeba planeta. Pokud bychom jej však vztáhli ke konkrétnímu reálnému výskytu jednoho prvku, už by se jednalo o objekt (např. tento člověk, toto auto, váš pes nebo naše planeta). O takovýchto specifických objektech se také říká že jsou instancemi dané třídy (např. *objekt Alois je instancí třídy člověk*, viz Obr. 1). A tyto objekty pak dodávají konkrétní hodnoty vlastnostem, které třída jejíž jsou instancí definovala (např. jméno = Alois, výška = 170, věk = 20, barva očí = černá…).



Obr. 1 – Ilustrace vztahu mezi třídou a objekty (instancemi třídy)

V jazyce C# vytvoříme novou instanci třídy např. pomocí příkazu na Obr. 2.



Obr. 2 – Popis příkazu v C# pro vytvoření nové instance třídy

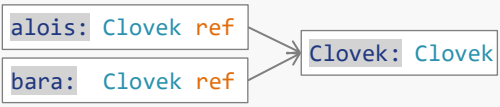
Tímto příkazem do proměnné **alois** typu **Clovek** tedy vytvoříme **instanci** třídy **Clovek**, čili nový objekt. Proměnná **alois** tak bude obsahovat **referenci** na tento objekt.

1.1.2 Reference

Reference (*reference*) znamená něco jako odkaz na jediný zdroj objektu v paměti. Je to jako mít adresu na to, kde Alois bydlí, tam kde ho vždy najdete a můžete k němu „přistupovat“ (např. číst nebo měnit hodnoty jeho vlastností). V proměnné však není uložen tento objekt samotný, jako je tomu u proměnných hodnotového typu (např. **int**, **double** nebo **bool**), ale pouze tato adresa (reference).

Pokud tedy přiřadíme proměnnou **alois1** do jiné proměnné **alois2**, zkopíruje se do ní pouze tato reference, nikoli objekt samotný (viz Kód 1).

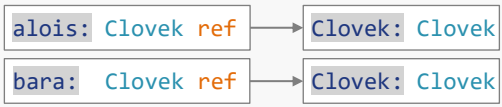
```
1: Clovek alois1 = new Clovek();
2: Clovek alois2 = alois1;
```



Kód 1 – Ukázka kódu a schématu dvou referencí na jeden objekt, instanci třídy *Clovek*

V případě, že bychom potřebovali dvě nezávislé instance třídy **Clovek**, je nezbytné každou z nich vytvořit zvlášť pomocí klíčového slova **new** (viz Kód 2).

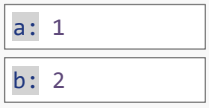
```
1: Clovek alois = new Clovek();
2: Clovek bara  = new Clovek();
```



Kód 2 – Ukázka kódu a schématu dvou referencí na dva různé objekty

Při předávání hodnot mezi proměnnými hodnotového typu (resp. struktury, viz kap. 2.1) tedy dochází ke kopírování této hodnoty a její následné úpravy se ve druhé proměnné nijak neprojeví (viz Kód 3).

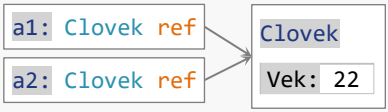
```
1: int a = 1;    // a: 1
2: int b = a;    // b: 1
3: b = b + 1;    // b: 2, a: 1
```



Kód 3 – Ukázka kódu a schématu vzájemné nezávislosti dvou proměnných hodnotového typu

Pokud ovšem změníme hodnoty u vlastností objektů v proměnných referenčního typu, bude tato změna pozorovatelná ze všech proměnných majících referenci na tento objekt (viz Kód 4).

```
1: Clovek a1 = new Clovek(); // a1.Vek: 20
2: Clovek a2 = a1;           // a2.Vek: 20
3:
4: a1.Vek = 21; // a1.Vek: 21, a2.Vek: 21
5: a2.Vek = 22; // a1.Vek: 22, a2.Vek: 22
6:
7: // Deklarace třídy Clovek
8: public class Clovek {
9:     public int Vek = 20; // Výchozí hodnota
10: }
```



Kód 4 – Ukázka kódu a schématu vzájemné závislosti dvou proměnných referenčního typu

Při porovnávání proměnných s referencemi na instance tříd nezáleží na tom, jaké jsou hodnoty jejich vnitřních proměnných, ale porovnává se pouze to, zdali tyto reference směřují na stejný objekt či nikoli (viz Kód 5).

```
1: Clovek a1 = new Clovek(); // 1. nová instance třídy Clovek
2: Clovek a2 = new Clovek(); // 2. nová instance třídy Clovek
3: Clovek a3 = a2;           // Druhá refernece na 2. instanci
4:
5: Console.WriteLine(a1 == a2); // False (reference na různé instance)
6: Console.WriteLine(a2 == a3); // True  (reference na tutéž instanci)
7:
8: public class Clovek {
9:     public int Vek = 20;
10: }
```

Kód 5 – Ukázka kódu porovnávajícího (==) proměnné s referencemi na různé (ř. 5) a na stejné (ř. 6) instance téže třídy

Zevnitř případného kódu třídy (např v metodě či vlastnosti) se aktuální instanci lze kdykoli dostat přes klíčové slovo **this**.

1.1.3 Dostupnost

Třídy, jejich složky (viz kap. 1.2) a další členy (viz kap. 2) mají vždy určenou svou dostupnost pro ostatní třídy, členy a projekty. Úrovně dostupnosti jsou následující:

- **public** (veřejné) – K těmto členům lze přistupovat z jakéhokoli jiného kódu bez omezení.
- **private** (soukromé) – K těmto členům lze přistupovat pouze z kódu téže třídy nebo struktury (tedy z kódu jejich metod či vlastností).
- **protected** (chráněné) – K těmto členům lze přistupovat pouze z kódu téže třídy nebo tříd z této třídy zděděných (viz kap. 3).
- **internal** (interní) – K těmto členům lze přistupovat z jakéhokoli jiného kódu, který se nachází ve stejném sestavení (assembly, projektu, součásti stejné kompilace).
- **protected internal** (chráněné nebo interní) – K těmto členům lze přistupovat z kódu téže třídy nebo tříd z této třídy zděděných, a zároveň lze přistupovat také z jakéhokoli jiného kódu, který se nachází ve stejném sestavení.
- **private protected** (soukromé chráněné) – K těmto členům lze přistupovat z kódu téže třídy nebo tříd z této třídy zděděných, ovšem pouze takových, které jsou zároveň součástí téhož sestavení.

Možné kombinace různých úrovní dostupnosti členů fiktivní třídy AT1 z kódu jejího (AT1), jiné třídy (AT2), třídy z ní zděděné (AT3) v témže sestavení (A), a zároveň nezávislé třídy BT1 a z ní zděné třídy (BT2) z jiného sestavení (B), ukazuje Tab. 1.

Tab. 1 – Přehled dostupností členů třídy AT1 z jiných tříd různého vztahu

	Sestavení A			Sestavení B	
Dostupnost členů třídy AT1	Třída **AT1**	Třída AT2	Třída AT3 : AT1	Třída BT1	Třída BT2 : AT1
`public`	✓	✓	✓	✓	✓
`private`	✓	✗	✗	✗	✗
`protected`	✓	✗	✓	✗	✓
`internal`	✓	✓	✓	✗	✗
`protected internal`	✓	✓	✓	✗	✓
`private protected`	✓	✗	✓	✗	✗

Dostupnost se zapisuje jako první údaj v deklaraci daného člena. Pokud není v této deklaraci dostupnost explicitně uvedena, bude dosazena dostupnost výchozí, která závisí na druhu tohoto člena. Zároveň ne všechny dostupnosti je možné přiřadit všem druhům členů. Možnosti použití různých kombinací dostupností a druhů členů ukazuje Tab. 2. Označeny v ní jsou i dostupnosti výchozí (☀) pro dané druhy členů.

Tab. 2 – Možné kombinace dostupností, jednotlivých druhů členů a jejich výchozí stav

Dostupnost	Třída, struktura, záznam	Vnořené třídy, struktury a záznamy	Složky tříd a struktur
`public`	✓	✓	✓
`private`	✗	✓ ☀	✓ ☀
`protected`	✗	✓	✓
`internal`	✓ ☀	✓	✓
`protected internal`	✗	✓	✓
`private protected`	✗	✓	✓

Kromě kombinací v Tab. 2 je zde možnost nastavit dostupnost kódů **get** a **set** u vlastností (viz kap. 1.2.2). Ty mají ve výchozím stavu stejnou dostup-

nost jako daná vlastnost, lze jim však nastavit i dostupnost nižší (např. u **public** vlastnosti nastavit **privat** pro **set** kód), přičemž to že bude tato dostupnost skutečně nižší (tzn. bude danou část omezovat více než vlastnost pod kterou patří), je přitom podmínkou pro validitu kódu.

1.2 Složky tříd

Třídy (ale i struktury a záznamy) neobsahují kód přímo v sobě, ale mohou obsahovat složky, které tyto „šablony pro objekty“ teprve definují. Základný druhy těchto složek jsou atributy, vlastnosti, metody a události.

1.2.1 Atributy (proměnné)

Atributy, nebo též datové položky či *fields*, jsou v podstatě proměnné, které třída definuje a s jejichž hodnotami pracuje v rámci svých dalších složek[1]. Tyto proměnné jsou tedy pro třídu jakoby globální, tzn. mohou je číst a zapisovat do nich všechny části kódu této třídy, bez jakéhokoli omezení. Možnost přistupovat k nim z kódu jiných tříd pak závisí na nastavení jejich dostupnosti (viz kap. 1.1.3).

A právě z důvodu neomezeného přístupu (jak pro čtení, tak i zápis, resp. změnu hodnoty) bez možnosti jeho další kontroly, bývají tyto vnitřní proměnné třídy většinou soukromé (**private**). Přístup k nim zvenčí (z jiných tříd) pak bývá realizován prostřednictvím vlastností a tzv. zapouzdření (viz kap. 1.2.2).

U soukromých atributů, především (ale nejen) u těch zapouzdřených, je zažitou konvencí používání malého prvního písmene (odpovídající vlastnost má pak tento znak velký; C# je totiž case-sensitive jazyk, tzn. rozlišuje malá

[1] Jako atributy v této knize budeme označovat právě tyto proměnné třídy (v AJ se nejčastěji používá *fileds*), nikoliv příznaky zapisované v hranatých závorkách nad deklaracemi členů, kteréžto se také nazývají atributy a spadají spíše do tématu reflexe.

a velká písmena jako rozdílné znaky). Druhým nejčastějším typem jejich pojmenovávání je pak přidání podtržítka před jejich název. Tedy např. **jmeno** nebo **_jmeno**.

Datový typ atributů musí být vždy striktně deklarován, nelze tedy použít zkratku **var**, a to ani v případě, že je součástí této deklarace přiřazení výchozí hodnoty (viz Kód 6).

```
1: public class Clovek
2: {
3:     private string jmeno; // Plná deklarace včetně dostupnosti
4:     string prijmeni;      // Bez uvedení dostupnosti => private
5:     int vek = 20;         // S nastavením výchozí hodnoty
6:     public static int Pocet = 0; // Veřejná statická proměnná
7: }
```

Kód 6 – Ukázka možných deklarací atributů

Každá instance třídy (objekt) pak má v paměti vyhrazený prostor pro všechny své atributy a nastavuje jim tak vlastní hodnoty. Výjimkou jsou pouze atributy statické.

Statické složky

Předně statické (*static*) mohou být nejen atributy, ale všechny složky třídy, a dokonce i třída sama (statická třída pak smí obsahovat pouze statické složky). Od klasických složek jsou ty statické rozlišeny klíčovým slovem **static** zapsaným hned za určením dostupnosti (popř. místo ní, je-li použita výchozí dostupnost, viz např. Kód 6, řádek 6).

Statické atributy jsou pak skutečnými globálními proměnnými, jelikož mají jen jednu hodnotu společnou pro všechny instance této třídy, v případě veřejného (**public**) atributu pak dokonce společnou pro celou aplikaci.

Vzhledem ke značné nekontrolovatelnosti přístupu k takové proměnné se však v klasické hodnotové formě příliš nepoužívají, a často bývají omezeny alespoň proti zápisu (např. číslo verze aplikace), anebo jsou referenčního typu (např. objekt s nastavením aplikace). Statická proměnná **Pocet** by tak například mohla sloužit k počítání vytvořených instancí třídy člověk (její hodnota by se navyšovala v konstruktoru této třídy, viz Kód 7), což se většinou používá spíše pro diagnostické účely.

```
1: public class Clovek // Třída
2: {
3:     public static int Pocet = 0; // Počet vytvořených objektů
4:     public Clovek() // Konstruktor třídy
5:     {
6:         Pocet++;    // Zvýšení hodnoty v proměnné Pocet o +1
7:     }
8: }
```

Kód 7 – Ukázka použití statické proměnné *Pocet* pro počítání instancí třídy *Clovek*

Jelikož jsou tyto atributy nezávislé na konkrétní instanci, pro přístup k nim stačí uvést název třídy (ten lze vynechat v kódu dané třídy, viz Kód 7, řádek 6) a název dané proměnné (např. viz Kód 8, řádek 5).

```
1: public static class JinaTrida             // Statická třída
2: {
3:     public static void ResetujPocetLidi() // Statická metoda
4:     {
5:         Clovek.Pocet = 0; // Nastavení hodnoty statické
                             // proměnné Pocet třídy Clovek
6:     }
7: }
```

Kód 8 – Ukázka přístupu k veřejné statické proměnné z jiné třídy

Konstanty

Konstanty jsou v podstatě statické atributy s přístupem pouze pro čtení. Jejich hodnotu jim lze přiřadit jedině v rámci jejich deklarace (viz např. Kód 9, který je reálnou součástí .NET). Klíčové slovo **static** je v deklaraci nahrazeno slovem **const** (viz Kód 9).

```
1: public static class Math
2: {
3:     public const double PI = 3.1415926535897931;
4:     public const double E  = 2.7182818284590451;
5:     // …
6: }
```

Kód 9 – Ukázka části kódu třídy *Math* (součást .NET) obsahující deklaraci konstant

Pro čtení hodnoty těchto konstant z jiných tříd se pak stejně jako u statických atributů uvádí název třídy, tečka, název konstanty (např. viz Kód 10).

```
1: public class Kruh
2: {
3:     public static double Obvod(double r)
4:     {
5:         return 2 * Math.PI * r;
6:     }
7: }
```

Kód 10 – Ukázka použití konstanty v metodě jiné třídy

Atributy pouze pro čtení

Atributy pouze pro čtení nejsou totéž, co konstanty. Jednak nejsou automaticky statické, takže každá instance třídy v nich může mít uloženy jiné hodnoty, a za druhé jejich hodnota nemusí být určena pouze v jejich deklaraci, ale lze ji nastavit také až v konstruktoru třídy (viz str. 33). Výjimkou

jsou atributy pouze pro čtení, které jsou statické, neboť jejich hodnota musí být určena stejně jako u konstant již při jejich deklaraci.

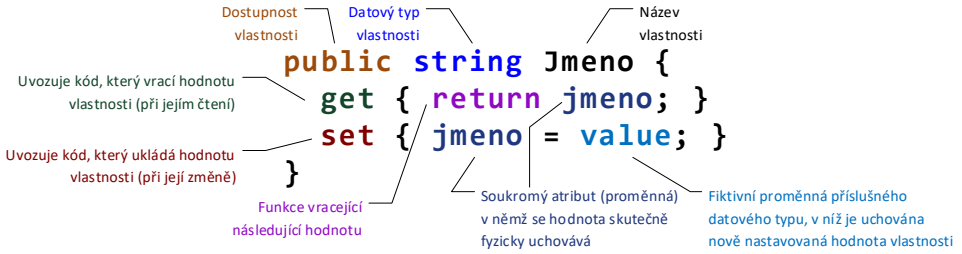
Aby byl atribut pouze pro čtení, stačí do jeho deklarace přidat klíčové slovo (modifikátor) **readonly** (RO), a to za nebo místo slova **static** (viz Kód 11).

```
 1: public class Clovek
 2: {
 3:     static readonly int maxVek = 150; // Statický RO atribut
 4:     public readonly int RokNarozeni;  // Veřejný RO atribut
 5:
 6:     public Clovek(int rokNarozeni)    // Konstruktor třídy
 7:     {
 8:         RokNarozeni = rokNarozeni;    // Nastavení RO atributu
 9:     }
10: }
```

Kód 11 – Ukázka deklarace a nastavení atributů pouze pro čtení

1.2.2 Vlastnosti

Vlastnosti (*properties*) navenek vypadají jako atributy, tzn. jejich hodnotu lze číst a měnit stejným způsobem jako u klasických proměnných. Uvnitř je však zásadní rozdíl v tom, že na pozadí jak jejich čtení, tak i zápisu hodnoty je kód, který daný proces obsluhuje. Kód pro čtení se nachází v bloku kódu **get** a část pro zápis v bloku kódu **set** (viz Obr. 3).



Obr. 3 – Popis kódu klasické zapouzdřující vlastnosti

Zapouzdření

Celý kód třídy tohoto **zapouzdření** (*encapsulation*) pak ukazuje Kód 12.

```
1: public class Clovek
2: {
3:     string jmeno;       // Atribut, v němž je hodnota uložena
4:     public string Jmeno // Vlastnost zapouzdřující atribut jmeno
5:     {
6:         get { return jmeno; }  // Blok kódu pro čtení
7:         set { jmeno = value; } // Blok kódu pro zápis
8:     }
9: }
```

Kód 12 – Kód třídy *Clovek* obsahující vlastnost *Jmeno*, která zapouzdřuje atribut *jmeno*

Automatické vlastnosti

Pokud vlastnost slouží k takovémuto prostému zapouzdření atributu, bez jakéhokoli dalšího obslužného kódu, lze namísto tohoto důkladného rozepsání použít i takzvanou automatickou vlastnost (*auto property*), zápis jejíž deklarace je značně zkrácen (viz Kód 13), při kompilaci je však přeloží do svého kompletního zápisu (viz Kód 12), včetně vytvoření soukromé proměnné (atributu) pro uchování hodnoty vlastnosti.

```
1: public class Clovek {
2:     public string Jmeno { get; set; } // Automatická vlastnost
3: }
```

Kód 13 – Ukázka zápisu automatické vlastnosti

Zkrácený zápis kódu zapouzdření

Mezi plným kódem zapouzdření a automatickou vlastností existuje ještě mezistupeň, který v případě že má **get** (kde zároveň nahrazuje i příkaz **return**) a/nebo **set** blok kódu pouze jediný příkaz umožňuje zápis zkrátit pomocí operátoru lambda (`=>`; viz Kód 14).

```
1: public class Clovek
2: {
3:     string jmeno;       // Atribut, v němž je hodnota uložena
4:     public string Jmeno // Vlastnost zapouzdřující atribut jmeno
5:     {
6:         get => jmeno;         // Vrátí hodnotu z atributu jmeno
7:         set => jmeno = value; // Uloží nastavovanou hodnotu
8:     }
9: }
```

Kód 14 – Zkrácený kód zapouzdřující vlastnosti pomocí lambda operátorů

Vlastnost s kódem

Pokud ale potřebujeme při čtení či zápisu provést nějaký výpočet či kontrolu, pak je nezbytné vlastnost rozepsat do kompletního kódu zapouzdření a příslušný kód tam zakomponovat (např. viz Kód 15).

```
 1: public class Clovek
 2: {
 3:     private string rodneCislo = "720316/1234";
 4:     public string DatumNarozeni
 5:     {
 6:         get
 7:         {
 8:             return                                  // 01234567890
 9:                 rodneCislo.Substring(4, 2) + "." + // 720316/1234
10:                 rodneCislo.Substring(2, 2) + "." + // 720316/1234
11:                 rodneCislo.Substring(0, 2);        // 720316/1234
12:         }                                          // "16.03.72"
13:     }
14: }
```

Kód 15 – Ukázka vlastnosti pouze pro čtení, jejíž hodnota není nikde uložena a při každém čtení se znovu vypočítává na základě hodnoty jiného atributu

Jak je z Kód 15 patrné, vlastnost nemusí vždy obsahovat oba bloky kódu, tedy **get** a **set**. Je-li tam pouze část **get**, pak je vlastnost pouze pro čtení, jelikož není k dispozici kód, který by nastavovanou hodnotu uložil. Ve vlastnosti však také může být pouze kód **set**, což by představovalo vlastnost pouze pro zápis, jejíž hodnotu by naopak nebylo možné přečíst, ale pouze ji nastavovat.

Omezení dostupnosti části kódu vlastnosti

Jednomu z těchto bloků lze také určit dostupnost, která čtení či zápis ještě více omezí pro ostatní třídy (např. klíčové slovo **private** před **set** učiní tuto vlastnost neměnitelnou pro její okolí a měnit hodnotu tak bude možné pouze zevnitř této třídy; např. viz Kód 16, kde je zápis zakázán členům z jiných sestavení, a navíc kontrolována validita nastavované hodnoty).

```
 1: public class Datum
 2: {
 3:     private int mesic = 1; // Zapouzdřovaný atribut
 4:     public int Mesic       // Zapouzdřující vlastnost
 5:     {
 6:         get => mesic;  // Zkrácený zápis kódu pro čtení
 7:         internal set   // Zápis jen pro členy téhož sestavení
 8:         {
 9:             if (value >= 1 && value <= 12) // Validace hodnoty
10:                 mesic = value;    // Podmíněné uložení hodnoty
11:         }
12:     }
13: }
```

Kód 16 – Ukázka vlastnosti, která je pouze pro čtení členům mimo aktuální sestavní, a i u nich je zápis kontrolován tak, aby měsíc mohl obsahovat pouze čísla 1-12

Místo klíčového slova **set** lze také použít klíčové slovo **init**, které znemožní změny vlastnosti odvšad, i ze třídy samotné, výjimkou je/jsou pouze její

konstruktor(y) , nebo nastavení v rámci vytváření nové instance (viz násl. kap.).

Nastavování hodnot v rámci vytváření nové instance

Při vytváření nové instance třídy lze nastavit hodnoty vlastností (a atributů, které nejsou pouze pro čtení) v rámci tohoto jediného příkazu tak, že se vypíší do složených závorek hned za příkazem pro vytvoření nového objektu a přiřadí se jim jejich hodnoty. Tento způsob hromadného nastavení jejich hodnot by měl být efektivnější než jejich následné přiřazování jednotlivými příkazy (viz Kód 17, ř. 1–5).

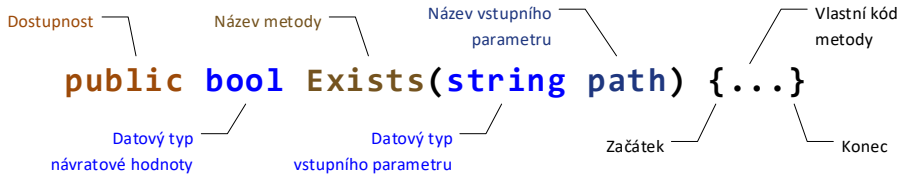
```
 1: var alois = new Clovek() {
 2:     Jmeno = "Alois",
 3:     Prijmeni = "Vomáčka",
 4:     Vek = 23,   // Čárka může být i za posledním prvkem
 5: };
 6:
 7: public class Clovek
 8: {
 9:     public string Jmeno { get; init; }
10:     public string Prijmeni { get; set; }
11:     public int Vek;   // Atribut
12: }
```

Kód 17 – Nastavení hodnot vlastností jako součást příkazu vytvoření nové instance

Zatímco tímto způsobem lze nastavovat i hodnoty vlastností, které mají místo **set** části jen **init** (viz Kód 17, ř. 9, nastaven na ř. 2), a lze je jinak nastavovat pouze v konstruktoru třídy, tak atributy, které jsou pouze pro čtení (mají modifikátor **readonly**, viz str. 18) takto nastavovat nelze a jedinou možností, kde se dají nastavit tak zůstává konstruktor třídy nebo výchozí hodnota atributu.

1.2.3 Metody

Metody (*methods*) jsou složky třídy obsahující ucelené části kódu, který vykonává určitou činnost. Ty sice mohly mít i vlastnosti, nicméně spouštět funkční kód čtením či změnou hodnoty není zrovna nejvhodnější. Metody pak krom návratové hodnoty mohou mít také vstupní parametry. Hlavička jedné takové metody je popsána na Obr. 4.



Obr. 4 – Popis částí hlavičky metody

Návratové hodnoty

Povinnou součástí hlavičky metody je tedy určení datového typu hodnoty, kterou metoda bude vracet. Jakmile je tento typ určen, metoda hodnotu tohoto typu vracet musí, a to ve všech svých možných větvích (např. v obou alternativních částech podmínky **if**). Pokud má však metoda pouze výkonný charakter a není zapotřebí, aby nějakou hodnotu vracela, lze v její hlavičce místo datového typu návratové hodnoty uvést klíčové slovo **void**, jež není datovým typem, ale signálem toho, že daná metoda nic vracet nebude[2].

Vrácení výstupní hodnoty zajišťuje příkaz **return**, který jednak vrátí hodnotu (popř. výsledek výrazu) zapsanou za ním, a za druhé, jelikož už metoda svůj účel splnila, tak okamžitě ukončí její běh a veškerý případný následující kód metody bude přeskočen (viz Kód 19).

[2] V některých programovacích jazycích (např. Pascal) se metody nazývají funkce (ty vždy vracejí výstupní hodnotu) a procedury (ty naopak nic nevracejí). V C# jsou to vždy pouze metody, o jejichž návratové politice rozhoduje **void** (metoda nic nevrací) nebo cokoli jiného (vrací).

```
1: public string PorovnejHodnoty(int a, int b) // Metoda vrací text
2: {
3:     if (a > b)
4:         return "A je větší než B"; // Vrátí text a ukončí metodu
5:     if (a < b)
6:         return "B je větší než A"; // Vrátí text a ukončí metodu
7:     return "Hodnoty jsou stejné"; // Nastane jen při (a == b)
8: }
```

Kód 18 – Ukázka metody vracející výstupní hodnotu a využívající return pro své předčasné ukončení

Jelikož se tato možnost předčasného ukončení často hodí i u metod typu **void** (tj. těch, které nic nevrací, např. v určité větvi programu, třeba nejsou-li platné všechny vstupní parametry), lze i zde použít příkaz **return**, ovšem pouze ten samotný, bez následného určení návratové hodnoty (viz Kód 19).

```
 1: public void VypisPorovnani(int a, int b) // Metoda nic nevrací
 2: {
 3:     if (a > b)
 4:     {
 5:         Console.WriteLine("A je větší než B"); // Vypíše text
 6:         return;   // Ukončí běh metody
 7:     }
 8:     if (b > a)
 9:     {
10:         Console.WriteLine("B je větší než A"); // Vypíše text
11:         return;   // Ukončí běh metody
12:     }
13:     Console.WriteLine("Hodnoty jsou stejné"); // Jen při (a==b)
14: }
```

Kód 19 – Ukázka metody, která nic nevrací (*void*) a přesto používá výhod předčasného ukončení svého běhu pomocí příkazu *return*

Zkrácení zápisu jedno-příkazové metody

Podobně jako se dalo u **get** či **set** kódů vlastností zkrátit zápis pomocí lambda operátoru (viz Kód 14 na str. 21), lze totéž provést i u metod, pakliže je jejich vnitřní kód tvořen jediným, byť vícenásobným, příkazem (neplatí pro podmínky a cykly; viz Kód 20).

```
1: public int VypoctiTretiMocninu(int x)   // Metoda vrací číslo
2:     => x*x*x;          // Operátor ukazuje na výstupní hodnotu
3:
4: public void VypisTretiMocninu(int x)    // Metoda nic nevrací
5:     => Console.WriteLine(x*x*x);   // Operátor ukazuje na příkaz
```

Kód 20 – Ukázka zkráceného zápisu metod pomocí lambda operátoru

Vícenásobná návratová hodnota

Pokud potřebujeme, aby metoda vracela více než jen jednu hodnotu, a z nějakého důvodu nechceme použít výstupní parametry (viz str. 29), ani výstupní hodnotu referenčního typu (třídu, strukturu či záznam), můžeme použít ještě tzv. typ řazené kolekce členů (*tuple type*).

V takovém případě se jako typ návratové hodnoty zapíše do kulatých závorek čárkou oddělený seznam klasických datových typů, přičemž jejich pořadí je pak závazné. Při vracení hodnoty příkazem **return** se následně uvede sada hodnot daných datových typů, opět zapsaná do kulatých závorek. Při volání takové metody je pak pro proměnou uchovávající výsledek tohoto volání nejvhodnější použít anonymní datový typ **var**. Tento výsledek se tváří jako objekt s vlastnostmi pojmenovanými **Item*X***, kde ***X*** určuje pořadí hodnoty v kulatých závorkách (viz Kód 21).

```
 1: (bool, int, double) Vydel(int a, int b)
 2: {
 3:     if (b == 0)
 4:         return (false, int.MinValue, double.NaN);
 5:     return (true, a / b, a / (double)b);
 6: }
 7:
 8: var v = Vydel(10, 3);
 9: if (v.Item1)
10:     Console.WriteLine($"10/3: div={v.Item2}, přesně={v.Item3}");
```

Kód 21 – Ukázka metody vracející trojici výstupních hodnot formou typu řazené kolekce členů a jejího použití

Pro přehlednější následnou práci s výsledkem je také možné si jednotlivé hodnoty pojmenovat a při následném čtení k nim pak přistupovat pod těmito jejich názvy.

```
 1: (bool OK, int Div, double Presne) Vydel(int a, int b)
 2: {
 3:     if (b == 0)
 4:         return (false, int.MinValue, double.NaN);
 5:     return (true, a / b, a / (double)b);
 6: }
 7:
 8: var v = Vydel(10, 3);
 9: if (v.OK)
10:     Console.WriteLine($"10/3: div={v.Div}, přesně={v.Presne}");
```

Kód 22 – Ukázka metody vracející trojici výstupních hodnot formou typu řazené kolekce pojmenovaných členů a jejího použití

Pro obě verze metody **Vydel** (Kód 21 a Kód 22) lze její výstupní hodnoty načíst i následující způsobem (viz Kód 23), tedy do lokálních proměnných uvedených ve správném pořadí v kulatých závorkách (jejich deklarace může

proběhnout i před tímto jejich použitím, a jejich názvy nemusí odpovídat druhé verzi metody).

```
(bool ok, int div, double presne) = Vydel(10, 3);
```

Kód 23 – Ukázka načtení vícenásobné výstupní hodnoty metody *Vydel* (Kód 21 a Kód 22) do individuálních lokálních proměnných

Parametry metod

Metody mohou mít vstupní parametry. Ty se píšou do kulatých závorek za název metody, přičemž tyto závorky je tam nezbytné uvést i v případě, že metoda žádné vstupní parametry nemá. Parametrů tak lze určit libovolné množství a to tak, že nejprve je u každého z nich je nejprve určen jeho datový typ a následuje název parametru, pod kterým bude v těle metodě dostupný jako lokální proměnná, jejíž případné změny ovšem neovlivní hodnotu případné proměnné, byla-li vstupní hodnota metodě předána jejím prostřednictvím (viz Kód 24).

```
1: void DvakratDva()
2: {
3:     int a = 2;
4:     int b = Mocnina(a);     // Hodnota v a se nezmění
5:     Console.WriteLine(a);   // Vypíše "2"
6:     Console.WriteLine(b);   // Vypíše "4"
7: }
8:
9: int Mocnina(int x)
10: {
11:     x = x * x;   // Změna hodnoty ve vstupní proměnné
12:     return x;
13: }
```

Kód 24 – Ukázka volání metody, která ač mění hodnotu vstupní proměnné, tak navenek se to neprojeví

Parametry předávané referenčně

Pokud bychom naopak chtěli, aby se případná změna ve vstupním parametru (**x**) promítla i ven z této metody do zadávající proměnné (**a**), pak je třeba do hlavičky metody před určení dané proměnné připsat klíčové slovo **ref** (viz Kód 25, řádek 8), které stanoví, že hodnota bude předána referenčně a veškeré změny této proměnné uvnitř metody se projeví i v proměnné zadávající tuto hodnotu. Zároveň pak musí být povinně klíčovým slovem **ref** označena i tato proměnná předávající parametru metody výchozí hodnotu (viz Kód 25, řádek 4), přičemž musí jít vždy o právě jednu proměnnou daného typu, nikoli např. o konstantu či matematický výraz, které mohou být pouze u nereferenčních parametrů.

```
1: void DvakratDva()
2: {
3:     int a = 2;
4:     Mocnina(ref a);          // Hodnota v a se může změnit
5:     Console.WriteLine(a);    // Vypíše "4"
6: }
7:
8: void Mocnina(ref int x)    // Vstupní parametr x zadán referencí
9:     => x = x * x;          // Změna hodnoty ve vstupní proměnné
```

Kód 25 – Ukázka volání metody se vstupním parametrem předaným formou reference

Hodnoty pro referenční parametry musí tedy být předávány prostřednictvím proměnné, přičemž tato proměnná musí mít předem definovanou hodnotu, a k její změně může, ale nemusí dojít.

Výstupní parametry

Alternativou k parametrům předávaným referenčně jsou takzvané výstupní parametry (místo **ref** se používá klíčové slovo **out**; viz Kód 26).

```
 1: bool Deleni(int delenec, int delitel, out double podil)
 2: {
 3:     if (delitel == 0)
 4:     {
 5:         podil = double.NaN; // Musí být nastaveno před return
 6:         return false;  // Po return už kód metody nepokračuje
 7:     }
 8:     podil = delenec / (double)delitel;
 9:     return true;
10: }
```

Kód 26 – Ukázka metody s výstupní parametr (*podil*), která před dělím kontroluje možnou chybu dělení nulou, a přes výstupní hodnotu oznamuje validitu tohoto výsledku

Výstupní parametry v proměnné, jež jim zprostředkovává hodnotu, před voláním metody být definovány nemusí, a po jejím volání již definovány zaručeně budou (viz Kód 27, řádek 5–6). Jejich deklaraci lze dokonce spojit přímo s jejich použitím při volání metody (viz Kód 27, řádek 3–4[3]).

```
 1: void VydelHodnoty()
 2: {
 3:     if (!int.TryParse(Console.ReadLine(), out int a)) return;
 4:     if (!int.TryParse(Console.ReadLine(), out int b)) return;
 5:     double c;
 6:     if (Deleni(a, b, out c))
 7:         Console.WriteLine(c);
 8:     else
 9:         Console.WriteLine("Hodnoty nelze vydělit");
10: }
```

Kód 27 – Ukázka volání metody z Kód 26 a dále zkoumající validitu tohoto výsledku

[3] Metoda **TryParse** struktury **Int32** (s aliasem **int**) překládá zadaný textový řetězec (první vstupní parametr metody) na číslo, přičemž toto číslo vrací přes druhý tentokrát výstupní parametr, ovšem o platný výsledek se jedná pouze v případě, že návratová hodnota této metody je **true** (jinak se text nepodařilo na číslo převést), podobně jako ukazuje Kód 26.

Pokud by nás hodnota výstupního parametru nezajímala, nemusíme ji zbytečně přijímat do proměnné, pročež stačí místo ní uvést pouze znak podtržítka (viz Kód 28).

```
1: if (Deleni(a, b, out _))
2:     Console.WriteLine("Hodnoty lze vydělit");
3: else
4:     Console.WriteLine("Hodnoty nelze vydělit");
```

Kód 28 – Ukázka kódu volající metodu (viz Kód 26) s výstupním parametrem, jehož hodnotu nepotřebujeme znát

Nepovinné parametry

Hodnoty některých parametrů není nezbytné znát vždy, popř. mají většinou stejnou hodnotu, kterou je potřeba změnit jen vyjímečně. Abychom mohli zkrátit zápis při jejich volání a nemuseli neustále dokola vyplňovat tyto téměř vždy stejné hodnoty, lze takové parametry definovat jako nepovinné a tuto nejčastější hodnotu jim přednastavit.

U nepovinných parametrů stačí v hlavičce metody přidat jejich výchozí hodnotu (např. `int x = 0`) a následně je již nebude nezbytné při volání takové metody vyplňovat, ani jakkoli zmiňovat. A jelikož mohou být z volání metody zcela vynechány, musí se takovéto parametry nacházet až za těmi povinnými (viz Kód 29). Pokud chceme zadat některý z pozdějších parametrů, ale ty dřívější přeskočit, pak je nezbytné u těch pozdějších přidat před hodnotu jejich název s dvojtečkou (viz Kód 29, řádek 7).

```
1: void Vypis(string text, string pred = "", string za = "")
2:   => Console.WriteLine("{1}{0}{2}", text, pred, za);
3:
4: Vypis("Ahoj");                          // "Ahoj"
5: Vypis("Ahoj", "Říkám: ");               // "Říkám: Ahoj"
6: Vypis("Ahoj", "Říkám: ", " pardále"); // "Říkám: Ahoj pardále"
7: Vypis("Ahoj", za: " pardále");          // "Ahoj pardále"
```

Kód 29 – Ukázka metody s nepovinnými parametry a varianty jejího použití

Vícenásobné parametry

Pokud potřebujeme metodě předat sadu hodnot stejného typu, ale ne přesně předem specifikovaného počtu, lze tak učinit buď pomocí parametru typu pole či seznam, nebo prostřednictvím vícenásobného parametru. Ten může být maximálně jeden, musí být uveden až jako poslední vstupní parametr metody, musí být jednorozměrným polem libovolného datového typu, a je třeba jej označit klíčovým slovem **params** uvedeným na začátku jeho deklarace (viz Kód 30, řádek 1).

Tomuto parametru je pak možné předat hodnoty dvojím způsobem. Tím pohodlnějším je vypsat je za ostatní parametry metody tak, jako kdyby se jednalo o klasické parametry (viz Kód 30, řádek 4), druhou možností je předání jako jedné hodnoty, a to stejného typu jakým tento parametr je (1D pole, viz Kód 30, řádek 5).

```
1: void Vypis(string pred = "", string oddel = ",", params int[] data)
2:     => Console.WriteLine("{0}{1}", pred, String.Join(oddel, data));
3:
4: Vypis("Fb: ", ", ",  1, 1, 2, 3, 5, 8);    // "Fb: 1, 1, 2, 3, 5, 8"
5: Vypis(data: new [] { 1, 1, 2, 3, 5, 8 });  // "1,1,2,3,5,8"
```

Kód 30 – Ukázka metody vícenásobným parametrem a možnosti jejího použití

Přetěžování metod a zkracování zápisu

Přetěžování metod (*overload*) znamená, že existuje více než jedna metoda se stejným názvem. Aby i přesto bylo vždy jednoznačně určeno, kterou z takovýchto přetížených metod voláme, musí se lišit alespoň v počtu či datových typech svých parametrů, a to tak, aby při jejím volání nemohlo dojít k jejich záměně (viz Kód 34).

```
1:  public class Kruh
2:  {
3:      // Metoda vykreslí kruh dle souřadnic středu x,y a poloměru r
4:      public void NakresliKruh(int x, int y, int r) { … }
5:
6:      // Vykreslení kruhu do prostoru obdélníku vymezeného
        // souřadnicemi levého hodního a pravého dolního rohu
7:      public void NakresliKruh(int x1, int y1, int x2, int y2)
8:          => NakresliKruh((x1 + x2) / 2, (y1 + y2) / 2,
9:                          Math.Min(Math.Abs(x1 - x2) / 2,
10:                                   Math.Abs(y1 - y2) / 2));
11: }
```

Kód 31 – Ukázka přetížení metody a volání první z nich tou druhou

Stejnojmenné metody by měly vykonávat totéž, různými způsoby (na základě různých vstupních parametrů). Visual Studio pak v „našeptávači“ (*IntelliSense*) nabízí tyto metody pouze jako jedinou položku, a až při zadávání jejich parametrů umožňuje listovat mezi jejími jednotlivými přetíženími. Často tak skutečný funkční kód mívá pouze jedna z těchto přetěžovaných metod a ty ostatní pouze volají ji (nebo sebe navzájem) s postupně upravovanými (dopočítávanými či prázdnými) parametry (viz Kód 34, řádek 7–10, kde druhá metoda volá tu první, přičemž přepočítává parametry tak, aby jí vyhovovaly).

Konstruktor

Konstruktor je speciální metoda či metody (třída může mít více přetížených konstruktorů), která je spuštěna maximálně jednou, a to při vytváření nové instance třídy, kterou vrací jako výstupní hodnotu.

Název metody konstruktoru musí být stejný, jako je název třídy. V hlavičce konstruktoru se neuvádí datový typ návratové hodnoty (ani **void**), protože jím bude právě instance této třídy (viz Kód 32).

```
 1: public class Clovek
 2: {
 3:     public string Jmeno { get; set; }
 4:
 5:     public Clovek() {              // Bezparametrický konstruktor
 6:         Jmeno = "Nový člověk";
 7:     }
 8:     public Clovek(string jmeno) {  // Konstruktor s parametrem
 9:         Jmeno = jmeno;
10:     }
11: }
```

Kód 32 – Ukázka bezparametrického konstruktoru (ř. 5–7) a jeho přetížení (ř. 8–10)

Pokud třída nemá definovaný žádný konstruktor, je jí automaticky přidán výchozí bezparametrický konstruktor neobsahující žádný kód, aby vůbec bylo možné vytvářet její instance. Jakmile však nějaký konstruktor třídě vytvoříme, tento výchozí se už nepřidává a používat tak lze pouze konstruktory námi definované. Konstruktory se na rozdíl od metod navzájem volat nemohou, neboť objekt vniká jejich voláním a takto by byl vytvořen vlastně dvakrát.

Chceme-li konstruktor, a tedy i vytváření instancí třídy prostřednictvím klíčového slova **new** zakázat úplně, můžeme konstruktor nastavit jako soukromý a vytváření instancí tak buď zcela zakázat, nebo jej řešit třeba přes statickou metodu třídy (např. **Create**, viz Kód 33), která konstruktor, přestože je soukromý, může používat i tak.

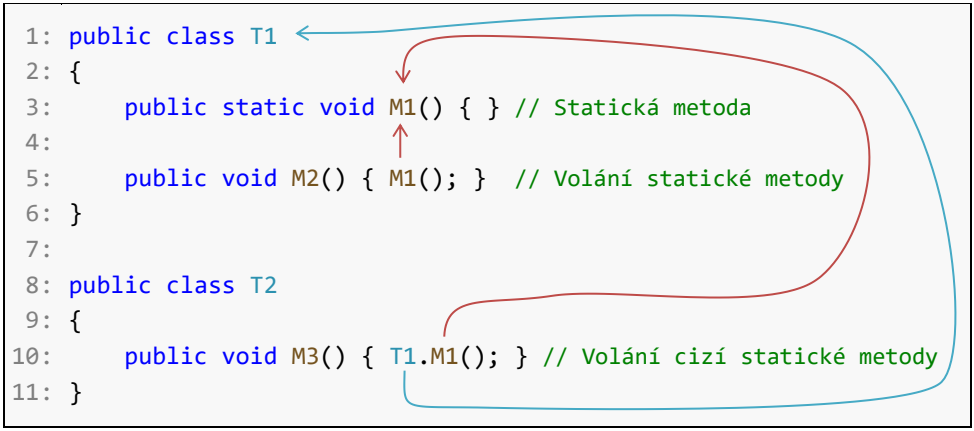
```
1: Clovek clovek = Clovek.Create(); // Vytvoření nové instance třídy
2:
3: public class Clovek
4: {
5:     private Clovek() { } // Soukromý konstruktor nelze zvenčí volat
6:
7:     public static Clovek Create()
8:         => new Clovek(); // Uvnitř třídy lze volat i soukromý kons.
9: }
```

Kód 33 – Ukázka „zapouzdření“ soukromého konstruktoru třídy do veřejné metody

Instanční a statické metody

Jak již bylo uvedeno v podkapitole Statické složky (str. 16), tak i metody mohou být statické a fungovat nezávisle na konkrétní instanci třídy. Co do deklarace, stačí na začátek jejich hlavičky (za nebo místo dostupnosti) přidat klíčové slovo **static** (viz Kód 34).

Takové metody pak ovšem nemají přístup k nestatickým členům třídy (ty naopak počítají se svým fungováním v rámci konkrétní instance a čerpají hodnoty z jejích atributů). Volání statické metody zevnitř třídy pak probíhá normálně pod jejím názvem (viz Kód 34, řádek 5), volání odjinud pak již není přes proměnou s instancí třídy, ale pod názvem této třídy a názvem příslušné metody (viz Kód 34, řádek 10).



Kód 34 – Ukázka volání statické metody

Metody rozšíření

Metody rozšíření (*extension methods*) jsou pevně svázány s určitým datovým typem (včetně tříd, struktur atd.), pod jejichž proměnnými se nabízejí a volají, jako kdyby byly součástí kódu tohoto typu, třídy či jiného druhu člena.

Pokud bychom tak měli klasickou metodu pro vypsání např. jednorozměrného pole `Vypis(int[] pole)`, pak bychom ji použili jako `Vypis(pole)`, avšak jednalo-li by se o metodu rozšíření, pak by byla volána jako metoda proměnné pole, tedy `pole.Vypis()`.

Tento přístup tak může být v některých případech intuitivnější a usnadňující při následném používání těchto metod, neboť jsou našeptávačem nabízeny pouze u proměnných takového typu, pro který jsou určeny. Zároveň kvůli jejich přidání není nutné zasahovat nebo rozšiřovat kód stávajících datových typů, ale kompletní kód těchto metod se zapisuje zcela zvlášť do jiné třídy. Metody rozšíření se dokonce celkem často přidávají do projektu zcela

externě, třeba v rámci NuGet[4] balíčků, což je třeba i základní princip .NET 6 (a výše) technologií, kdy k základnímu jádru .NET přidáváte pouze skutečně potřebnou funkcionalitu právě prostřednictvím balíčků NuGet.

Metoda rozšíření musí mít použitelnou dostupnost (**public** či **internal**), musí být statická a zapsána v dostupné statické třídě. Její první vstupní parametr musí být datového typu, pro který je tato metoda rozšíření určena (pro jehož proměnné či instance se bude tato metoda nabízet a používat). Tento první parametr pak musí být ještě označen klíčovým slovem **this** (viz Kód 35, ř. 3).

```
 1: public static class Rozsireni
 2: {
 3:     public static void Vypis(this int[] pole) // Metoda rozšíření
 4:         => Console.WriteLine(String.Join(", ", pole));
 5: }
 6:
 7: internal class Pole
 8: {
 9:     public static void VytvorAVypisPole(int delka)
10:     {
11:         int[] a = new int[delka]; // Deklarace 1D pole čísel
12:         for (int i = 0; i < a.Length; i++) // Naplnění náhodnými
13:             a[i] = Random.Shared.Next(10); // celými čísly 0-9
14:         a.Vypis(); // Vypsání pole vlastní metodou rozšíření
15:     }
16: }
```

musí být stejného typu

Kód 35 – Ukázka zápisu a použití metody rozšíření

[4] NuGet balíčky jsou knihovny s rozšiřujícím kódem, přídavnými funkcemi či celými komponentami, které můžete snadno přidat do svých projektů a rovnou je používat, např. z jejich hlavního úložiště na www.nuget.org, ideálně přes správce těchto balíčků vestavěného přímo ve Visual Studiu.

Delegát

Delegát (*delegate*) je typ určený pro specifikaci předpisu metody. Stanovuje, jaké musí mít metoda vstupní parametry (jejich počet a typy) a jaký bude typ její návratové hodnoty (viz str. 24).

Delegát může být deklarován zcela samostatně, jako např. třída, jen v rámci daného jmenného prostoru (viz kap. 2.4), popř. v rámci třídy jako její vnořený člen (viz kap. 1.2.5). Nejprve se uvádí jeho dostupnost (není-li uvedena, bude **internal** viz kap. 1.1.3), následuje klíčové slovo **delegate**, a pak již předpis (hlavička) samotné metody, tj. typ návratové hodnoty, název delegátu, a v kulatých závorkách seznam vstupních parametrů. Jelikož se jedná pouze o předpis metody, následuje již pouze středník (viz Kód 36).

```
1: public delegate void EventHandler(object sender, EventArgs e);
2: public delegate void Kliknuti(object sender, int x, int y);
3: public delegate bool PovolitZavreniOkna(Window okno);
4: public delegate double Vypocet(double a, double b);
```

Kód 36 – Ukázky deklarace delegátů

Kód 36 obsahuje několik ukázek deklarace delegátů. První z nich (ř. 1) ukazuje předpis systémového delegátu **EventHandler**, používaného v předchozích ukázkách. Druhý delegát (ř. 2) by bylo možné použít např. pro událost oznamující kliknutí myši na nějaký objekt, kdy krom reference na tento objekt (**sender**) předává také souřadnice (**x**, **y**), na které bylo kliknuto. Třetí delegát (ř. 3) může při pokusu o uzavření okna desktopové aplikace ověřovat, zda může toto uzavření proběhnout či nikoli. Předává referenci na objekt zavíraného okna, tentokrát nikoli obecně jako **object**, ale pomocí konkrétnější třídy **Window**, a výsledek rozhodnutí zde toto okno obdrží přes

návratovou hodnotu metody (**bool**), kterou vyvolá např. při kliknutí na křížek v pravém horním rohu okna. Čtvrtý delegát (ř. 4) se pak může hodit pro předávání metody určitého výpočtu mezi dvěma čísly.

Pomocí delegátů totiž lze, krom určení typu události, také předávat odkazy na metody (jakoby reference na jejich „instance“). Takové „proměnné“ typu delegáta, pak stačí spustit, jako klasickou metodu pod jejím názvem (viz Kód 37).

```
 1: // Metoda pracující s výpočtem
 2: static double Suma(int[] a, int[] b, Vypocet vypocet)
 3: {
 4:     if (a == null || b == null || // Kontrola kompatiblity polí
 5:         a.Length == 0 || a.Length != b.Length) return 0;
 6:     double suma = 0; // Počítání sumy kombinací prvků z a a b
 7:     for (int i = 0; i < a.Length; i++)
 8:         suma += vypocet(a[i], b[i]); // Výpočet dodanou metodou
 9:     return suma;
10: }
11:
12: public static double SumaPrumeru(int[] cisla1, int[] cisla2)
13:     => Suma(cisla1, cisla2, Prumer);     // Odkaz na metodu
14:
15: static double Prumer(double a, double b) // Vyhovuje delegátovi
16:     => (a + b) / 2.0;
17:
18: public static double SumaSouctu(int[] cisla1, int[] cisla2)
19:     => Suma(cisla1, cisla2, (a, b) => a + b); // Anonymní metoda
20:                          anonymní metoda odpovídající předpsiu delegáta Vypocet
21: // Deklarace delegáta
22: public delegate double Vypocet(double a, double b); // Delegát
```

Kód 37 – Ukázka předávání metody jako „proměnné“ typu delegáta

Anonymní metody

Kromě toho, že metody můžeme zapsat klasickým způsobem, jako složku třídy (popř. struktury či záznamu), lze ji také vytvořit ad-hoc přímo v rámci zápisu kódu, třeba uvnitř nějaké jiné metody. Jelikož takové metody pak nejsou dostupné jinde než ve složce, v níž jsou zapsány, a mnohdy ani nemají svůj název (odkazujeme se na ně a voláme je přes proměnné typu **delegate**, **Action** či **Func**), říká se jim metody anonymní.

Vzhledem ke stylu jejich zápisu obvykle nemívají příliš dlouhý kód a řeší určitý konkrétní problém specifický pro dané místo, protože jinak by se vyplatilo zapsat je klasickým způsobem, který umožňuje použití i odjinud.

```
1:  var r = Console.ReadLine;    // r = odkaz na metodu ReadLine
2:  var rint = delegate () {     // rint = anonymní metoda (s kódem)
3:      return Convert.ToInt32(Console.ReadLine()); // kód a. metody
4:  };
5:  var w = delegate (string t) { Console.Write(t); }; // parametr t
6:
7:  Osoba osoba = new Osoba();      // Nová instance třídy Osoba
8:  var pozdrav = osoba.Pozdrav;    // Odkaz na nestatickou metodu
9:
10: w("Zadejte jméno: ");   // Vypsání do konzole přes an. metodu
11: osoba.Jmeno = r();      // Načtení z konzole odkazem na ReadLine
12: pozdrav();              // Pozdravení konkrétní osoby
13:
14: public class Osoba
15: {
16:     public string Jmeno { get; set; }            // Vlasntost
17:
18:     public void Pozdrav()                        // Metoda
19:         => Console.WriteLine("Ahoj {0}!", Jmeno);
20: }
```

Kód 38 – Ukázka deklarace a použití anonymních metod (statických i „instančních“)

Příklad (Kód 38) ukazuje možnosti vytvoření zkratky/proměnné (**r** pro přečtení řádku textu z konzole) odkazující na statickou metodu (ř. 1), vytvoření anonymní metody (**rint** pro čtení celého čísla z konzole) s návratovou hodnotou (ř. 2–4) a vytvoření anonymní metody (**w** pro výpis do konzole) se vstupním parametrem (**t** – textu, který se má vypsat), a odkazem (**pozdrav**) na nestatickou metodu (**Pozdrav** třídy **Osoba**), která je vztažena ke konkrétní instanci třídy a ve svém kódu používá hodnoty jejích vlastností (ř. 8).

Zde je zásadním rozdílem, že pro získání reference na statickou metodu se na ni odkazuje přes název třídy a název metody (např. `Console.ReadLine`, bez závorek, jinak bychom místo reference do proměnné uložili výsledek provedení této metody), zatímco u nestatických („instančních“) metod odkaz vede přes proměnnou s referencí na danou instanci a název metody (taktéž bez závorek, ty budou spolu s parametry zapsány až při jejím volání).

Deklarace vlastních anonymních metod pak obsahuje klíčové slovo **delegate**, kulaté závorky, jež mohou obsahovat seznam vstupních parametrů metody, a složené závorky s vlastním kódem metody. Ten v rámci těchto závorek může obsahovat libovolné množství příkazů.

Anonymní metoda z přechozí ukázky (Kód 37), která byla zapsána přímo v místě vstupního parametru metody **Suma** (ř. 21), se pak dokonce obejde bez klíčového slova **delegate**, jelikož je zde přímo „napasována“ na konkrétní typ metody (delegát, ř. 1) určený v deklaraci její hlavičky (ř. 4).

1.2.4 Události

Události (*events*) jsou co do deklarace podobné atributům (viz kap. 1.2.1), ovšem místo hodnot uchovávají odkazy na metody (k jedné události jich může být přiřazeno více), které by třída měla spustit ve chvíli, kdy k dané události dojde. Událost je při deklaraci od atributu rozlišena klíčovým slovem **event**. Metody, jež chceme přidat k odběru události, musí mít stejný

počet a datové typy vstupních parametrů (názvy se lišit mohou), jako předepisuje typ (*delegát*, viz str. 38) události (viz Kód 39).

```
1:  public class Auto                    typ události (object sender, EventArgs e)
2:  {
3:      public event EventHandler ZmenaPolohy;    // Událost
4:      public void Jed(double vzdalenost)        // Jeď
5:      {
6:          // ...
7:          if (ZmenaPolohy != null)  // Odebírá tuto událost někdo?
8:              ZmenaPolohy(this, EventArgs.Empty); // Jeho spuštění
9:      }
10: }
```

Kód 39 – Ukázka deklarace a vyvolání události základního typu

Kód 39 ukazuje příklad deklarace (ř. 3), testu (ř. 7) a vyvolání (ř. 8) události základního předdefinovaného typu delegáta **EventHandler**. Pokud se k odběru události žádná metoda nepřihlásí, bude v této „proměnné“ (**ZmenaPolohy**), hodnota **null**, a pokus o její zavolání (ř. 8) by skončil chybovým hlášením. Proto je nezbytné vždy událost nejprve otestovat není-li null (ř. 7) a pokud ano, její spouštění neprovádět (událost, která nikoho nezajímá, není třeba ohlašovat).

Toto ověření se dá provést i kratším způsobem, rovnou v rámci vyvolávání události, pomocí tzv. *Elvis operátoru* „**?.**“ (viz Kód 40), který v části příkazu napravo od něho pokračuje pouze v případě, že nalevo od něho nevyšlo **null**. Reflexní metoda **Invoke** pak zařídí spuštění metody předávané takto přes „proměnnou“ (**ZmenaPolohy**).

```
ZmenaPolohy?.Invoke(this, EventArgs.Empty);
```

Kód 40 – Ukázka vyvolání události včetně testu, že není *null* (náhrada ř. 8–9 v Kód 39)

Událost v ukázce Kód 39 (řádek 3) je tedy typu **EventHandler**, což je základní předdefinovaný delegát, který svou deklarací předepisuje metodám, jež by danou událost chtěly odebírat, že musí mít dva parametry: první typu **object** (*sender*) a druhý typu **EventArgs** (*e*). První parametr by tak měl předávat referenci (viz 1.1.2) na instanci třídy, která událost vyvolává (zde tedy objekt typu **Auto**, což zajišťuje klíčové slovo **this**), druhý parametr pak bývá určen pro předávání dalších dodatečných podrobností souvisejících s událostí. V tomto základním typu volání je však pro druhý parametr **e** použit pouze typ **EventArgs**, který sám o sobě nemá žádné vlastnosti ani veřejné atributy, kromě statického atributu pouze pro čtení **Empty**, obsahující referenci na prázdnou instanci sebe sama. Událost tak pouze oznámí, že došlo ke změně pozice auta, nepředává však žádné dodatečné informace (např. původní či novou polohu), ty si ovšem lze zjistit z hodnot vlastností daného objektu (odesílatele události – **sender**). A právě takovouto hlavičku pak musí mít metody, které se hlásí k odběru této události (viz Kód 41).

```
1:  public class Navigace
2:  {
3:      public Navigace(Auto auto)
4:      {
5:          // Přihlášení metody k odběru události ZmenaPolohy
6:          auto.ZmenaPolohy += Auto_ZmenaPolohy;
7:      }
8:
9:      private void Auto_ZmenaPolohy(object sender, EventArgs e)
10:     {
11:         // Reakce na událost změny polohy auta (sender)...
12:     }
13: }
```

reference na instanci třídy Auto, které změnilo polohu

vstupní parametry metody odpovídají předpisu typu EventHandler

Kód 41 – Ukázka přihlášení metody *Auto_ZmenaPolohy* k odběru události *ZmenaPolohy* konkrétní instance třídy *Auto* (Kód 39) předané v parametru konstruktoru

1.2.5 Vnořené členy

Jednotlivé členy (třídy, struktury, záznamy, delegáty, výčty apod.), které jsou obvykle definovány samostatně pouze v rámci nějakého jmenného prostoru, lze také do sebe vnořovat, tzn. že například uvnitř třídy je možné zapsat kód třídy jiné, kterážto tak bude třídou vnořenou, což lze provádět i opakovaně (viz Kód 42, ř. 1–5).

Tito vnoření členové mívají obvykle pomocný význam uvnitř dané třídy a zvenčí se jmenovitě nepoužívají (dostupnost mají standardně **private**, viz Tab. 2 na str. 14). Nicméně, jsou-li deklarovány jako veřejné, tak je lze používat i zvenčí, jen se na ně je třeba odkazovat i názvem třídy, do které jsou vnořeny (viz Kód 42, ř. 9–10). Pro tento jejich složený název (např. `A.B.C`) lze ovšem definovat i zkratku (např. `using C = NameSpace.A.B.C`) v rámci definicí **using** (viz kap. 2.4) a používat pak pouze ji (`C`).

```
 1: public class A {              // Samostatná třída A
 2:     public class B  {         // Vnořená třída
 3:         public class C  {  } // Třída vnořená do vnořené třídy
 4:     }
 5: }
 6:
 7: public class D                // Samostatná třída D
 8: {
 9:     public A.B   CreateInstanceOfB() => new A.B();
10:     public A.B.C CreateInstanceOfC() => new A.B.C();
11: }
```

Kód 42 – Ukázka vnořených tříd a odkazování se na ně

Jako veřejné se obvykle vnořují členy, jejichž přímé volání zápisem v kódu se pak odjinud nepoužívá. Například jde o delegáty, které definují parametry metody použité jako událost u třídy, do které jsou vnořeny. Metody takového typu, byť se pak používají i jinde, tak přímo název delegáta se již

nikde znovu neuvádí, pouze se na základě něho (ve Visual Studiu převážně automaticky) vygeneruje hlavička metody pro odběr dané události.

Třídy zapsané v konzolové aplikaci od verze .NET 6 (C# 10) v souboru **Program.cs** za kódem (před ani mezi ním být nesmí), vnořené do třídy **Program** nejsou, protože při kompilaci jsou umístěny jako samostatné (viz).

1.3 Parciální třídy

Parciální (*partial*, též částečné) třídy jsou z pohledu jejich používání zcela klasickými třídami. Rozdíl je v možnosti zápisu jejich kódu (zápisu složek tříd). Standardně totiž platí, že kód třídy je vždy zapsán v jediném souboru a v jediné deklaraci (pod jednou hlavičkou **class**) a při pokusu hlavičku téže třídy se stejným názvem (v rámci jednoho jmenného prostoru, viz kap. 2.4) dojde k chybě. Přidáme-li však do hlavičky třídy klíčové slovo **partial** (to pak musí být ve všech jejích částech), kód třídy tak může být rozepsán do více částí, které mohou být rozmístěné ve více souborech.

```
1:  // Soubor Osoba.cs
2:  public partial class Osoba
3:  {
4:      public string Jmeno
5:          { get; set; }
6:      public string Prijmeni
7:          { get; set; }
8:  }
```

```
// Soubor OsobaRozsireni.cs
public partial class Osoba
{
    public string CeleJmeno =>
        $"{Jmeno} {Prijmeni}";
}
```

Kód 43 – Ukázka parciální třídy

Můžeme tak jednak rozdělit a zpřehlednit delší kódy rozsáhlejších tříd, ale především takovým třídám přidávat další složky, aniž bychom museli zasahovat do jejich hlavního kódu. Toho se například využívá ve Windows

Forms, kde je design oken vytvořený přes WYSIWYG editor zapsaný prostřednictvím automaticky generovaných klasických C# příkazů oddělen od kódu na pozadí, který píše programátor ručně. Obdobně pak postupují i další typy .NET projektů. Kupříkladu složky tříd, automaticky generovaných při používání ORM[5], se běžně rozšiřují tímto způsobem, jelikož jinak bychom o tento přidaný kód při každé aktualizaci (přegenerování třídy, aby odpovídala změnám v DB) přišli.

Kód parciálních tříd je při kompilaci spojen do jediného, takže tato možnost rozdělení kódu tříd (na rozdíl třeba od dědičnosti, viz kap. 3) je dostupná vždy pouze v rámci jednoho projektu (viz kap. 2.5).

[5] ORM (objektově relační mapování) zjednodušeně řečeno mapuje databázové tabulky na kód třídy tak, že vlastnostem těchto tříd, jež odpovídají sloupcům tabulek v relační DB, přidá obslužný kód, který změny jejich hodnot u konkrétních instancí ukládá do příslušného záznamu (řádku) této tabulky v DB (viz např. *Entity Framework Core*).

2 Další členové

Sice by se mohlo zdát, že s pomocí tříd lze vyřešit libovolný programovací scénář, nicméně v C# je k dispozici ještě řada dalších členů a konstrukcí, které lze používat, a jejichž vlastnosti jsou pro určité případy užití mnohem vhodnější.

2.1 Struktury

Struktury (*struct*) jsou třídám velmi podobné. Stejně jako ony mohou obsahovat atributy (kap 1.2.1), vlastnosti (kap. 1.2.2), metody (kap 1.2.3) se všemi jejich možnostmi, události (kap. 1.2.4) i vnořené členy (kap. 1.2.5). Podobně jako u tříd se i u struktur vytvářejí jejich instance (viz kap. 1.1.1), které však fungují s určitými rozdíly. Strukturované proměnné také nelze ve výchozím stavu porovnávat (např. `if (s1 == s2)…`), pokud v kódu struktury nejsou přetíženy operátory porovnání.

2.1.1 Výchozí hodnoty struktury

V hodnotových proměnných založených na strukturách nemůže být **null**. To je určeno pouze pro referenční typy a značí, že reference neexistuje, kdežto hodnota vždy nějaká být musí (např. proměnná typu **int** také vždy obsahuje celé číslo). Proto také kompilátor hlídá, abych se v kódu nedocházelo ke čtení hodnot z proměnných, které dosud nebyly deklarovány. Nicméně v určitých případech to ohlídat nelze (např. u atributů nebo při vytvoření pole: `var pole = Struktura[10]`) a pak přichází na řadu výchozí hodnoty. U referenčního typu je v takových případech tedy výchozí **null**[6], klasické hodnotové typy mají výchozí hodnotu pevně stanovenou (např. u čísel je to nula). No a právě tyto hodnoty jsou dosazeny do atributů struktury, která

[6] Datový typ **string** (*String*) není hodnotového typu, ale třída, a jeho výchozí hodnotou tak není prázdný textový řetězec **""**, ale **null**. Je však naprogramován tak, že při předávání takové hodnoty mezi proměnnými se vytváří vždy nová kopie stejně jako u struktur.

vznikne takto „samovolně“ (viz Kód 44). Výchozí hodnotu hodnotového typu lze získat pomocí klíčového slova **default** a názvu příslušné struktury (např. `int x = default(int)`).

```
1: var a = new Struktura[5];  // Pole pro každý člen dosadí default
2: Console.WriteLine(a[3].X); // Vypíše: 0
3:
4: struct Struktura {
5:     public int X;          // Výchozí hodnota int je 0
6: }
```

Kód 44 – Ukázka dosazení výchozí hodnoty struktury i jejího atributu

2.1.2 Konstruktor struktury

Při výše uvedeném způsobu „samovolného“ vytváření instancí struktury pomocí výchozích hodnot bez klíčového slova **new** se nepoužívá konstruktor (viz str. 34) struktury, ani případné výchozí hodnoty atributů nastavené jim v rámci jejich deklarace (např. viz Kód 6, ř. 5), ale čistě dosazování výchozích hodnot. Konstruktor nicméně může být struktuře deklarován, ovšem pro to, aby se použil, musí být v kódu explicitně zavolán (např. `var s = new Struktura()`).

Konstruktor struktury, pokud tedy nějaký deklarujeme (může jich být i přetíženo více), povinně musí, na rozdíl od třídy, ve svém kódu nastavit hodnotu všem atributům a automatickým vlastnostem struktury, které nemají přiřazenu výchozí hodnotu ve své deklaraci[7] (viz Kód 45).

[7] Přiřazovat hodnotu atributům u struktur v rámci jejich deklarace lze až od C# 10, dříve tato možnost nebyla dostupná, a stejně tak nebylo povoleno deklarovat bezparametrický konstruktor.

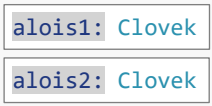
```
1: public struct Bod3D
2: {
3:     public int X;     // Atribut bez výchozí hodnoty
4:     public int Y = 0; // Atribut s výchozí hodnotou
5:     public int Z { get; set; } // Automatická vlastnost
6:
7:     public Bod3D()          // Bezparametrický konstruktor
8:         => (X, Z) = (0, 0); // Y má výchozí hodnotu,
                               // nemusí tak zde být nastavováno
9:     public Bod3D(int x, int y, int z) // Konstruktor s parametry
10:        => (X, Y, Z) = (x, y, z);
11: }
```

Kód 45 – Ukázka povinného nastavení hodnot všech atributů a automatických vlastností ve všech konstruktorech struktury

2.1.3 Automatické kopírování instancí

Hlavním rozdílem struktury oproti třídě je, že struktury nejsou referenčního typu, ale typu hodnotového (defaultně nedědí ze třídy **Object**, ale ze třídy **ValueType**, a až ta dědí z **Object**), stejně jako například typy **int** (**Int32**) či **bool** (**Boolean**), které jsou také strukturami. Důsledkem tohoto faktu je, že každé přiřazení již vytvořené instance struktury vytvoří její kompletní kopii, tedy novou instanci (viz Kód 46, porovnejte s Kód 1 na str. 10, kdy bylo totéž provedeno se třídou).

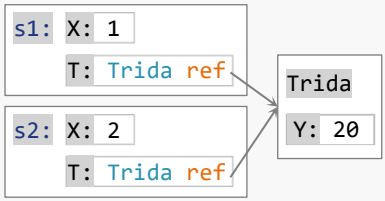
```
1: Clovek alois1 = new Clovek();
2: Clovek alois2 = alois1;
3: public struct Clovek { }
```



Kód 46 – Ukázka kódu a schématu přiřazení instance struktury *Clovek* do jiné proměnné, čímž vznikne její kopie

Vytvoření kopie struktury probíhá automaticky, přičemž jsou zkopírovány veškeré hodnoty jejích atributů (popř. automatických vlastností) do této vytvářené kopie. Atributy hodnotového typu (a případné instance struktur) jsou zkopírovány klasickým způsobem, pokud má struktura i hodnoty referenčního typu (tj. typu nějaké třídy), bude zkopírována pouze reference na daný objekt, který zůstane v paměti pouze jeden (viz Kód 47).

```
1:  var s1 = new Struktura() { // Nová instance struktury
2:      X = 1,                 // Nastavení X
3:      T = new Trida()};      // Instance třídy
4:  var s2 = s1; // Vytvoří kopii struktury
5:  s2.X   =  2; // s1.X:    1, s2.X:    2
6:  s2.T.Y = 20; // s1.T.Y: 20, s2.T.Y: 20
7:
8:  class Trida {              // Deklarace třídy
9:      public int Y = 10; // Výchozí hodnota
10: }
11:
12: struct Struktura {  // Deklarace struktury
13:     public int X;   // Proměnná hodnotového typu
14:     public Trida T; // Referenční proměnná
15: }
```



Kód 47 – Ukázka kódu a schématu vzájemné závislosti dvou proměnných referenčního typu

Kopírování se změnou hodnoty

Při každém předání hodnoty do nové proměnné je tedy vytvořena nová nezávislá kopie instance struktury. Jelikož k takovémuto kopírování dost často dochází za účelem následných změn hodnot ve vzniklé kopii, byla přidána možnost tyto změny provést rovnou již v rámci samotného kopírování (tu později oceníme především u záznamů, viz kap. 2.2, třídy jí však nedisponují). Zároveň je to jediná možnost, jak u kopie změnit hodnoty vlastností

majících **init** (lze je nastavit jen při inicializaci, tj. v konstruktoru či v rámci vytváření nové instance) místo **set**.

Syntaxe je taková, že při přiřazování jedné proměnné do druhé, následně uvedeme klíčové slovo **with** a poté do složených závorek, podobně jako při vytváření nové instance, vypíšeme názvy veřejných vlastností či atributů a jim přiřadíme nové hodnoty, v rámci jednoho příkazu (viz Kód 48).

```
1: var alois = new Clovek() { Jmeno = "Alois", Vek = 20 };
2: var bara = alois with { Jmeno = "Bára" };
3:
4: public struct Clovek
5: {
6:     public string Jmeno { get; init; }
7:     public int Vek { get; set; }
8: }
```

Kód 48 – Ukázka kopírování struktury současně se změnou hodnoty vlastnosti

2.1.4 Null u hodnotového typu proměnné

V proměnné hodnotového typu (např. **int**, či jakékoli vlastní struktury) tedy nemůže být **null**, nicméně pokud by byla tato variant zapotřebí, lze použít strukturu s možností nulování (*nullable struct*). Tyto rozšířené typy lze deklarovat dvojím způsobem: plným zápisem definice takového typu pomocí generické struktury **Nullable** (např. `Nullable<int>`) nebo zkráceným, kdy se za název struktury pouze přidá otazník (např. `int?`).

Proměnná tohoto typu pak v obou případech zaobaluje původní strukturu do systémové struktury **Nullable**, která ji rozšiřuje o možnost nabývat hodnoty **null**. Tu tak lze takové proměnné klasicky přiřazovat (např. `int? x = null`) a ověřovat je-li v ní **null** a to hned dvojím způsobem: buď

klasicky porovnáním (např. `if (x == null) …`), nebo přes přidanou vlastnost pouze pro čtení **HasValue** typu **bool** (např. `if (x.HasValue) …`).

Nenulové hodnoty či instance struktury se přiřazují a čtou stejným způsobem (např. `int? x = 7`). Potřebujeme-li však přistupovat ke složkám takové struktury, pak se k její instanci (není-li **null**) dostaneme přes vlastnost **Value** (např. zatímco z proměnné **d** typu klasické struktury **DateTime** zjistíme rok jako `d.Year`, tak ze struktury s možností nulování **DateTime?** by to bylo `d.Value.Year`, viz Kód 49).

```
1: static int Vek(DateTime dNarozeni, DateTime? kDatu = null)
2: {
3:     if (!kDatu.HasValue)        // Je-li paremtr kDatu null…
4:         kDatu = DateTime.Today; // … nastaví se mu dnešní datum
5:     return kDatu.Value.Year - dNarozeni.Year -
6:         (kDatu.Value.DayOfYear < dNarozeni.DayOfYear ? 1 : 0);
7:     // Místo kDatu.Value lze též použít (kDatu ?? DateTime.Today)
8: }
```

Kód 49 – Ukázka práce se strukturou s možností nulování v metodě pro výpočet věku

2.1.5 Struktury pouze pro čtení

Struktury mohou být dále označeny jako pouze pro čtení, přidáním klíčového slova **readonly** do jejich hlavičky (před klíčové slovo **struct**). Toto označení struktuře sice žádnou funkcionalitu nepřidá, ale ohlídá, aby veškeré její atributy byly pouze pro čtení (označeny také modifikátorem **readonly**, viz str. 18) a vlastnosti neměly část **set** (mohou mít buď **init**, nebo nic, resp. pouze část pro čtení **get**).

Jelikož struktury nemohou mít výchozí hodnoty u atributů ani vlastností, je třeba všechny jejich hodnoty nastavit v konstruktoru struktury. U **init** vlastností lze sice hodnotu ještě změnit v rámci vytváření této instance

(viz str. 23), ale i tak musí být v konstruktoru vlastnosti nějaká hodnota nastavena (viz Kód 50).

```
var alois = new Clovek(2001) {
    Jmeno = "Alois"   // Přepíše výchozí hodnotu z konstruktoru
};

public readonly struct Clovek   // Strukutra jen pro čtení
{
    public string Jmeno { get; init; } // init vlastnost
    public int Vek { // Vlasntost jen pro čtení (bez set kódu)
        get => DateTime.Today.Year - RokNarozeni; }
    public readonly int RokNarozeni;   // Atribut jen pro čtení

    public Clovek(int rokNarozeni) {   // Konstruktor struktury
        Jmeno = "Neznámé";    // Výchozí hodnota (musí zde být)
        RokNarozeni = rokNarozeni; // Hodnotu již nepůjde změnit
    }
}
```

Kód 50 – Ukázka struktury pouze pro čtení

2.2 Záznamy

Záznamy (*record*) jsou další alternativou ke třídám a strukturám. Předpokládaným případem jejich užití je především u datových objektů, tedy místo tříd s převážně automatickými vlastnostmi (popř. atributy), jejichž hodnoty se nastaví pouze jednou a nebudou se již měnit (ideálně pojištěno použitím **init** místo **set**), a nejlépe bez vlastního kódu (metod). Použít v záznamech veškeré složky jako ve třídách (viz str. 15) sice lze, ale pokud je tam (krom vlastností) potřebujeme v nějaké větší míře, bude lepší použít místo záznamu rovnou třídy.

2.2.1 Záznamy jako třídy

Záznamy se v základní formě chovají jako třídy (jsou referenčního typu), a při kompilaci se dokonce záznam na třídu převede (viz). Této třídě vzniklé ze záznamu je však také automaticky přidáno několik složek. Jde především o přetížení operátorů pro porovnání (metody **Equals**, **==** a **!=**) a o metodu pro převod objektu na textový řetězec (**ToString**).

Použitím klíčového slova **record** místo **class** tedy získáváme třídu, které se při kompilaci automaticky doplní části kódu, jež se u těchto typů tříd velmi často stejně běžně řeší.

Porovnávání záznamů

Při porovnávání proměnných s referencemi na jednotlivé instance (např. `a1 == a2`) záznamů tak rovnost nastane nejen pokud obě proměnné mají referenci na stejnou instanci, ale dochází k porovnávání hodnot všech jejich vlastností a atributů, a pokud se tyto rovnají, budou se rovnat i tyto dvě proměnné (objekty), byť jde o různé instance (viz Kód 51, ř. 4, srovnejte s Kód 5, ř. 5–6 na str. 12). Shodný tedy samozřejmě musí být i datový typ obou porovnávaných proměnných.

```
 1: var a1 = new Clovek();
 2: var a2 = new Clovek();
 3:
 4: Console.WriteLine(a1 == a2); // True (hodnoty složek se rovnají)
 5:
 6: Console.WriteLine(a1);   // "Clovek { Jmeno = Alois, Vek = 20 }"
 7:
 8: public record Clovek {
 9:     public string Jmeno = "Alois";       // Atribut
10:     public int Vek { get; set; } = 20;   // Vlastnost
11: }
```

Kód 51 – Ukázka kódu porovnávajícího dvě různé instance záznamu

Převod záznamu na textový řetězec

Kód 51 také ukazuje použití metody **ToString** při výpisu proměnné **a1** (ř. 6). Zatímco pokud by **Clovek** byla třída, vypsal by se pouze její název („Clovek“, popř. by se před něj přidal ještě název jmenného prostoru, ve kterém je třída deklarována), tak u záznamu se do složených závorek za tento název vypíší i všechny jeho veřejné vlastnosti a atributy s jejich aktuální hodnoty (např. „Vek = 20“, viz komentář na ř. 6).

Zjednodušená deklarace záznamů

Jelikož se tedy záznamy téměř výhradně skládají z vlastností, jejichž hodnoty se navíc definují jen v okamžiku vytváření instance záznamu, byl pro jejich deklaraci přidán alternativní způsob zápisu, který ji může ještě více zkrátit (zestručnit, zjednodušit a zpřehlednit).

Tento způsob deklarace se trochu podobá metodám, kdy jsou za název třídy (přímo do její hlavičky) přidány kulaté závorky, a v nich jednotlivé vlastnosti vyjmenovány jako vstupní parametry „metody“ (resp. konstruktoru třídy). U každé je tedy uveden nejprve její datový typ, pak její název (jelikož jde o název vlastnosti nikoli parametru, je vhodné jej zapisovat s prvním velkým písmenem), a případně může následovat ještě znak přiřazení `=` a výchozí (defaultní) hodnota (viz Kód 52, ř. 12–15).

Mějme však na paměti, že hodnoty těchto vlastností již nebude možné později změnit, tak je třeba dobře zvážit, má-li se daná vlastnost díky své výchozí hodnotě stát nepovinnou (či bez ní zůstat povinnou), aby se pak při vytváření nové instance záznamu nezapomínalo na její nastavení. Uvnitř deklarace třídy pak lze nezávisle na těchto parametrech přidat i její další složky, např. třeba další vlastnosti (viz Kód 52, ř. 17).

```
1: var a = new Clovek("Alois") { Vyska = 165, Prijmeni = "Malý" };
2: var b = new Clovek("Bára", "Nováková");
3: var c = new Clovek("Ctirad", Vek: 45) { Vyska = 170 };
4:
5: // Clovek { Jmeno = Alois, Prijmeni = Malý, Vek = 20, Vyska = 165 }
6: Console.WriteLine(a);
7: // Clovek { Jmeno = Bára, Prijmeni = Nováková, Vek = 20, Vyska = 160 }
8: Console.WriteLine(b);
9: // Clovek { Jmeno = Ctirad, Prijmeni = Neznámé, Vek = 45, Vyska = 170 }
10: Console.WriteLine(c);
11:
12: public record Clovek(
13:     string Jmeno,                 // Povinná vlastnost
14:     string Prijmeni = "Neznámé", // Vlasntost s výchozí hodnotou
15:     int Vek = 20)                 // Také nepovinná vlasnost
16: {
17:     public int Vyska { get; init; } = 160; // Klasicky zapsná v.
18: }
```

Kód 52 – Ukázka kódu zkrácené deklarace záznamu a varianty jeho použití

Tyto „parametry“ jsou pak při kompilaci přeloženy jako vlastnosti mající místo **set** uvedeno klíčové slovo **init**, což znamená, že jejich hodnotu lze určit pouze výchozí hodnotou nebo v konstruktoru. Třídě vzniklé ze záznamu při kompilaci je také přidán konstruktor s týmiž parametry jako byly uvedeny v hlavičce třídy a v jeho kódu jsou jejich hodnoty nastaveny těmto vlastnostem. Pro vytvoření nové instance třídy je pak nezbytné použít tento konstruktor, v něm zadat hodnoty alespoň všem povinným parametrům, a případné další hodnoty lze dodefinovat ve složených závorkách za tímto voláním konstruktoru v rámci tohoto příkazu (viz Kód 52, ř. 1–3).

Klonování objektů

Záznamy se tedy v základu chovají jako třídy a pokud přiřadíme do proměnné jinou proměnnou takového typu (např. `var r1 = r2`), předáváme

tím pouze referenci na jedinou instanci (objekt) v paměti. Záznamy však disponují i možností, jak instanci zkopírovat do nové, přičemž je možné (nikoli však povinné) některé hodnoty do nového záznamu v rámci tohoto kopírování změnit, což se určitě hodí, vzhledem k tomu, že jinak bývají vlastnosti záznamu často pouze pro čtení (**init**). Tento způsob je podobný kopírování struktur se změnou hodnoty (viz Kód 48 na str. 51). Má-li jít o čistou kopii instance záznamu beze změn, stačí složené závorky za **with**, určené pro nastavení nových hodnot vlastností či atributů záznamu, nechat prázdné (viz Kód 53).

```
1: var a1 = new Clovek() { Jmeno = "Alois", Vek = 20 };
2: var a2 = a1;                // Předání reference na stejný objekt
3: var a3 = a1 with { Vek = 25 }; // Klon s úpravou hodnot(y)
4: var a4 = a1 with { };          // Klon objektu beze změn
5:
6: public record Clovek
7: {
8:     public string Jmeno { get; init; }
9:     public int Vek { get; init; }
10: }
```

Kód 53 – Ukázka klonování (kopírování) instance záznamu s (ř. 3) a beze změny (ř. 4) hodnot vlastností

2.2.2 Záznamy jako struktury

Záznamy se tedy při kompilaci překládají na třídy s určitým přidaným kódem, a při přiřazování do dalších proměnných předávají pouze svou referenci (viz Kód 1 na str. 10). Ukázali jsme si, jak tuto vlastnost obejít a pomocí **with** vytvářet klony (kopie) objektu (viz Kód 53), podobně jako když se přiřazují struktury.

Pokud bychom tuto funkcionalitu „kopírování“ upřednostňovali před předáváním referencí, lze záznamy upravit tak, aby se z nich při kompilaci

místo tříd generovaly struktury. Za tímto účelem stačí do deklarace záznamu za klíčové slovo **record** přidat ještě klíčové slovo **struct** (viz Kód 54).

```
1:  var a1 = new Zaznam() { X=1 };
2:  var a2 = a1;
3:  a2.X = 2; // Upravuje spol. obj.
4:
5:  Console.WriteLine(a1); // X=2
6:  Console.WriteLine(a2); // X=2
7:
8:  record Zaznam {
9:      public int X;
10: }
```

```
var a1 = new Zaznam() { X=1 };
var a2 = a1;
a2.X = 2; // Upravuje svou kopii

Console.WriteLine(a1); // X=1
Console.WriteLine(a2); // X=2

record struct Zaznam {
    public int X;
}
```

Kód 54 – Porovnání klasického záznamu (*record*) a záznamu který se překládá jako struktura (*record struct*)

Záznamy pouze pro čtení

Stejně jako lze struktury opatřit modifikátorem **readonly**, který ohlídá, aby veškeré její atributy a vlastnosti byly pouze pro čtení (viz kap. 2.1.5 na str. 52), lze stejně rozšířit (resp. ohlídat omezení) i záznam, který má být zpracován jako struktura (**readonly record struct**, pro klasický záznam bez **struct** není tento modifikátor dostupný). Díky tomu kompilátor striktně ohlídá, aby se nezapomněl některý z atributů či vlastností omezit pro zápis mimo konstruktor (popř. u vlastností v rámci vytváření nové instance záznamu). Ukázku takového záznamu obsahuje Kód 55 (porovnejte s Kód 50 na str. 53).

```
1:  public readonly record struct Clovek   // Záznam jen pro čtení
2:  {
3:      public string Jmeno { get; init; } // init vlastnost povinně
4:      public readonly int RokNarozeni;   // Atribut jen pro čtení
5:
6:      public Clovek(int rokNarozeni) {   // Konstruktor záznamu
7:          Jmeno = "Neznámé";   // Výchozí hodnota (musí zde být)
8:          RokNarozeni = rokNarozeni; // Hodnotu již nepůjde změnit
9:      }
10: }
```

Kód 55 – Ukázka záznamu pouze pro čtení

Vytvořením instance takovéhoto záznamu tak získáme strukturu podobnou té, kterou vytvořil Kód 50 (str. 53). Ovšem jelikož se zároveň jedná o záznam, lze proměnné s těmito instancemi porovnávat (což u struktur standardně nelze vůbec), přičemž rovnost bude závislá na rovnosti hodnot všech veřejných vlastností a atributů, a navíc získáme propracovanější převod této proměnné na text (metoda **ToString**). Totéž platí i u verze záznamu bez modifikátoru **readonly**.

2.3 Výčty

Výčty (*enum*) jsou určeny pro vytváření vlastních hodnotových typů, přičemž veškeré přípustné hodnoty jsou vyjmenovány v deklaraci tohoto výčtu. Předpokladem je tak nižší, rozhodně však konečný, počet hodnot, jež jsou do výčtu zařazeny pod svým názvem. Jako příklad si tak můžeme představit třeba dny v týdnu (viz Kód 56), měsíce v roce, roční období, ale třeba také základní barvy, klávesy na klávesnici, tlačítka myši apod.

```
1: public enum DnyVTydnu
2: {
3:     Pondeli,
4:     Utery,
5:     Streda,
6:     Ctvrtek,
7:     Patek,
8:     Sobota,
9:     Nedele,
10: }
```

Kód 56 – Ukázka kódu základní deklarace výčtu

Takovýto výčet pak lze používat jako typ pro proměnné, parametry, atributy, vlastnosti apod. Hodnota je pak v kódu vždy uváděna pod svým plným názvem, tzn. nejprve je uveden název výčtu, následuje tečka a až za ní název konkrétní hodnoty (např. viz Kód 57).

```
DnyVTydnu volnyDen = DnyVTydnu.Patek;
```

Kód 57 – Ukázka použití hodnoty výčtu

2.3.1 Číselná reprezentace hodnot výčtu

Jednotlivým hodnotám výčtu jsou zároveň přiřazeny číselné alternativy typu **int** (**Int32**). Neurčíme-li je explicitně ani jedné z hodnot (jako třeba v Kód 56), jsou jim přiřazeny automaticky, a to od nuly výše v pořadí v jakém jsou hodnoty v deklaraci výčtu zapsány (tzn. `Pondeli = 0, Utery = 1, … Nedele = 6`).

Pokud neexistuje vztah mezi hodnotou a jí přiřazenému číslu, popř. číselnou reprezentaci hodnot nebudeme v kódu používat, pak je takovýto stav dostačující. Potřebujeme-li však s číselnými verzemi hodnot nějak dále pracovat, lze jim je snadno přiřadit v rámci deklarace výčtu (viz Kód 58 vlevo).

```
 1: public enum DnyVTydnu
 2: {
 3:     Pondeli = 1,
 4:     Utery   = 2,
 5:     Streda  = 3,
 6:     Ctvrtek = 4,
 7:     Patek   = 5,
 8:     Sobota  = 6,
 9:     Nedele  = 7,
10: }
```

```
public enum DnyVTydnu
{
    Pondeli = 1,
    Utery,
    Streda,
    Ctvrtek,
    Patek,
    Sobota,
    Nedele,
}
```

Kód 58 – Ukázka dvou variant deklarace výčtu dnů v týdnu s číselnými hodnotami 1-7

Tyto číselné hodnoty nemusí tvořit souvislou řadu (mohou být přiřazovány v různém pořadí, např. `Pondeli = 100, Utery = 250, … Nedele = 30`). Čísla u jednotlivých hodnot se dokonce mohou i opakovat (např. `Pondeli = 10, Utery = 10`). Takovéto způsoby očíslování hodnot výčtu jsou nicméně spíše kontraproduktivní.

Pokud ale naopak chcete použít souvislou řadu čísel, jen potřebujete, aby místo nulou začínala např. jedničkou, stačí tuto výchozí hodnotu nastavit u první hodnoty výčtu a těm ostatním bude přiřazena hodnota následující po té předchozí (viz Kód 58 vpravo).

Můžeme-li se na určitý souvislý způsob číslování spolehnout, pak s ním lze provádět i základní výpočty, jako že zítra je dnes+1, pohlídáme-li si přechod z neděle (7) na pondělí (1) apod. Kód 59 např. ukazuje metody pro určení včerejšího (ř. 1–2) a zítřejšího (4–6) dne v týdnu s pomocí výčtu **DnyVTydnu** (viz Kód 58).

```
 1: DnyVTydnu Vcera(DnyVTydnu dnes)
 2:     => dnes == DnyVTydnu.Pondeli ? DnyVTydnu.Nedele : dnes-1;
 3:
 4: DnyVTydnu Zitra(DnyVTydnu dnes)
 5:     => dnes == DnyVTydnu.Nedele ? DnyVTydnu.Pondeli : dnes+1;
 6:
 7: int cisloDneUS = (int)DateTime.Today.DayOfWeek;    // 0-6 Ne-So
 8: int cisloDneCZ = cisloDneUS == 0 ? 7 : cisloDneUS; // 1-7 Po-Ne
 9: DnyVTydnu dnes = (DnyVTydnu)cisloDneCZ;     // Pondeli - Nedele
10:
11: Console.WriteLine("Včera: {0}", Vcera(dnes));
12: Console.WriteLine("Dnes:  {0}", dnes);
13: Console.WriteLine("Zítra: {0}", Zitra(dnes));
```

Kód 59 – Ukázka možnosti výpočtů s číselnou reprezentací hodnot výčtu

Na ř. 7–9 pak Kód 59 ukazuje, jak lze pro tento náš výčet určit aktuální den v týdnu prostřednictvím struktury **DateTime**. Ta vrací aktuální datum a pro něj umí určit den v týdnu ve vlastním systémovém výčtu **DayOfWeek**, který je podobný tomu našemu (**DenVTydnu**), ovšem dny čísluje od 0 do 6 a navíc používá anglický týden, tj. prvním dnem v týdnu (0) je neděle a posledním sobota (6). Díky možnosti převést hodnotu výčtu na číslo obyčejným přetypováním (ř. 7), můžeme snadnou úpravou (ř. 8) toto číslo převést na naše číslování (1–7 pro Po–Ne), a to následně přetypovat tentokrát již na náš výčet **DnyVTydnu** (ř. 9).

2.3.2 Souhrnné hodnoty výčtu (příznaky)

Proměnné typu výčtu obvykle obsahují právě jednu z jeho hodnot. Je zde ale i možnost nastavit výčet tak, aby proměnná jeho typu mohla obsahovat jeho hodnot vícero (např. by proměnná typu **DnyVTydnu** mohla obsahovat všechny pracovní dny, tedy Po–Pá, nebo dny víkendu So–Ne).

Pro tuto funkčnost je však nezbytné každé jednotlivé hodnotě výčtu explicitně nastavit jejich číselnou reprezentaci, a to tak, aby se z výsledného součtu jejich libovolné kombinace dalo zpětně určit, které z hodnot tam zahrnuty byly a které nikoli. Jasnou volbou pro tyto účely je řada tvořená mocninami čísla dvě (2^0=1, 2^1=2, 2^2=4, 2^3=8, 2^4=16, 2^5=32, 2^6=64, 2^7=128, 2^8=256…). Binární reprezentace těchto hodnot (dvojková soustava: `0001`, `0010`, `0100`, `1000`…) totiž pro každou hodnotu poskytuje informační bit, kde může být buď `0` (hodnota zahrnuta není) nebo `1` (hodnota zahrnuta je).

Takže například číslo 13, binárně zapsané jako `1011` nese informaci, že zahrnuty jsou hodnoty s čísly 1, 4 a 8. Chceme-li takovouto souhrnnou hodnotu vypočítat, stačí čísla jednotlivých hodnot, které do ní chceme zahrnout buď sečíst (1+4+8 = 13) nebo na binární úrovni propojit operátorem **OR**[8] (`0001 OR 0100 OR 1000 = 1101`, tj. $2^0+2^2+2^3$ = 1+4+8 = 13).

Dekódování jednotlivých hodnot pak lze provádět jak na binární úrovni (příslušný bit obsahuje `0` nebo `1`) nebo zpětným odečítáním od nejvyšší možné hodnoty (zde 8) až po tu nejnižší (1). Čili 13-8 provést lze, hodnota s číslem 8 je tedy zahrnuta a dále pokračujeme s 5 (výsledek rozdílu 13-8). Dále 5-4 lze, 4 je zahrnuta, pokračujeme s 1 (výsledek 5-4). Zkusíme 1-2, což v oboru přirozených nezáporných čísel nelze, hodnota s číslem 2 tedy zahrnuta není, a i na dále pokračujeme s číslem 1. Nakonec zkusíme 1-1, což opět lze, hodnota s číslem 1 tedy zahrnuta je a výpočet (dekódování zahrnutých hodnot) tím končí. Zahrnuty jsou tedy hodnoty s čísly 1, 4 a 8.

[8] Binární operátor **OR** se v jazyce C# zapisuje jako jeden znak svislé čárky `|` (Ctrl+W). Jeho výsledkem je číselná hodnota vytvořená binárním součtem hodnot nalevo a napravo od tohoto operátoru. Tímto operátorem lze sice slučovat i logické hodnoty **true/false** (bity `0`/`1`), oproti logickému **OR** zapsanému dvěma čárkami `||` však nejprve vždy pro určení výsledku vyhodnotí výrazy na obou stranách. Logické **OR** však vyhodnotí nejprve hodnotu nalevo a vyjde-li tato **true**, hodnotu napravo již neřeší (to pouze pokud ta první vyjde **false**).

Takovýmto očíslováním hodnot lze tedy souhrnné hodnoty vytvářet, nicméně aby zbylý kód věděl, že má s touto funkcionalitou (souhrnnými hodnotami) počítat a reprezentovat je stále jako hodnoty daného výčtu nikoli pouze jako číslo, je třeba ještě takový výčet označit příznakem (atributem) **Flags**, který zapíšeme do hranatých závorek nad hlavičku výčtu (viz Kód 60, ř. 9). Díky tomu se program bude snažit každé číslo přetypované na typ výčtu dekódovat jako souhrnnou hodnotu (bez tohoto příznaku by překládal pouze čísla reprezentující právě jednou z hodnot výčtu).

```
 1: var vikend1 = (DnyVTydnu)96;                          // 96 = 32+64
 2: var vikend2 = (DnyVTydnu)((int)DnyVTydnu.Sobota + // 32 +
                              (int)DnyVTydnu.Nedele); // 64 = 96
 3: var vikend3 = DnyVTydnu.Sobota |   // 0100000 OR
                  DnyVTydnu.Nedele;    // 1000000
 4:                                     // 1100000 = 2^5+2^6 = 32+64=96
 5: Console.WriteLine(vikend1); // Sobota, Nedele
 6: Console.WriteLine(vikend2); // Sobota, Nedele
 7: Console.WriteLine(vikend3); // Sobota, Nedele
 8:
 9: [Flags]
10: public enum DnyVTydnu
11: {                   // 2^... 6543210 (indexy)
12:     Pondeli =  1,   // 2^0 = 0000001
13:     Utery   =  2,   // 2^1 = 0000010
14:     Streda  =  4,   // 2^2 = 0000100
15:     Ctvrtek =  8,   // 2^3 = 0001000
16:     Patek   = 16,   // 2^4 = 0010000
17:     Sobota  = 32,   // 2^5 = 0100000
18:     Nedele  = 64,   // 2^6 = 1000000
19: }
```

Kód 60 – Ukázka tří možných způsobů definice souhrnné hodnoty výčtu typu *Flags*

Kód 60 ukazuje tři možné způsoby definice souhrnné hodnoty výčtu. Jako číslo přetypované na výčet (ř. 1), jako součet číselných reprezentací jednotlivých hodnot přetypovaných zpět na výčet (ř. 2) a jako spojení přímo daných hodnot výčtu operátorem **OR**, což je zároveň i doporučený způsob použití v kódu (nicméně i předchozí varianty mohou posloužit např. při ukládání a opětovném načítání hodnot).

Výpis všech tří souhrnných hodnot (Kód 60, ř. 5–7) je tak vždy stejný. Kdyby však nebyl výčet označen příznakem **Flags** (ř. 9), pak by všechny tři výpisy souhrnné hodnoty vypsali pouze číslo 96. To by bylo způsobeno tím, že by se program nesnažil dané číslo, byť typu výčtu, dekódovat jako souhrnnou hodnotu, a pokud by nenašel právě jednu hodnotu nastavenou na dané číslo (96), pak by pouze toto číslo vrátil, byť v něm je vše potřebné pro zpětné dekódování na hodnotu souhrnnou.

Je-li určitá hodnota výčtu součástí hodnoty souhrnné, lze ověřit metodou **HasFlag** (*má příznak*, viz Kód 61, ř. 4). Tato metoda funguje se všemi výčty, ať již příznak (atribut) **Flags** mají či nikoli. Číselné reprezentace hodnot výčtu však musí i v tomto případě být tvořeny řadou mocnin čísla dvě.

```
 1: void JeVikend(DnyVTydnu den) //Metoda vypíše je-li den víkendový
 2: {
 3:     var vikend = DnyVTydnu.Sobota | DnyVTydnu.Nedele;
 4:     if (vikend.HasFlag(den)) // Obsahuje vikend i testovaný den?
 5:         Console.WriteLine("{0} je víkend", den);
 6:     else
 7:         Console.WriteLine("{0} není víkend", den);
 8: }
 9:
10: JeVikend(DnyVTydnu.Pondeli); // "Pondeli není víkend"
11: JeVikend(DnyVTydnu.Sobota);  // "Sobota je víkend"
```

Kód 61 – Ukázka použití metody *HasFlag* na výčtu z Kód 60

2.3.3 Procházení a parsování hodnot výčtu

Pokud potřebujeme nějak zpracovat či ověřit všechny hodnoty určitého výčtu, není třeba je všechny vypisovat, ale jejich kompletní seznam lze získat ze statické metody **GetValues** třídy **Enum** (viz Kód 62, ř. 1). Ten vrátí hodnoty jako jednorozměrné pole typu konkrétního výčtu (zde `DnyVTydnu[]`). K dispozici je též metoda **GetNames**, která vrací názvy těchto hodnot jako pole textových řetězců (`string[]`).

V případě, že bychom danou hodnotu získali právě jako textový řetězec obsahující název některé z hodnot, pak jej lze na tuto hodnotu výčtu převést statickou metodou **Parse** třídy **Enum** (viz Kód 62, ř. 6).

```
1:  var dny = Enum.GetValues<DnyVTydnu>();  // Všechny hodnoty výčtu
2:  foreach (DnyVTydnu den in dny) // Projdeme všechny hodnoty výčtu
3:      JeVikend(den);             // Každou z nich můžeme zpracovat
4:
5:  // Převod textu na hodnotu výčtu
6:  DnyVTydnu pondeli = Enum.Parse<DnyVTydnu>("Pondeli");
7:  JeVikend(pondeli);         // Převedenou hodnotu můžeme zpracovat
```

Kód 62 – Ukázka získání a zpracování pole všech hodnot a převodu (parsování) textu na hodnotu výčtu daného typu

Kód 62 počítá s deklarací výčtu **DnyVTydnu**, kterou obsahuje Kód 60, a metodou **JeVikend**, jež definuje Kód 61.

2.4 Jmenné prostory

Jmenné prostory (*namespace*) umožňují rozdělovat (třídit, kategorizovat) jednotlivé členy (třídy, struktury, záznamy, výčty, rozhraní…) do hierarchicky řazených skupin. Tyto skupiny se mohou pojmenovávat zcela libovolně, samozřejmě při dodržení základních pravidel pro názvy[9].

Systém těchto skupin je tedy hierarchický čili pracující se „stromovou strukturou“ (kmen stromu se dělí na větve, ty zase na další a další až po konečné listy), kterou například známe z uspořádání pevného disku počítače, kde je disk (kmen stromu, též kořenová složka či *root*), složky a podsložky (větve stromu) a soubory (listy). Podobně tak můžeme členit i strukturu členů ve svých projektech. Tato struktura přitom nemusí odpovídat umístění jednotlivých souborů s kódem do složek projektu a řešení (jeden jmenný prostor může přesahovat i přes různé projekty/knihovny), byť bývá dobrým zvykem tyto dva způsoby uspořádání držet v souladu.

Jelikož podobné dělení mají nejen systémové třídy (v kořenovém jmenném prostoru **System**), ale i knihovny třetích stran, bývá dobrým zvykem hned v názvu hlavního názvu (rootu) uvést název firmy, či přezdívku autora, který daný kód vytváří, aby při jeho používání nedocházelo k záměně s členy jiných autorů. Druhou dělící větví by měl být název produktu, pro který je kód vytvářen (u rozšířených open source projektů se mnohdy začíná až tímto názvem), dále pak název projektu (viz kap. 2.5) a pak jeho jednotlivé části (např. stránky, komponenty, datové třídy, funkční třídy apod.).

[9] Názvy veškerých členů a jejich složek v C# nesmí obsahovat mezery, pomlčky či spojovníky (mínus), čárky, tečky i jiné oddělovací znaky atd. Je dobré vyhýbat se i národnostním znakům (mimo základní anglickou abecedu), tedy v našem případě veškerým písmenům s diakritickými znaménky. Prvním znakem názvu také nesmí být číslo. Zbytek názvu pak může obsahovat malá a velká písmena, čísla či podtržítka.

Hlavním účelem je tedy jednak vnést do rozsáhlejších projektů pořádek a vnitřní uspořádání členů a za druhé vyřešit případy, kdy by se např. dvě třídy jmenovaly stejně. To je samozřejmě v rámci jednoho jmenného prostoru nepřípustné, jelikož by kompilátor nevěděl, kterou z nich chcete použít, takže si takové případy hlídá, nicméně, pokud se dvě stejnojmenné třídy budou nacházet v různých jmenných prostorech, pak lze vždy jednoznačně určit, o kterou z nich se jedná.

Jmenný prostor se pro danou skupinu členů definuje klíčovým slovem **namespace** za kterým následuje jeho kompletní název, tj. název rootu a do hloubky zadanou adresou jeho podskupin až po tu, do které mají být členové zařazeni (podobně jako adresa souboru na disku, např. C:\Windows\System32). Zápis je tedy veden z leva (od rootu) doprava (po skupinu nejnižší úrovně) a jednotlivé části jsou odděleny tečkou.

Platnost jmenného prostoru lze následně vymezit složenými závorkami (viz Kód 63, vlevo), nebo od verze .NET 6, jen tuto definici uvést hned pod direktivy **using** a zakončit středníkem, čímž se jeho platnost automaticky promítne do celého souboru (pro všechny nevnořené členy) a ušetří se jedno odsazení v kódu (viz Kód 63, vpravo).

```
1: namespace Firma.Produkt.Projekt1
2: {
3:     public class TridaA { }
4:     public class TridaB { }
5: }
```

```
namespace Firma.Produkt.Projekt2;

public class TridaA { }
public class TridaB { }
```

Kód 63 – Ukázka dvou variant zařazení tříd do jmenného prostoru

Pokud pak chceme používat např. třídu z nějakého jmenného prostoru, je třeba splnit jednu z následujících podmínek:

- nacházet se ve stejném jmenném prostoru (např. `namespace Firma.Produkt.Projekt1`),
- uvést název tohoto jmenného prostoru na začátku souboru do seznamu **using**, tj. ke jmenným prostorům do nichž zařazené členy chceme v následujícím kódu používat (např. `using Firma.Produkt.Projekt1;`),
- někde (v jakémkoli „.cs“ souboru) v projektu nastavit tento jmenný prostor jako globálně používaný (např. `global using Firma.Produkt.Projekt1;`, funguje až od .NET 6),
- v souboru projektu (*.csproj*) jej zapsat mezi v projektu globálně používané jmenné prostory do vlastní skupiny (**ItemGroup**) jako jeho XML podelementy `<Using Include="Firma.Produkt.Projekt1" />`,
- při každém uvedení názvu třídy zapsat její celý název, tj. včetně jmenného prostoru (např. `Firma.Produkt.Projekt1.TridaA`).

Pokud bychom do seznamu používaných tříd (**using**) v témže souboru uvedli dva takové, které obsahují stejně pojmenovanou třídu, a tu chtěli v kódu používat, pak máme následující možnosti, jak rozlišit, kterou z nich právě chceme použít:

- v každém zápisu názvu třídy v kódu tento název rozšířit buď na jeho plný název (např. `Firma.Produkt.Projekt1.TridaA`), nebo jen jeho podčást, která už bude jedinečná (např. `Projekt1.TridaA`, nebo `Projekt2.TridaA`),
- v seznamu **using** si pro takovou třídu (popř. třeba i pro obě) definovat její alias (viz Kód 64, ř. 4 a použití na ř. 12), což lze používat i při jiných příležitostech (např. pro zkrácení dlouhého názvu třídy).

```
1:  using System;                          // Používané jmenné prostory
2:  using Firma.Produkt.Projekt1;
3:  using Firma.Produkt.Projekt2;          // odkaz nejen na jmenný prostor, ale až na konkrétní třídu
4:  using TA1 = Firma.Produkt.Projekt1.TridaA;  // Alias názvu třídy
5:
6:  namespace Firma.Produkt.Projekt3;  // Jmenný prostor souboru
7:
8:  public class Program
9:  {
10:     public static void Main()
11:     {
12:         var x = new TA1();         // Použití aliasu třídy
13:     }
14: }
```

Kód 64 – Ukázka definice a použití aliasu třídy

V rámci seznamu používaných tříd (**using**) lze též uvést určitou třídu (nebo i více tříd, či jiných členů) tak, aby veškeré její statické složky byly dostupné v kódu přímo pod jejich názvy, bez nutnosti před nimi uvádět i název této třídy. Stačí za klíčové slovo **using** přidat **static** a poté vypsat celý název dané třídy (viz Kód 65, ř. 2 a použití na ř. 6).

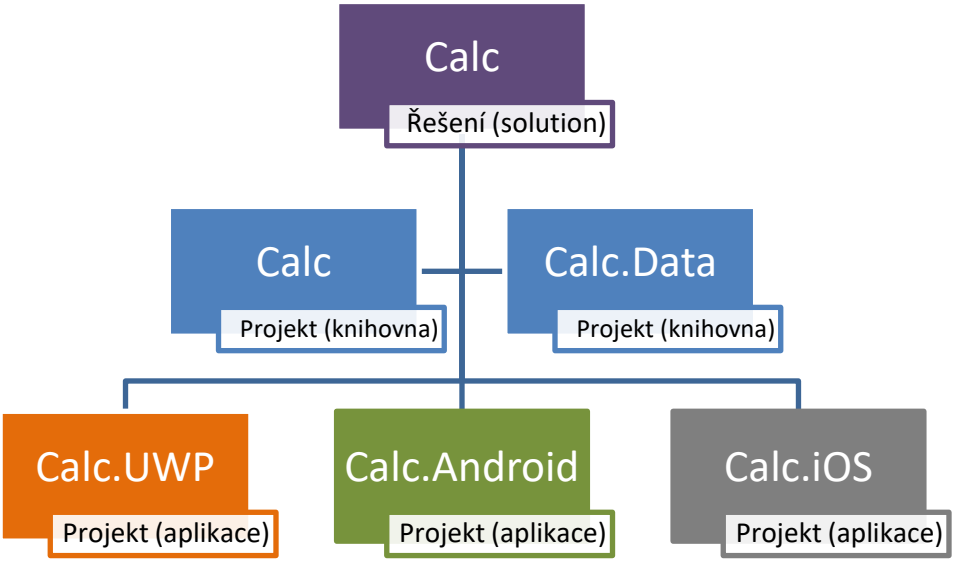
```
1:  using System;
2:  using static System.Console; // Zpřístupnění stat. složek třídy
3:  global using static System.Math; // Zpříst. sl. v celém projektu
4:
5:  Console.WriteLine("Ahoj");   // Funguje klasicky toto
6:  WriteLine("Ahoj");           // ale zároveň také toto
7:  WriteLine(Sqrt(3*3));        // metody z Math lze použít kdekoli
```

Kód 65 – Ukázka zpřístupnění statických složek jiné třídy přímo pod jejich názvy

Modifikátor **static** a **global** lze vzájemně kombinovat, díky čemuž je takovýmto jediným příkazem možné zpřístupnit statické složky třídy v celém projektu (viz Kód 65, ř. 3).

2.5 Projekty a řešení

Kód některých programů (aplikací) může být dosti rozsáhlý a obsahovat desítky i stovky souborů. Zároveň některé jeho části mohou být použitelné ve více různých aplikacích. Naštěstí je možné je kromě klasických složek či jmenných prostorů třídit ještě do projektů (viz Obr. 5).



Obr. 5 – Struktura řešení a projektů programu Kalkulačka (Calc) typu Xamarin.Forms

Celý program je obvykle zastřešen do jednoho **řešení** (*solution*, soubor s koncovkou „.sln“), což je textový soubor, který je ve Visual Studiu je zobrazen a editován ve stromové struktuře v plovoucím okně Průzkumník řešení (*Solution Explorer*). Řešení obsahuje především seznam, pro C# sou-

bory s koncovkou „.csproj“), tzn. relativní (popř. absolutní) cesty k souborům s deklarací v řešení zahrnutých projektů. Může také obsahovat virtuální složky, do kterých lze tyto projekty vizuálně roztřídit.

Projekty (*project*) mohou být různého typu, např. podle platformy pro kterou jsou určeny (Windows, Android, iOS, Linux, Azure, Xbox…), jazyk, ve kterém jsou psány (C#, Visual Basic, F#, C++, JavaScript, Python…) nebo typu výstupu, který mají generovat (konzole, desktop, web, IoT, plugin, hra, strojové učení, servisní služba, cloud, testy…). Nejzákladnější rozlišení výstupu však je na knihovnu a aplikaci.

Z projektů typu **knihovna** obvykle vznikne soubor s koncovkou „.dll“ (na jiných platformách může být odlišná), jejíž prvky pak hlavní program pouze používá, ale sama o sobě spustit či nějak využít nelze. Tyto knihovny jsou nicméně samostatné jednotky, které může používat více různých aplikací. Někdy je dokonce cílem celého programu pouze vytvoření takovéto knihovny, aniž by předem bylo známo, které aplikace ji budou používat. Může se například zkompilovat do NuGet balíčku, jehož vývoj a distribuce je zcela samostatná, a v případě jeho zveřejnění např. na nuget.org jej může používat v podstatě kdokoli (jejich vyhledání, instalaci i aktualizaci lze řešit v rámci Visual Studia vizuálně nebo pomocí příkazů přes jeho konzoli).

Projekty **aplikačního** typu pak generují výstup přímo použitelný bez nutnosti nějakého dalšího programování. Může jím být spustitelný program (soubor s koncovkou „.exe“), aplikační balíček pro Microsoft Store, Google Play či App Store, webová aplikace, doplněk do Microsoft Office, program pro zařízení internet věcí (IoT, „chytré krabičky“) apod. Tyto „aplikační“ projekty často používají knihovny či NuGet balíčky jiných projektů, ať již jsou přidány jako zkompilované (např. soubory dll), nebo jako projekty se zdro-

jovým kódem přidané do jednoho řešení. V řešení pak může být více projektů obou typů, jedna aplikace a více knihoven bývá typická např. u webových aplikací, a jedna knihovna s více spustitelnými projekty se pak třeba používá u multiplatformních aplikací (např. Xamarin.Forms, viz Obr. 5).

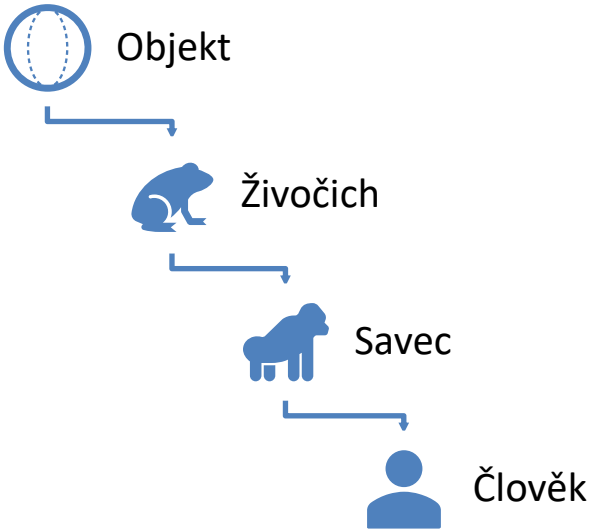
Z pohledu především reflexe se pak projektům také říká **sestavení** (*assembly*), čímž je označován ucelený prvek (soubor, např. dll knihovna) tříd a dalších členů. Pouze ve speciálních případech může sestavení obsahovat i více projektů propojených dohromady, např. u tzv. sdílených (*shared*) projektů.

V závěru zmiňme ještě pojem **repozitář** (*repository*). Tím je míněno úložiště (lokálně např. složka na disku), do které je organizovaným způsobem uloženo vše týkající se daného programu (tj. řešení, projekty atd., včetně např. dalších zdrojů, jako jsou obrázky, ikony, dokumentace apod., byť u větších projektů mohou být některé části řešeny odděleně v samostatných repozitářích). Takové složce pak bývá obvykle přiřazen nějaký nástroj pro správu verzí (např. Git), v cloudové podobě pak bývá synchronizována s nějakým online repozitářem (např. na GitHub), kde je řešena i souběžná práce na kódech více osobami, včetně souběžného vývoje jejich různých větví.

3 Dědičnost

Dědičnost (*inheritance*) umožňuje třídám (a záznamům) odvozovat jednu ze druhé tak, aby převzala („zdědila“) veškeré její dostupné složky. Nejedná se tedy o dědičnost genetickou (kdy potomek zdědí po rodičích nějaké charakteristické rysy) ani právní (kdy potomek dědí majetek), ale o dědičnost programátorskou. V tomto případě „potomek“ (odvozená třída nebo též **podtřída**) přebírá od „rodiče“ (nadřazené třídy nebo též **nadtřídy** či předka) veškeré její veřejné (**public**, popř. v témže projektu i **internal**), a pro vlastní vnitřní použití také chráněné (**protected**), složky (dostupnost podrobněji viz Tab. 1 na str. 14, sloupec AT3 a BT2).

Nap. lze vytvořit dědičnost tříd z hlediska biologického, kdy by ze základní třídy **Object** (což je prapředkem všech tříd v C#) byla odvozena podtřída *živočich*, z ní pak podtřída *savec* a z ní ještě třeba podtřída *člověk* (viz Obr. 6).



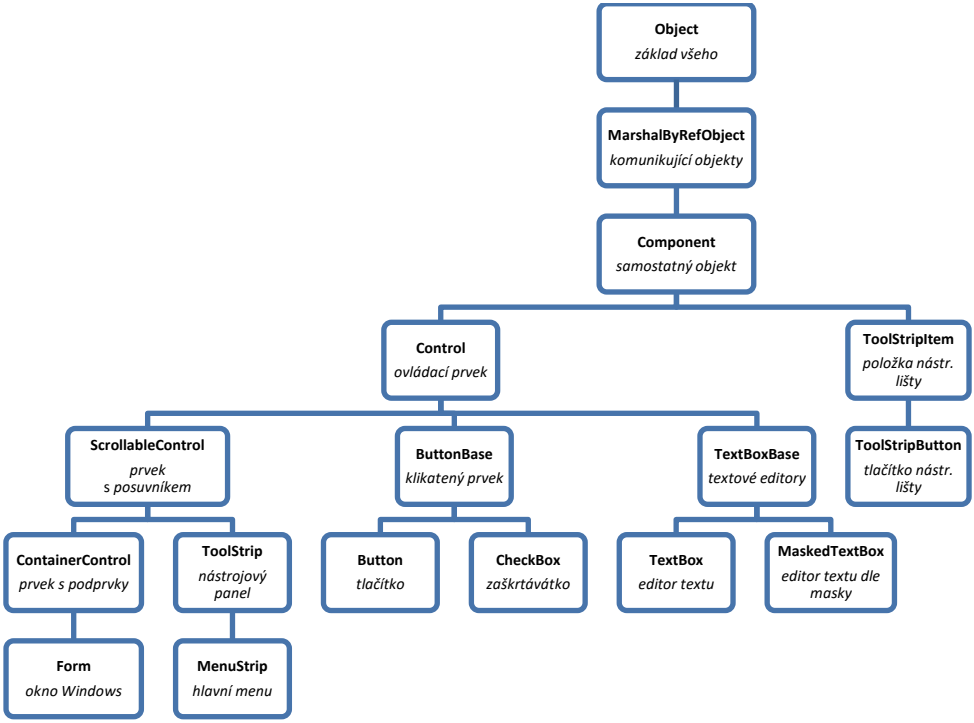
Obr. 6 – Ilustrační ukázka možné dědičnosti tříd na příkladu biologické klasifikace

V C# kódu se pak to, že jedna třída dědí z druhé zapisuje do hlavičky podtřídy tak, že se za její název přidá znak dvojtečky a za ní název nadtřídy, ze které se tato odvozuje (ze které dědí, viz Kód 66). V případě dědičnosti ze základní třídy **Object** není nezbytné toto do deklarace třídy zapisovat, jelikož ta je výchozí předek pro každou třídu, u které není explicitně uvedena jiná nadtřída (viz Kód 66, ř. 1).

```
1:  public class Zivocich { }   // Automaticky dědí ze třídy Object
2:  public class Savec  : Zivocich { }   // Savec dědí ze Zivocich
3:  public class Clovek : Savec { }      // Clovek dědí od Savec
```

Kód 66 – Ukázka dědičnosti tříd

Tato ukázka na příkladu biologické struktury vztahů podtříd a nadtříd je sice dobře pochopitelná, ale v praxi se spíše používají třídy, které touto dědičností postupně doplňují své vlastnosti, metody a události tak, aby je mohly podtřídy rovnou používat, a sami je pouze rozšiřovat o další složky. Zároveň takové třídy obvykle neslouží ke katalogizaci druhů (od toho jsou tu databáze), ale především k řešení konkrétní úlohy. Například Obr. 7 ukazuje dědičné vztahy (hierarchii dědičnosti) některých reálných tříd z aplikace typu Windows Forms.



Obr. 7 – Ukázka vztahů dědičnosti některých reálných tříd z aplikace typu Windows Forms

3.1 Polymorfismus

Polymorfismus (*polymorphism*) je jeden ze zásadních prvků OOP dědičnosti, díky níž může třída potomka (podtřída) přepsat (nahradit) či doplnit (rozšířit) kód předka (nadtřídy) v konkrétní složce. Tou je nejčastěji metoda (viz kap. 1.2.3), ale může se to týkat i vlastnosti (viz kap. 1.2.2), resp. jejích **get** a **set** bloku kódu.

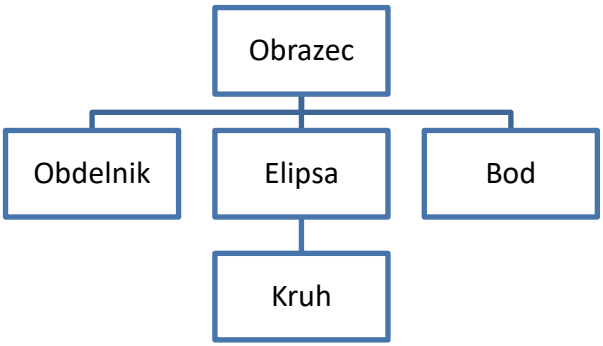
Aby předek tuto funkcionalitu svým potomkům zpřístupnil, musí danou složku označit klíčovým slovem **virtual** (virtuální, *přepsatelná*, viz Kód 67, ř. 2) a potomek pro znamení, že tohoto mechanismu používá použije klíčové slovo **override** (*přepisující*, ř. 5, 8, 11 a 14). Samozřejmostí pak je, že

dostupnost (viz kap. 1.1.3) takto přepisované složky musí být alespoň **protected** (nebo vyšší), resp. nesmí být **private**, jinak by k ní podtřída neměla přístup.

```
1:  public class Obrazec {                  // Základní třída
2:      public virtual void Kresli() { }    // Virtuální metoda
3:  }
4:  public class Obdelnik : Obrazec {       // Podtřída 1. úrovně
5:      public override void Kresli() { }
6:  }
7:  public class Elipsa : Obrazec {         // Podtřída 1. úrovně
8:      public override void Kresli() { }
9:  }
10: public class Kruh : Elipsa {            // Podtřída 2. úrovně
11:     public override void Kresli() { }
12: }
13: public class Bod : Obrazec {            // Podtřída 1. úrovně
14:     public override sealed void Kresli() { } // Konec polymorf.
15: }
```

Kód 67 – Ukázka polymorfismu metody *Kresli*

Kód 67 ukazuje použití polymorfismu na struktuře tříd, které jsou znázorněny na Obr. 8.



Obr. 8 – Ukázka hierarchie dědičnosti tříd

Tyto třídy polymorficky používají metodu **Kresli**, která je v základní třídě **Obrazec** označena jako virtuální (**virtual**) a v jejích podtřídách pak označena jako přepisující (**override**). Tento mechanismus pak funguje i v případných dalších podtřídách (např. u třídy **Kruh**, viz Kód 67 na ř. 10–12, která je potomkem třídy **Elipsa**, ř. 7–9).

3.1.1 Speciální případy

Zamezení dalšího přepisování podtřídami

Pokud je však z nějakého důvodu žádoucí dalším případným podtřídám přepisování této metody zakázat, stačí do její hlavičky přidat klíčové slovo **sealed** (viz Kód 67, ř. 14). Stejným klíčovým slovem lze mimochodem také označit i celá třída (např. `public sealed class Bod : Obrazec`), čímž se zcela zamezí vytváření jejich dalších podtříd.

Překrytí metody bez polymorfismu

V případě že u přepisované (v předkovi virtuální) složky podtřídy není uvedeno klíčové slovo **override**, popř. použití složky se zcela identickou hlavičkou (u metody název a počet i typy vstupních parametrů) i když tato virtuální není, dojde pouze k překrytí dané složky. V takovém případě zde nebude fungovat princip polymorfismu (viz násl. kap.) a programovací prostředí (Visual Studio) zobrazí varování upozorňující na tuto skutečnost.

Je-li takovýto krok záměrný, pak by mělo být do hlavičky metody přidáno klíčové slovo **new** (např. `public new void Kresli() { }` ve třídě **Kruh**), na znamení toho, že si je programátor plně vědom toho, co dělá, čímž zároveň i zruší toto varování. Tento způsob jde ale spíše proti principům OOP a používá se pouze vyjímečně.

3.1.2 Princip polymorfismu

Metoda potomka tedy může v přepisované metodě nahradit kód metody předka svým vlastním (viz Kód 68). V rámci něho však může (ř. 9), ale nemusí (ř. 16–19), použít i předkův kód a ten tak v podstatě jen doplnit svým vlastním (ř. 10–13). Učiní se tak zavoláním „bázové“ metody pomocí klíčového slova **base**, které odkazuje na předka této třídy, a zavoláním jeho příslušné metody (např. `base.Kresli();` na ř. 9).

```
 1: public class Obrazec {              // Předek
 2:     public int X, Y, Sirka, Vyska; // Určení polohy a rozměrů
 3:     public PascalCanvas c;  // Plocha, na kterou se bude kreslit
 4:     public virtual void Kresli() {
 5:         c.MoveTo(X, Y);              // Přesun na [X, Y]
 6: }   }
 7: public class Obdelnik : Obrazec {   // 1. potomek
 8:     public override void Kresli() {
 9:         base.Kresli();       // Zavolání kódu metody předka
10:         c.LineRel(Sirka, 0); // Čára a posun z akt.pos. o (X, Y)
11:         c.LineRel(0, Vyska);
12:         c.LineRel(-Sirka, 0);
13:         c.LineRel(0, -Vyska);
14: }   }
15: public class Elipsa : Obrazec {     // 2. potomek
16:     public override void Kresli() { // Nevolá kód předka (base)
17:         c.MoveTo(X + Sirka/2, Y + Vyska/2);    // Přesun na střed
18:         c.Ellipse(rX = Sirka/2, rY = Vyska/2); // Dva poloměry
19: }   }
```

Kód 68 – Ilustrativní ukázka polymorfismu s volitelným použitím složek předka

Kód 68 tedy ukazuje možnost volitelného použití kódu přepisované metody předka **Kresli**. Předek (**Obrazec**) zde v metodě **Kresli** pouze umístí kreslící ukazatel do levého horního rohu plochy vymezené pro obrazec (ř. 5). To-

hoto kódu využije **Obdelnik** (ř. 9), jež z tohoto bodu zahájí vykreslování obdélníka. **Elipsa**, která ale místo rohu potřebuje začít na středu plochy, kód předka nepoužije a přemístění provede vlastním příkazem (ř. 17).[10]

Jak ukazuje Kód 69, tak to může být i obráceně, tedy že kód předka používá kód potomka (ř. 12). To je zároveň i výchozí chování, takže zde již není třeba žádného klíčového slova jako bylo **base**, a vše funguje automaticky.

```
 1: Obrazec o = new Obdelnik() { X = 5, Y = 8 };
 2: Obrazec e = new Elipsa()   { X = 6, Y = 7 };
 3:
 4: Console.WriteLine(o);   // "Obdélník [5, 8]"
 5: Console.WriteLine(e);   // "Ovál [6, 7]"
 6:
 7: public class Obrazec {
 8:     public int X { get; set; }
 9:     public int Y { get; set; }
10:     protected virtual string Nazev { get; }
11:     // Přepis textového vyjádření objektu s názvem jeho typu
12:     public override string ToString() => $"{Nazev} [{X}, {Y}]";
13: }
14: public class Obdelnik : Obrazec { // Dodá název s diakritikou
15:     protected override string Nazev => "Obdélník";
16: }
17: public class Elipsa : Obrazec {
18:     protected override string Nazev => "Ovál";
19: }
```

Kód 69 – Ukázka polymorfismu vlastnosti *Nazev*, kterou používá i předek

[10] V této ukázce kódu (Kód 68) jsou pro zjednodušení použity příkazy (*MoveTo*, *LineRel* a *Ellipse*) inspirované jazykem Pascal (viz ř. 3 a fiktivní *PascalCanvas*), neboť v C# jsou tyto příkazy odlišné pro různé technologie vykreslování, a nezbytná je i příprava plochy, což by zde snižovalo názornost uvedeného příkladu.

Kód 69 definuje ve třídě **Obrazec** přepisovatelnou (**virtual**) vlastnost **Nazev** pouze pro čtení (má jen **get**, viz ř. 10). Tuto vlastnost pak přepisují (**override**) potomci třídy **Obdelnik** a **Elipsa**, které v této vlastnosti vracejí konstantu názvu typu obrazce, který vykreslují („Obdélník“, ř. 15 a „Ovál“, ř. 18). Bázová třída **Obdelnik** pak s hodnotou vracenou vlastností **Nazev** pracuje tak, že ji dosadí do textové reprezentace objektu (ř. 12) odvozeného z této třídy či jejích potomků. Pokud jsou tedy vytvořeny instance těchto podtříd (ř. 1–2), pak při jejich zobrazení jako textových řetězců (ř. 4–5) každý dosadí do názvu ten svůj, a jejich předek je spojí do výstupního řetězce spolu s hodnotami dalších vlastností v metodě **ToString** přepsané od třídy **Object** (ř. 12).

Při hlubší hierarchii tříd se pomocí klíčového slova **base** odkazujeme na nejbližšího předka, který danou metodu implementoval (ať již prvotně jako **virtual**, nebo přepsáním přes **override**).

Zpracování instance potomka jako jeho předka

Podstatu celého principu vystihují hned první dva řádky z této i následující ukázky (Kód 70, ř. 1 a 2), kdy je do proměnné **o** typu **Obrazec** vytvořena instance třídy **Obdelnik** (popř. **Elipsa** do **e**). Proměnná **o** tak má bez dalšího přetypování (např. `(Obdelnik)o`) dostupné pouze složky ze třídy **Obrazec** (popř. jejích nadtříd, tj. minimálně **Object**), nikoli však již případně další složky, které by přidávala podtřída **Obdelnik** (popř. třídy nacházející se v hierarchii dědičnosti mezi nimi). Nicméně u složek označených jako virtuální je i tak použit kód přepsaný skutečnou instancí třídy v této proměnné uložené (uvnitř třídy v celé hierarchii je pak tento přístup k objektu buď přímý nebo přes klíčové slovo **this**).

V uvedeném příkladě lze samozřejmě jako typ proměnné použít stejnou třídu, ze které je vytvářena její instance (`Obdelnik o = new Obdelnik();`,

nebo použít klíčové slovo **var**, které by tam tento typ dosadilo při kompilaci), čímž navíc získáme přímý přístup i k případným dalším složkám, které by tato třída přidávala, a zároveň s ní i nadále můžeme pracovat obecněji jako s obrazcem. Například pokud vytvoříme metodu, která bude univerzálně vykonávat nějakou činnost s libovolným obrazcem, pak nám možnost použití nadtřídy **Obrazec** jako typ vstupního parametru ušetří nutnost vytvářet pro každý podtyp obrazce zvláštní metodu s tímtéž kódem (viz Kód 70).

```
 1: Obdelnik o = new Obdelnik();
 2: Elipsa e = new Elipsa();
 3:
 4: ResetujObrazec(o);   // Podtřídy z Obrazec jsou také Obrazec
 5: ResetujObrazec(e);
 6:
 7: public static void ResetujObrazec(Obrazec obr) {
 8:     obr.X = 0;
 9:     obr.Y = 0;
10:     obr.Sirka = 120;
11:     obr.Vyska = 90;
12: }
```

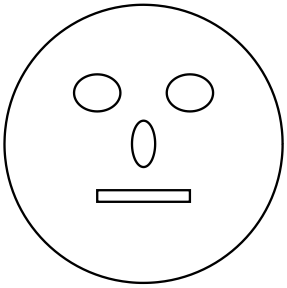
Kód 70 – Ukázka univerzální metody pracující se všemi typy obrazců

Díky stejnému principu pak lze podobným způsobem zpracovávat nejen jednotlivé proměnné, ale prostřednictvím cyklů i jejich celá pole či seznamy (viz Kód 71).

```
 1: List<Obrazec> vykres = new List<Obrazec>(); // Prázdný výkres
 2: vykres.Add(new Elipsa(  100, 100, 60, 60)); // Obličej
 3: vykres.Add(new Elipsa(  100, 100, 5,  10)); // Nos
 4: vykres.Add(new Elipsa(   80,  80, 10, 16)); // Levé oko
 5: vykres.Add(new Elipsa(  120,  80, 10, 16)); // Pravé oko
 6: vykres.Add(new Obdelnik( 80, 120, 40,  5)); // Pusa
 7:
 8: NakresliVykres(vykres); // Nakreslí výkres
 9:
10: void NakresliVykres(List<Obrazec> obrazce)
11: {
12:     foreach (Obrazec obr in obrazce) // Projde všechny obrazce
13:         obr.Kresli();                // a postupně je vykreslí
14: }
```

Kód 71 – Ukázka hromadného zpracování instancí různých podtypů

Kód 71 ukazuje metodu **NakresliVykres** (ř. 10–14), která na vstupu přijme seznam obrazců (instancí třídy **Obrazec** či jejich podtříd), cyklicky jej projde (ř. 12) a pro každý z nich zavolá metodu **Kresli** (ř. 13), jež použije kód podle skutečného typu dané instance. Výsledek pak může vypadat např. jako na Obr. 9.



Obr. 9 – Ukázka možného výstupu výkresu z Kód 71 a jeho pokračování v Kód 72

Dědičnost u dalších členů

Kód 71 také použil zadání parametrů obrazce přímo do konstruktorů jeho podtříd (ř. 2–6). Tuto možnost lze samozřejmě pomocí parametrického konstruktoru (viz str. 34) každé třídě nastavit, nicméně pro zkrácení zápisu lze použít i záznamy (**record**, viz kap. 2.2), které z těchto vstupních parametrů konstruktoru při kompilaci vytvoří vlastnosti automaticky (viz Kód 72).

```
1:  public record Obrazec(int X, int Y, int Sirka, int Vyska) {
2:      public virtual void Kresli() { }
3:  }
4:
5:  public record Obdelnik(int X, int Y, int Sirka, int Vyska)
6:      : Obrazec(X, Y, Sirka, Vyska) { // Předání hodn. kon. předka
7:      public override void Kresli() { }
8:  }
9:
10: public record Elipsa(int X, int Y, int Sirka, int Vyska)
11:     : Obrazec(X, Y, Sirka, Vyska) {
12:     public override void Kresli() { }
13: }
```

Kód 72 – Ukázka dědičnosti a polymorfismu u záznamů (*record*) s vlastnostmi vytvořenými přes konstruktor předka a načítanými i v jeho podtřídách

U tříd pak mají konstruktory speciální způsob zápisu při volání kódu konstruktoru jejich předka. Ten se nevolá až v kódu konstruktoru potomka, ale zapíše se podobně jako dědičnost samotná, tedy rovnou za dvojtečku v hlavičce konstruktoru potomka, přidáním klíčového slova **base** jako zástupce názvu „metody“ konstruktoru potomka, a do kulaté závorky se předají jeho vstupní parametry (viz Kód 73).

```
1:  public class Obrazec {
2:      public int X { get; init; }      // Vlastnost
3:      public Obrazec(int x) { X = x; } // Konstruktor předka
4:  }
5:
6:  public class Obdelnik : Obrazec {
7:      public Obdelnik(int x) : base(x) { } // Konstruktor potomka
8:  }
```

Kód 73 – Ukázka polymorfismu u konstruktorů tříd

Dědičnost je tedy možná jak mezi třídami (kap. 1), tak mezi záznamy (kap. 2.2), nelze je však mezi sebou křížit (třída nemůže dědit ze záznamu ani naopak). U struktur (kap. 2.1) pak dědičnost nelze používat vůbec.

3.2 Abstraktní třídy

Abstraktní (*abstract*) třídy umožňují definovat abstraktní složky (metody a vlastnosti), které na rozdíl od složek virtuálních, jež potomci třídy přepsat mohli, tak tyto potomci abstraktních tříd přepsat musí povinně. Tento fakt pak přináší několik důležitých pravidel:

- Jelikož je jisté, že potomci složce (obvykle metodě) určitě dodají implementaci (kód), tak tato abstraktní složka žádný kód neobsahuje.
- Protože abstraktní složky abstraktní třídy sami kód neobsahují, protože se plně spoléhají až na své potomky, tak z abstraktní třídy nelze vytvářet instance (to až z jejích neabstraktních potomků).
- Aby nemohli vzniknout instance bez kódu, tak zároveň abstraktní složky může obsahovat pouze abstraktní třída.

Pokud je v hierarchii dědičnosti mezi abstraktní třídou a podtřídou nějaká neabstraktní mezitřída, pak její potomci již nejsou povinni (byť stále mohou) původně abstraktní složky implementovat, neboť jejich kód již povinně dodala tato mezitřída.

```
1: public abstract class Obrazec {    // Abstraktní třída
2:     public abstract void Kresli(); // Abstraktní metoda nemá kód
3: }
4: public class Obdelnik : Obrazec {
5:     public override void Kresli() { } // Musí přepsat
6: }
7: public class Elipsa : Obrazec {
8:     public override void Kresli() { } // Musí přepsat
9: }
10: public class Kruh : Elipsa {
11:     public override void Kresli() { } // Může přepisovat dále,
12: }
13: public class Ctverec : Obdelnik {   } // ale nemusí
```

Kód 74 – Ukázka dědění z abstraktní třídy

Abstraktní třídy mohou krom abstraktních složek (povinně přepisovatelných) obsahovat i složky virtuální (nepovinně přepisovatelné), i zcela nepolymorfní, jež pak jejich potomci pouze přebírají bez možnosti jejich úprav.

3.3 Rozhraní

Rozhraní (*interface*) posouvá možnosti abstrakce ještě o něco dále než abstraktní třídy. Rozhraní si lze totiž zjednodušeně představit jako abstraktní třídu, jejíž veškeré složky jsou abstraktní, tedy bez kódu, a podtřída je musí povinně implementovat. Jsou tu ale určité rozdíly.

Předně z rozhraní třída nedědí (už jen proto, že od něho žádný kód nezíská), ale tzv. implementuje jej (veškerý kód musí dodat sama). Je to jako když se

firma zaváže dodržovat určité normy – prvotně tím nic nezíská (žádný kód), přinese jí to pouze starosti navíc (musí implementovat příslušné složky, bez některýchž by se třeba i obešla), ovšem díky tomu může rozšířit přístup na nové trhy. Například pokud třída implementuje rozhraní **IEnumerable**, lze její data procházet pomocí cyklu **foreach**.

Rozhraní bývá zvykem pojmenovávat tak, že prvním znakem názvu je velké písmeno **I** (jako *Interface*, např. **IList**), aby jej na první pohled bylo poznat od ostatních členů. Výchozí dostupnost složek, které rozhraní předepisuje, je veřejná (**public**), takže není nezbytné takto označovat veškeré řádky definice rozhraní, ovšem lze jim zajistit také dostupnost chráněnou či interní (popř. jejich kombinaci, viz kap. 1.1.3), a ty už je třeba uvést explicitně. Dostupnost typu soukromá (**private**) pak rozhraní nepodporuje, neboť slibuje pouze dostupné složky, a takové, ke kterým se zvenčí nikdo nedostane, nemá důvod řešit.

```
 1: public interface IObrazec {     // Rozhraní
 2:     int X { get; set; }
 3:     int Y { get; set; }
 4:     void Kresli();              // Neobsahuje žádný kód
 5: }
 6: public class Obdelnik : IObrazec {
 7:     public int X { get; set; }  // Vše musí implementovat
 8:     public int Y { get; set; }
 9:     public void Kresli() { }
10: }
11: public class Elipsa : IObrazec {
12:     public int X { get; set; }  // A další třída zase znovu
13:     public int Y { get; set; }
14:     public void Kresli() { }
15: }
```

Kód 75 – Ukázka rozhraní a jeho implementace

3.3.1 Důvody pro používání rozhraní

Proč tedy místo dědičnosti a abstraktních tříd, které mohou obsahovat alespoň základní kód a jím nějak přispět k celku (např. v Kód 75 by abstraktní třída mohla implementovat alespoň vlastnosti **X** a **Y** a podtřídy by je již nemuseli znovu zmiňovat) vůbec používat rozhraní, jež pouze předepisuje, co třídy musí? Důvodů je tu několik, přičemž jedním z hlavních je, že v C# (na rozdíl od některých jiných jazyků), může třída dědit pouze z jedné třídy (mít právě jednoho předka), zatímco rozhraní může implementovat libovolné množství. Použitelnost třídy tak lze postupně navyšovat dle potřeby implementací jednotlivých rozhraní, která jsou pro jejich podporu vyžadována.

Dalším důvodem je právě vysoká míra abstrakce, jež zaručuje absenci kódu v definici rozhraní. Příslib existence určitých složek, které třída implementací rozhraní dává, přičemž veškerý kód je pouze na její straně, umožňuje snazší provázání různých komponent (např. NuGet balíčků), modulů i systémů, zvláště podílí-li se na jejím vývoji více nezávislých programátorů.[11]

Krom toho není problém pro libovolné rozhraní připravit i abstraktní třídu, která by jej implementovala, a dodala základní kód složkám, jež by byly u všech tříd implementující toto rozhraní stejné (např. u Kód 75 vlastnosti X a Y). Složky, jež bývají stejné *téměř* vždy by pak byly virtuální a ty zcela individuální přidány jako abstraktní. Pokud v jednom projektu plánujeme rozhraní implementovat do většího počtu tříd, může tento přístup přinést značné úspory v kódu a zároveň ponechat výhody z používání rozhraní.

[11] Například služba WCF (*Windows Communication Foundation*) pro komunikaci mezi internetovým serverem a klientskými aplikacemi třetích stran, používá rozhraní, jehož definici poskytuje ve veřejně dostupném popisném XML souboru (WSDL). Autoři klientských aplikací si pak pouze na základě tohoto popisu nechají Visual Studiem vygenerovat toto rozhraní do své aplikace, a třídy v ní, které jej implementují, mohou používat serverové metody, stejně jednoduše, jako kdyby byly součástí kódu této lokální aplikace.

Rozhraní však, vzhledem k možnosti, že jich třída může implementovat libovolný počet, bývají často maximálně rozděleny tak, aby obsluhovaly jedinou konkrétní základní funkčnost, na což jim obvykle stačí jedna metoda. Např. rozhraní **IDisposable**, které lze používat v příkazech dočasného používání **using** (např. `using (var file = File.Open(...))`) předepisuje jedinou metodu **Dispose**, jež má uvolnit používané či připojené zdroje (např. uzavřít otevřený přístup do souboru). V takových případech je pak abstraktní mezi-třída spíše kontraproduktivní.

Rozhraní samozřejmě mohou implementovat i záznamy (viz kap. 2.2).

3.3.2 Dědičnost mezi rozhraními

Rozhraní mezi sebou jako třídy dědit nemohou, jelikož ani nemají co (neobsahují žádný kód). Disponují však podobným mechanismem možnosti přebírání svých složek mezi sebou. Pokud tedy do jednoho rozhraní chceme zanést prvky z jiného, či z více rozhraní, aniž bychom je tam chtěli vypisovat, stačí do hlavičky tohoto souhrnného rozhraní za jeho název přidat (stejně jako při zápisu dědičnosti tříd) název rozhraní (jednoho či více oddělených čárkou), jehož složky chceme automaticky používat i zde (viz Kód 76, ř. 9).

```
 1: public interface ISaver {
 2:     void Save(string path); // Uloží svá data do souboru
 3: }
 4:
 5: public interface ILoader {
 6:     void Load(string path); // Načte si data ze souboru
 7: }
 8:
 9: public interface IDataAccessor : ISaver, ILoader { // Obojí
10:     void Delete(string path); // Navíc bude i umět data mazat
11: }
```

Kód 76 – Ukázka „dědičnosti“ rozhraní

Tento způsob zápisu, místo opětovného přidání složek do nového rozhraní (tj. zde v Kód 76 do **IDataAccessor** přidat hlavičky metod **Save** a **Load**), má zároveň i tu výhodu, že pokud bude třída implementovat takovéto rozhraní (zde **IDataAccessor**), pak zároveň implementuje i rozhraní k němu tímto způsobem připojená (zde **ISaver** a i **ILoader**), aniž by je musela explicitně zmiňovat mezi implementovanými. Do tohoto rozhraní je také možné (ale nepovinné) přidat i další složky (např. zde metodu **Delete**, viz ř. 10).

4 Závěr

Jazyk C# disponuje obrovskou spoustou různých možností, technik a pomůcek, které při psaní kódu můžeme používat, přičemž neustále přibývají další. Jejich vhodné používání nám může při psaní kódu velmi usnadnit práci, avšak nevhodným užitím mohou i kód značně zkomplikovat, co do jeho čitelnosti, funkčnosti a udržitelnosti.

Není tedy úplně nezbytné vše ze zde uvedeného dokonale znát a neustále všude používat. Funkční programy lze napsat i se základní sadou několika málo příkazů, kterými disponovaly jazyky už ve svých prvopočátcích. Nové funkcionality jsou do C# přidávány tak, aby fungování těch starých nijak nenarušovaly, ale pouze rozšiřovaly jejich možnosti. Chceme-li však udržet krok s neustále se rozvíjejícími technologiemi a využívat i jejich možností, letmý přehled současných schopností programovacího jazyka není na škodu, v případě potřeby pak lze aktuální konkrétní způsoby užití kdykoli dohledat.

Mezi další možnosti a techniky C#, na které nám zde již nezbývá prostor, patří například LINQ, generické třídy, reflexe, vlákna a asynchronní přístup, dependency injection, přetěžování operátorů, design patterns, strojové učení a umělá inteligence atd. V programování je tedy možné se neustále učit něčemu novému, rozšiřovat tak své schopnosti o další možnosti, které mohou vývojáři usnadnit a zpestřit jeho každodenní činnost.

Programování tedy není o tom se jen naučit jeho základy a ty už dále pouze používat, ale jde o celoživotní poslání a objevování nových možností, které nikdy nekončí.

Rejstřík

Tento materiál vznikl v rámci realizace projektu
Strategický rozvoj Univerzity Hradec Králové, reg. č. CZ.02.2.69/0.0/0.0/16_015/0002427

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY