# C# pro 2. ročník

Skripta PVA pro druhý ročník.


- Výjimky a validace
- Práce s náhodnými čísly
- Vícerozměrné pole
- Práce s časem
- Práce s textovými soubory
- Struktura a třída (teorie OOP)
- Objektově orientované programování
- Maturita 2023

# Výjimky a validace

## Úvod

V této kapitole si řekneme, jak v C# ošetřovat nesprávné vstupy od uživatele a jak pracovat s výjimkami. Pro tyto účely nám poslouží třída `Exception`.

## K čemu slouží výjimka

Výjimky slouží k ošetření krajních situací při běhu programu a případnému ukončení programu alespoň s vypsáním chybové hlášky. Jejich používání by se tedy nemělo přehánět. Obecně platí, že pokud náš kód vyvolává fatální výjimky při běžném používání, je správně jej přepsat tak, aby i v krajních situacích fungoval alespoň na základní úrovni a nebylo nutné jej ukončit. Správné použití výjimek by mělo být spíše pro debuggovací (tzn. testovací) účely. Pokud program poskytneme uživatelům, měla by se výjimka objevit jen tehdy, když nastane situace, kterou jako programátoři nemůžeme ovlivnit. Mezi takovéto situace můžeme řadit například absenci administrátorských práv, absence důležitých balíčků .NET frameworku a tak dále.

**# Výjimka jako objekt**

Jak jsme si již v předchozích kapitolách řekli, C# je silně objektově orientovaný programovací jazyk. Nemělo by tedy být překvapením, že výjimky jsou také objekty a je tedy možné s nimi podle toho pracovat - tedy volat jejich metody a pracovat s jejich vlastnostmi. Je také možné vytvářet vlastní výjimky, kromě používání již vestavěných. C# má však bohatou zásobu výjimek a pro naše účely by nemělo být nutné přidávat žádné vlastní. Každá výjimka je v jazyce C# potomkem třídy `Exception`. Přesný význam toho, co je to potomek třídy se dozvíte později v kapitole `dědičnost`. Prozatím si jenom řekneme, že výjimek existuje velké množství a všechny vychází z již zmíněné třídy `Exception`.

## Jak vyvolat výjimku?

Každému už se to určitě alespoň jednou podařilo. Z edukativních důvodů si ale ukážeme například `DivideByZeroException`. Jak název napovídá, tuto chybu můžeme vyvolat při pokusu o dělení číslem nula. Vše si ukážeme na jednoduchém příkladě:

Dělení nulou

```
// Naše pomocná proměnná
int i = 0;

// Pokus o dělení nulou
int x = 14 / i;
```

Každý jistě z hodin matematiky zná, že v oboru reálných čísel nemá takováto operace žádný smysl. Nulou zkrátka nelze dělit. Stejného názoru je i kompilátor jazyka C# a pokud takováto situace nastane, dojde k neočekávanému ukončení programu a následnému vyhození výjimky `DivideByZeroException`. Jazyk C# nám ale dává možnosti, jak se s neočekávanými situacemi tohoto typu vypořádat. To může být užitečné v momentě, kdy například programujeme kalkulačku a nechceme, aby náš program při dělení nulou padal.

# Blok try-catch

Blok `try-catch` nám umožňuje zachytit výjimku hned po jejím vzniku a na danou situaci patřičně reagovat.

Výjimky

```
try
{
   int i = 14 / int.Parse(Console.ReadLine());
}
catch
{
   Console.WriteLine("Nelze dělit nulou.");
}
```

V případě zadání čísla `0` by tedy program dále pokračoval vypsáním řetězce "Nelze dělit nulou." Za výraz catch můžeme ještě dopsat kulaté závorky s libovolným výjimkovým typem čímž docílíme, že odchytávat budeme pouze právě zmíněný typ:

Zachycení konkrétní výjimky

```
Form form = null;

try
{
   int x = 100 / form.Size.Width;
}
catch (DivideByZeroExpception ex)
```

```
{
   Console.WriteLine(ex.Message);
}
```

Takovýto fragment kódu by vyvolal `NullReferenceException`, neboť objekt (proměnná) `form` není inicializována - odkazuje na hodnotu `null`. Ačkoliv je celé tvrzení obaleno `try` blokem, dojde i tak k pádu programu a vyhození výjimky, neboť blokem `catch` odchytáváme pouze výjimky typu `DivideByZeroExpception`. V těle bloku `catch` pak vypisujeme obsah chybového sdělení za pomocí vlastnosti `Message`.

# Vlastní vyvolání výjimky

Jazyk C# umožňuje vyvolat v krajních situacích výjimku a to buď naši vlastní nebo již předvytvořenou. Vše ilustruje následující ukázka:

Vyvolání výjimky

```
throw new Exception("Něco se nepovedlo...");
```

Klíčové slovo `throw` značí, že vyvoláváme výjimku. Následuje klíčové slovo `new`, jelikož vytváříme novou instanci třídy (objekt) a na závěr následuje název třídy, který označuje, jakou výjimku vyvoláváme. V tomto případě se jedná o obecnou generickou výjimku. Do konstruktoru výjimky můžeme předat řetězec popisující chybu.

# Validace vstupů pomocí tryParse

Pojďme si nyní představit metodu `TryParse()`. Tato metoda slouží pro konverzi mezi datovými typy. Přebírá dva argumenty. Prvním je hodnota, kterou chceme přetypovat a druhým je proměnná, do které se má uložit výsledek konverze. Návratovým typem této metody je `bool`, který signalizuje, zda konverze proběhla v pořádku, či nikoli:

Bezpečné přetypování

```
// Hodnota, jenž budeme převádět na int
string hodnota = "23abc";
int vysledek;

// Přetypování
bool uspech = int.TryParse(hodnota, out vysledek);
```

Pojďme si nyní rozebrat tento fragment kódu. Na začátku inicializujeme proměnnou, která je typu `bool` a do které bude uloženo, zda konverze proběhla v pořádku. Samotnou metodu `TryParse()` můžeme volat z jakéhokoliv primitivního datového typu. Protože náš řetězec `23abc` se chceme

pokusit převést na datový typ `int`, voláme metodu právě z tohoto datového typu. Prvním argumentem je řetězec, který se budeme pokoušet převádět a druhým argumentem je proměnná, do které se v případě úspěšného převodu uloží přetypovaná hodnota. Povšimněte si, že druhému argumentu předchází klíčové slovo `out`. Metoda `TryParse()` nemůže přetypovanou hodnotu vracet, neboť již vrací `bool` symbolizující úspěšnost či neúspěšnost konverze. Potřebujeme-li tedy, aby metoda vracela více jak jednu hodnotu, klíčové slovo `out` je cesta, jak toho docílit.

V případě naší ukázky bude konverze neúspěšná, neboť náš řetězec obsahuje písmena. Proměnná `uspech` by tedy nabyla hodnoty typu `false` a proměnná `vysledek` by zůstala nezměněná.

# Shrnutí

V této kapitole jsme si objasnili práci s výjimkami a třídou `Exception`. Jejich hierarchii, syntaxi, jak je vyvolávat a ošetřovat. Nakonec jsme si řekli, jak pracovat se uživatelským vstupem a ošetřit nesprávné vstupy od uživatele pomocí metody `TryParse()`.

# Práce s náhodnými čísly

## Úvod

V této kapitole si ukážeme, jak generovat náhodná čísla pomocí třídy `Random`.

## Problematika náhodných čísel v IT

Jedna z věcí, kterou počítač neumí udělat, je právě náhodné číslo. Algoritmy na generování takovýchto čísel dokážou vytvořit dlouhou sekvenci čísel, které pouze vypadají náhodně. Náhodná čísla se v programech generují podle tzv. `seedu`. Když nastavíme seed na jednu hodnotu, dostaneme vždy stejnou náhodnou sekvenci.

## Třída Random a její metody

Abychom mohli začít generovat čísla, musíme nejdříve vytvořit novou instanci třídy `Random`.

Vytvoření instance

```
// Vytvoření generátoru náhodného čísla s "náhodným" seedem
Random r = new Random();


// Vytvoření generátoru n. č. se specifickým seedem
Random r2 = new Random(123);
```

> Když neurčíme seed při vytvoření generátoru, nastaví se seed podle času. Tzn. že když v jedné sekundě vytvoříme více instancí třídy `Random`, budou mít všechny tyto instance stejný seed. V praxi to znamená, že bychom se měli vyhnout vytváření instancí třídy `Random` například uvnitř cyklů, jinak nám hrozí situace, že námi vygenerovaná čísla budou všechny shodná.

Nyní máme vytvořený náš generátor a můžeme začít generovat čísla. Třída `Random` nám k tomu poskytuje dvě hlavní metody:

Hlavní metody

```
Random r = new Random();
```

```
int i = r.Next();
double d = r.NextDouble();
```

### # Metoda Next()

Nyní začneme metodou `Next()`. Tu použijeme, pokud budeme chtít vygenerovat náhodné celé číslo (`int`). Když metodu napíšeme bez parametrů, tak integer, který dostaneme, bude náhodné číslo větší nebo rovno nule. Tato metoda má ale více možností, jak ji použít. Pokud do závorky u metody `Next()` dáme jeden parametr typu `int`, nastavíme tím vrchní hodnotu (maximum), ke které se generátor může přiblížit. Když přidáme ještě jeden další parametr, dostaneme tím možnost nastavit celý rozsah. Tzn. minimum i maximum:

Náhodné číslo

```
Random r = new Random();

// Hodnoty pro "ohraničení" náhodného čísla
int min = 5;
int max = 25;

// Číslo větší nebo rovno nule [dostaneme číslo v rozsahu 0 - ...]
r.Next();

// Číslo větší nebo rovno nule a menší než parametr max [dostaneme číslo v rozsahu 0 - 24]
r.Next(max);

// Číslo větší nebo rovno parametru min a menší než parametr max [dostaneme číslo v rozsahu 5 - 24]
r.Next(min, max);
```

> Minimum je inkluzivní (hodnota spadá do generované sekvence) zatímco maximum není.

### # Metoda NextDouble()

Tato metoda narozdíl od ostatních dvou nevyžaduje žádný parametr, neboť má přesně daný rozsah. Pomocí ní můžeme získat čísla datového typu `double`, která jsou v intervalu od 0 do 1:

Desetinné náhodné číslo

```
Random r = new Random();

// Vypíše náhodný double
Console.WriteLine(r.NextDouble());
```

# Realizace náhodného čísla bez opakování

Když generujeme náhodná čísla, našemu generátoru nevadí udělat klidně více stejných čísel za sebou. Někdy se to náhodou stane. Ale v nějakých případech chceme takovému chování zabránit. Například ve sportce, když se losují čísla, nikdy nemůže dostat více stejných čísel za sebou. Teď si ukážeme, jak takového výsledku docílit. Budeme si muset vytvořit list, do kterého postupně budeme přidávat hodnoty, které jsme už použili. Pomocí metody `Contains()` pak budeme kontrolovat, zda se číslo, které jsme vygenerovali, neshoduje s nějakým v našem listu. Proces budeme opakovat, dokud nenajdeme číslo, které jsme ještě nepoužili:

Náhodné číslo bez opakování

```
// List hodnot, které jsme již použili
List<int> pouziteHodnoty = new List<int>();

// Pokusíme se vypsat 100 čísel, které se neopakují
for (int i = 0; i < 100; i++)
{
    // Hodnota, se kterou právě pracujem
    int generovanaHodnota;

    do
    {
        // Necháme vytvořit náhodné číslo
        generovanaHodnota = r.Next();

    } while (pouziteHodnoty.Contains(generovanaHodnota)); // Pokud ho náš List již obsahuje, zkusíme vytvořit další

    // Až se nám podaří najít hodnotu, kterou jsme ještě nepoužili, tak ji přidáme do listu
    pouziteHodnoty.Add(generovanaHodnota);

    // Vypíšeme unikátní číslo
    Console.WriteLine(generovanaHodnota);
}
```

> Pokud bychom chtěli náhodná čísla bez opakování v nějakém intervalu (např. od 0 do 10), museli bychom si pohlídat i situaci, kdy nám žádná náhodná čísla už nezbývají. Pokud by tato situace nastala, mohli bychom například vyhodit náš vlastní `Exception`.

# Shrnutí

V této kapitole jsme se naučili pracovat s třídou `Random` a ukázali jsme si, jak vyřešit časté problémy při jejím používání.

# Vícerozměrné pole

## Úvod

V této kapitole se naučíme pracovat s různými druhy vícerozměrných polí.

## Deklarace dvourozměrného pole

Dvourozměrné pole se deklaruje podobně jako jednorozměrné. Jediný rozdíl je ten, že musíme nějak oddělit dva rozměry (2 dimenze). Např. takto deklarujeme prázdné dvourozměrné pole. Jediné co přibylo je čárka a údaj o velikosti druhého rozměru:

Deklarace

```
// Dvourozměrné pole o velikosti 5x5
int[,] pole = new int[5, 5];
```

K prvkům se přistupuje také podobně jako v jednorozměrném poli:

Přístup k prvku pole

```
// Takto přistoupíme k prvku na souřadnicích [2,3] a nastavíme mu hodnotu 42
pole[2, 3] = 42;
```

> Stejně jako u jednorozměrných polí se indexuje od nuly, tudíž souřadnicemi `[2,3]` přistoupíme k prvku, který je třetí v prvním rozměru a čtvrtý v druhém.

Dvourozměrné pole se dá deklarovat rovnou i s hodnotami, stejně jako to fungovalo u jednorozměrných polí:

Inicializace hodnotami

```
// Dvourozměrné pole o velikosti 3x3 předem naplněné
int[,] pole = new int[,] { { 51, 42, 69 }, { 999, 666, 888 }, { 360, 420, 480 } };
Pokud bychom chtěli v tomto poli přistoupit k číslu 360, museli bychom použít
```

Pokud bychom chtěli v tomto poli přistoupit k číslu `360`, museli bychom použít souřadnice `[2,0]`.

# Metoda GetLength()

Často se stane, že při práci s vícerozměrnými poli nebudeme vědět, jak je pole velké v jakých směrech. V jednorozměrném poli nám stačila vlastnost `Length`. Abychom tyto hodnoty zjistili ve vícerozměrném poli, musíme použít metodu `GetLength()`, jež jako svůj parametr přejímá dimenzi:

Délka vícerozměrného pole

```
// Pole 2x2
int[,] pole = { {6, 7}, {3, 9} };

// Rozměry
int delka = pole.getLength(0);
int vyska = pole.getLength(1);
```

Vlastnost Length u vícerozměrného pole vrací počet všech prvků v poli. Tzn. u našeho pole `2x2` nám vrátí číslo `4`.

# N-rozměrné pole

Jestli jste si mysleli, že pole může být pouze jedno či dvourozměrné, tak jste se mýlili. Pole může mít totiž nespočet dimenzí. Pro ukázku je zde pole, které má 34 dimenzí a stále funguje. Avšak pro většinu lidí už nemá takové pole reálné využití:

Pole o více dimenzích

```
int[,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,] nesmyslneVelkePole;
```

> Toto pole ještě nemá určené velikosti daných dimenzí, tzn. není inicializované.

# Pole polí

Toto je speciální druh vícerozměrných polí, jelikož všechny pole v tomto poli nemusí mít stejnou velikost. Můžeme si to ukázat např. na kinosálu. Tam nemají všechny řady stejně sedadel neboť některé chybí kvůli vchodu do sálu. Abychom popsali, jak se jmenují lidé, co v tomto sálu sedí, by nám normální dvourozměrné pole sice stačilo, ale zůstali by v něm místa, které ani neexistují. Takovéto pole můžeme založit následujícím způsobem:

Založení pole polí

```
// Deklarace pole, ve kterém je 5 polí o kterých nevíme, jak velká jsou
int[][] pole = new int[5][];
```

Dalším příkladem může být hotel, který má `X` pater a v každém patře je `Y` pokojů. Velikost `X` bude vždy jedna, ale každé patro může mít jiný počet pokojů, tudíž pro každé `X` bude jiná velikost `Y`:

Založení pole polí

```
// Předem naplněné pole polí
int[][] hotel = new int[][] {
    new int[]{101, 102, 103, 104, 105, 106},
    new int[]{201, 202, 203},
    new int[]{301, 302, 303}
};

// Tímto zjistíme, jaké je číslo čtvrtého pokoje v prvním patře
int cisloPokoje = hotel[0][3];
```

# Shrnutí

V této kapitole jsme se dozvěděli, jak používat n-rozměrné pole a jak zjistit velikosti jejich dimenzí. Také jsme si ukázali jak fungují pole polí též někdy nazývané jagged array.

# Práce s časem

## Úvod

V této kapitole si ukážeme, jak používat třídy `DateTime` a `TimeSpan` k ukládání a počítání s časem.

## Třída DateTime

Na práci s časem budeme používat třídu `DateTime`. Do její instance můžeme uložit daný bod v časové linii. Tudíž datum i čas (proto Date Time). Náš objekt typu `DateTime` založíme jednoduše:

Deklarace a inicializace

```
// Založení DateTimu bez parametrů
DateTime dt = new DateTime();
```

Když při založení nepoužijeme zádné parametry, tak se nám nastaví hodnoty na původní. Což je datum `1.1.0001 0:00:00`. Třída však obsahuje ještě několik dalších parametrických konstruktorů, které nám umožní určit náš vlastní čas:

Konstruktory třídy DateTime

```
// Založení se zadáním roku / měsíce / dne (3. ledna 2021)
DateTime dt = new DateTime(2021, 1, 3);

// Založení se zadáním roku / měsíce / dne a hodiny / minuty / sekundy (2. prosince 2021,
11:32)
DateTime dt2 = new DateTime(2021, 12, 2, 11, 32, 0);
```

> Toto není výčet všech možných konstruktorů, ale tyto jsou nejvyužívanější.

Počítače pracují s jednotkou času, které říkáme ticks. Každá desetitisícina milisekundy, se dá vyjádřit jedním `tickem` (tzn. 1ms = 10000 ticks). S touto jednotkou můžeme instanci třídy `DateTime` také založit, ale tento zápis se moc nepoužívá, jelikož z hlavy není tak jednoduché převést ticky na datum.

Konstruktory třídy DateTime

```
// Založení DateTimu s datem 19 /12 / 1539 09:13:53
DateTime dt = new DateTime(485649547898999489);
```

# Posun v čase

Pokud pracujeme s časem, nejspíš nebudeme chtít mít uložené jedny stálé hodnoty. Teď si tedy ukážeme, jak manipulovat s jednotlivými jednotkami času. Microsoft si pro nás připravil několik užitečných metod:

Posun času (1)

```
dt.AddYears(1);
dt.AddMonths(2);
dt.AddDays(5);
dt.AddHours(4);
dt.AddMinutes(1);
dt.AddSeconds(99);
dt.AddMilliseconds(700);
dt.AddTicks(9999);
```

Zadání samozřejmě lze i různě variovat:

Posun času (2)

```
dt.AddYears(-1);
dt.AddMonths(2);
dt.AddDays(5.99);
dt.AddHours(-4.156);
dt.AddMinutes(1.19156);
dt.AddSeconds(-2.11);
dt.AddMilliseconds(0.555);
dt.AddTicks(-4894498);
```

Pro každou jednotku existuje vlastní metoda `Add[jednotka]()`. Pomocí nich můžeme přidávat či odebírat jednotlivé jednotky času a posouvat tím datum, které jsme na začátku určili. Zadané hodnoty nemusí být pouze kladné. Můžeme metodě dát i mínusová čísla a tím posunem čas zpět (ukázka 2).

> Dny, hodiny, minuty, sekundy a milisekundy se dají zadávat i pomocí desetinného čísla. Roky a měsíce ne, jelikož nejsou pokaždé stejně dlouhé.

>

> Ticky se musí zadávat v celých číslech, jelikož jsou nejmenší možná jednotka.

# Získání aktuálního času

Dost často se při tvorbě programu, který nějakým způsobem používá `DateTime`, setkáme s tím, že budeme chtít získat čas, který je právě v tu danou dobu. Pro tyto účely nám slouží statická vlastnost `Now`.

Momentální čas

```
DateTime casTed = DateTime.Now;
```

Toto nám do nové proměnné casTed uloží datum a čas, kdy se tato operace provedla. Do teď jsme si ukázali jen manipulaci s třídou `DateTime`, ale nyní si představíme pár způsobů, jak z jejích hodnot získat lehce čitelný výstup. Jedná se o metodu `ToString()` a její speciální varianty.

Výpis času

```
// Dlouhý výpis data (úterý 3. září 2019)
Console.WriteLine( dt.ToLongDateString() );

// Dlouhý výpis času (13:03:16)
Console.WriteLine( dt.ToLongTimeString() );

// Krátký výpis data (3.09.2019)
Console.WriteLine( dt.ToShortDateString() );

// Krátký výpis času (13:03)
Console.WriteLine( dt.ToShortTimeString() );

// Krátký výpis data i času (3.09.2019 13:03:16)
Console.WriteLine( dt.ToString() );
```

Samozřejmě bychom mohli vzít ručně hodnoty z instance `DateTime` a nějak je poskládat, ale tyto formátovací metody nám práci velmi zjednoduší.

# Třída TimeSpan

Další důležitou třídou je `TimeSpan`. Ta nám narozdíl od `DateTime` neudává přesný bod v čase, ale časový rozsah. Můžeme ho vytvořit podobně jako `DateTime` pomocí konstruktoru, nebo ho můžeme získat díky odečtení dvou instancí třídy `DateTime`.

> `TimeSpan` neumí ukládat roky a měsíce. Pouze dny a menší jednotky, jelikož roky i měsíce nemusí být vždy stejně dlouhé.

Inicializace třídy TimeSpan

```
// Časový úsek dlouhý 5 hodin, 2 minuty a 1 sekundu
TimeSpan ts = new TimeSpan(5, 2, 1);

// Získání TimeSpanu pomocí rozdílu dvou DateTimů
TimeSpan rozdil = dt2 - dt1;

// Přičtení TimeSpanu k DateTimu pomocí metody Add()
dt1.Add(ts);

// Přičtení TimeSpanu k DateTimu pomocí operátoru
dt1 += ts;
```

Jak vidíte, se třídou `TimeSpan` se dá pracovat velmi jednoduše. Instance třídy `TimeSpan` mají rovněž přístupné vlastnosti jako je `Days`, `Hours`, `Minutes` a `Seconds`, které nám umožní získat jednotlivé časové úseky z intervalu.

# Ukázky užití

Chceme si ohřát jídlo v mikrovlnné troubě a poté budeme chtít vědět, jaký bude čas, až se nám jídlo ohřeje. Nejdříve tedy zjistíme momentální čas pomocí `DateTime.Now` a poté k němu přičteme `TimeSpan`, který si předtím vytvoříme

Ukázka 1

```
// Momentální čas (začátek)
DateTime zacatek = DateTime.Now;

// Vytvoření TimeSpanu o délce 5 minut a 30 sekund (doba ohřívání)
TimeSpan casOhrivani = new TimeSpan(0, 5, 30);

// Čas, kdy bude jídlo připraveno ke konzumaci
DateTime konecnyCas = zacatek + casOhrivani;

// Výpis konečného času
Console.WriteLine(konecnyCas.ToShortTimeString());
```

V dalším příkladu budeme chtít změřit čas auta v závodu. Na začátku budeme čekat na stisk klávesy, po kterém se nám uloží startovní čas. Po druhém stisku klávesy pak zjistíme, jak dlouho autu trvalo projet celou trať:

Ukázka 2

```
// Start závodu
Console.WriteLine("Auto čeká na odstartování");
Console.ReadKey();
DateTime start = DateTime.Now;

Console.WriteLine("Auto vyrazilo!");

Console.ReadKey();
DateTime konec = DateTime.Now;

// Rozdíl mezi koncem a začátkem závodu je čas jeho průběhu
TimeSpan casZavodu = konec - start;
Console.WriteLine("Auto dorazilo do cíle v čase:" + casZavodu.ToString());
Console.ReadKey();
```

# Shrnutí

V této kapitole jsme se naučili používat třídy `DateTime` a `TimeSpan`. Nesmíme zapomenout, že třída `DateTime` ukládá přesný čas a `TimeSpan` jen časový úsek.

# Práce s textovými soubory

## Úvod

V této kapitole si ukážeme, jak číst a zapisovat do textových souborů.

## Cesty

Než si ukážeme konkrétní třídy a metody pro práci se soubory, vysvětlíme si některá fakta ohledně zápisu cest k souborům v jazyce C#. K přístupu do podadresářů používáme lomítka. V unixových systémech používáme klasické lomítko (`/`) a na platformě Windows lomítko zpětné (`\`). Jazyk C# umožňuje použití obou možností, ale z historických důvodů bychom důrazně doporučovali držet se lomítka zpětného.

**# Zpětné lomítko**

Kromě znázorňování hierarchické struktury adresářů slouží lomítko i k tzv. escapování znaků. Pro lepší vysvětlení tohoto pojmu si vše ukážeme na jednoduchém příkladě. Představme si situaci, kdy chceme do stringové proměnné s uložit jednu uvozovku:

Ukázky

```
string s = """; // Ukázka 1
string s = "\""; // Ukázka 2
```

Kompilátor by uvozovku ve prostřed vnímal jako validní znak a ukončil by tak textový řetězec na pravé straně výrazu. Třetí uvozovka by pak byla vnímána jako další neukončený řetězec. Zpětné lomítko je cesta, jak uvozovku escapnout a říct tím kompilátoru, aby ji vnímal jako obyčejný literál a nikoliv jako klíčový znak (viz ukázka 2). Pokud tedy budeme chtít uložit cestu k souboru do stringové proměnné, musíme nějakým způsobem kompilátoru říct, aby zpětné lomítko vnímal jako literál pro oddělení adresářové struktury a nikoliv jako pokus o escapování následujícího znaku. Máme hned dvě možnosti, jak tohoto chování docílit

Možnosti

```
string cesta = "c:\\pepa\\aplikace\\data.txt"; // Možnost 1
string cesta = @"c:\pepa\aplikace\data.txt"; // Možnost 2
```

První možností je escapnout samotný znak escapnutí a použít tedy dvě zpětná lomítka. Druhou, více elegantnější možností je prefixovat celý řetězec znakem zavináče (`@`), čímž řekneme

kompilátoru, aby všechny znaky v řetězci vnímal jako literály.

> Pokud vyvíjíte na unixovém operačním systému (např. MacOS nebo Linux), musíte používat klasické lomítka (`/`)!

**# Relativní cesty**

Jazyk C# nám dává možnost odvíjet cesty i relativně od rootovského adresáře. Root neboli kořenový adresář programu je taková složka, kde se nachází spustitelný `.exe` soubor (nebo jiný spouštěcí soubor) naší aplikace. Standardně je to složka `název-projektu\bin\debug\`. Cestu označíme jako relativní, pokud na její začátek uvedeme tečku (`.`):

Cesty

```
string relativniCesta = @".\data"; // Relativní cesta
string absolutniCesta = @"c:\aplikace\mujProgram\data"; // Absolutní cesta
```

Vždy je lepší cesty odvíjet relativně od spouštěcího souboru, neboť absolutní cesty mohou přestat fungovat v momentě, kdy program spustíme na jiném počítači. V ukázce 2 je pak na prohlédnutí i cesta absolutní. Pro vstoupení do vyšší adresářové struktury (rozuměj pro opuštění současné složky) pak slouží tečky dvě:

Notace dvou teček

```
// Začneme v kořenovém adresáři, vystoupíme o úroveň výše
// a poté vstoupíme do složky se jménem "data" a vybereme
// soubor se jménem "soubor.txt":
string cesta = @".\..\data\soubor.txt";

/*

aplikace/
├── root/ (kořenový adresář)
│   └── aplikace.exe
├── data/
│   └── soubor.txt (vybraný soubor)
└── src/
    ├── obrazek.png
    └── hudba.mp3

*/
```

# Třída File

Nejjednodušší způsob, jak manipulovat s textovým souborem je třída `File`. Ta obsahuje statické metody, které pro nás obstarají všechny základní funkčnosti, které bychom mohli při práci se soubory potřebovat. Třída `File` je šikovná i v tom, že soubor sama otevře i uzavře, takže tuto činnost nemusíme provádět ručně. Abychom mohli třídu `File` vůbec používat, musíme našemu projektu říct, aby využíval namespace `System.IO`, který třídu obsahuje. Uděláme to pomocí klíčového slova `using`:

Namespace import

```
using System.IO;
```

> Tento namespace (i jakýkoliv jiný) si pochopitelně nemusíme pamatovat zpaměti. Každé dobré vývojové prostředí nám tento namespace sám nabídne k doplnění v momentě, kdy se na třídu `File` odkážeme.

**# Vytvoření souboru**

Začneme od nejjednoduššího a ukážeme si, jak vytvoříme nový `.txt` soubor. K tomuto účelu slouží metoda `Create()`:

Vytvoření souboru

```
// Vytvoří textový soubor v kořenovém adresáři
File.Create("soubor.txt");
```

> Pokud vytvářený soubor již existuje, dojde k přepsání toho stávajícího!

Třída dále nabízí metody `Move()` a `Delete()`, jejichž názvy jsou více méně sebepopisné:

Přesunutí souboru

```
// Přesun souboru "soubor.txt" z kořenového adresáře
// do složky "data"
File.Move("soubor.txt", @".\data\soubor.txt");
```

Smazání souboru

```
// Smaže soubor "soubor.txt" z
// kořenového adresáře
File.Delete("soubor.txt");
```

**# Čtení ze souboru**

Nyní se podíváme na metody určené ke čtení ze souborů. Dvě nejčastěji používané z nich jsou `ReadAllText()` a `ReadAllLines()`. Oběma stačí v závorce pouze cesta k souboru:

Čtení ze souboru

```
// Načte kompletně celý obsah souboru do stringové proměnné
string obsah = File.ReadAllText("soubor.txt");

// Načte obsah celého souboru po řádcích do pole
string[] radky = File.ReadAllLines("soubor.txt");
```

> Tyto metody načítají obsah souboru do RAM paměti počítače. Pokud bychom pracovali s opravdu velkým textovým souborem v řádech statisíců znaků, máme pro tyto účely třídu `StreamReader`, která je na takovéto situace lépe uzpůsobená.

Obě metody dělají totéž, ale každá vrátí výstupní data v jiném datovém typu. Metoda `ReadAllText()` vrací celý obsah v jedné stringové proměnné a nové řádky odděluje symbolem pro to určeným, kdežto `ReadAllLines()` vrací stringové pole, kde každý záznam je právě jeden řádek.

> Symbol(y) pro nový řádek je na platformě Windows `\r\n` a na unixových operačních systémech `\n`. Pokud si nejsme jistí, jaký symbol použít, máme pro tyto účely k dispozici i vlastnost `Environment.NewLine`

**# Zápis do souboru**

K jednoduchému zápisu využíváme metody `WriteAllText()` a `WriteAllLines()`. Fungují podobně, jako jejich protějšky na čtení:

Zápis do souboru

```
// Pole s textem
string[] pole = new string[] {"Věta 1", "Věta 2"};

// Zápis do souboru
WriteAllText("soubor.txt", "Text, který chceme zapsat do souboru.");
WriteAllLines("soubor.txt", pole);
```

Opět platí, že pokud cílový soubor neexistuje, vytvoří se nový. Pokud textový soubor již obsahuje nějaký text, dojde k jeho přemazání. Pokud nechceme přepsat celý soubor a ztratit tím všechna data, co v něm předtím byla, můžeme použít metodu `AppendAllText()`. Tato metoda vezme obsah, který v souboru byl a na konec přidá námi zadaný text:

Zápis do souboru

```
File.AppendAllText("soubor.txt", "Text, který se přidá na konec, bez přepsání původního obsahu souboru.");
```

Dalším způsobem jak pracovat s textovými soubory jsou třídy `StreamReader` a `StreamWriter`. Tyto třídy nenačítají soubory do své paměti celé najednou, ale umožňují z nich číst nebo do nich zapisovat řádek po řádku. Daní za tento fakt je zdlouhavější zápis, neboť musíme vytvářet jejich instance a poté i ručně uzavírat datový stream. Výhodou tohoto přístupu je však možnost pracovat s objemnými soubory v poměrně krátkém čase za rozumného vytížení paměti.

**# Třída StreamReader**

Když vytvoříme instanci třídy `StreamReader`, dojde k otevření souboru, který jsme mu dodali jako argument. V praxi to znamená, že od této doby k němu nedostane přístup žádný jiný program. Aby se nám nestalo, že necháme nějaký soubor "zamknutý" musíme ho vždy poté, co jsme s ním hotovi, zavřít pomocí metody `Close()`:

Založení StreamReaderu (1)

```
// Vytvoření nové instance třídy StreamReader
StreamReader reader = new StreamReader("soubor.txt");

// ... logika čtení

// Uzavření streamu
reader.Close();
```

Druhou naší možností je samotné instancování třídy obalit do bloku `using`, jež sám zajistí uzavření streamu:

> V prvním ročníku jsme zmínili existenci tzv. `garbage collectoru`, který automatizuje správu paměti. V praxi to znamená, že pokud založíme novou proměnnou, běhové prostředí automaticky rozpozná, kdy proměnná již nebude potřeba a alokovanou (rozuměj zabranou) paměť samo uvolní. Některé zdroje jako naše textové soubory jsou však mimo kompetence tohoto collectoru a o jejich uvolnění se musíme postarat sami. Třídy pracující s takovýmito zdroji implementují rozhraní `IDisposable`, které nutí danou třídu implementovat metodu `Dispose()`, jež se stará o uklizení "nepořádku" po volání takovýchto tříd. Mezi takovéto třídy spadají i naše třídy `StreamReader` a `StreamWriter`. Jejich použití v using bloku zajistí zavolání výše zmíněné `Dispose()` metody a vyhneme se tak tudíž potencionálním nepříjemnostem. Tato informace je spíše doplňující látkou pro pokročilé.

Díky tomu, že nám soubory zůstávají otevřené, nemusíme číst celý soubor najednou, ale můžeme s ním manipulovat jen po řádcích či znacích. Když používáme jednotlivé metody na čtení, náš reader si zapamatuje pozici, na které přestal znaky číst:

Čtení po znaku

```
// Vytvoření nové instance třídy StreamReader
using (StreamReader reader = new StreamReader("soubor.txt"))
{
    // Dokud nám zbývají nepřečtené znaky
    while (reader.Peek() >= 0)
    {
        // Převede výstup metody Read() z intu
        // (ASCCI hodnota znaku) na obyčejný znak (char)
        // a poté vypíše výsledek
        Console.Write((char)sr.Read());
    }
}
```

Takovýto kód by obsah souboru přečetl po jednom znaku a každý zvlášť vypsal do konzole. Podobně si počínáme i v případě, kdy chceme souborem iterovat po řádcích:

Čtení po řádcích

```
using (StreamReader reader = new StreamReader("soubor.txt"))
{
    // Pomocná proměnná pro uložení řádku
    string line;

    // Dokud nám zbývají další řádky
    while ((line = reader.ReadLine()) != null)
    {
        // Vypíšeme získaný řádek do konzole
        Console.WriteLine(line);
    }
}
```

**# Třída StreamWriter**

Pro zapisování máme ve třídě `StreamWriter` jen dvě metody - `Write()` a `WriteLine()`. Jejich použití je téměř shodné. Obě varianty do souboru zapíší předaný řetězec, ale metoda `WriteLine()` ještě na konec přidá nový řádek

Zápis do souboru

```
using (StreamWriter writer = new StreamWriter("soubor.txt"))
{
    // Zapíše řetězec na aktuální řádek
```

```
    writer.Write("Požadovaný řetězec");

    // Zapíše řetězec na nový řádek
    writer.WriteLine("Budu na novém řádku!");
}
```

Podobně jako u třídy `StreamReader` musíme opět uzavřít stream buď voláním metody `writer.Close()` nebo obalením celého výrazu `using` blokem tak, jako na ukázce výše.

# Shrnutí

V této kapitole jsme se seznámili se zápisem a čtením z textových souborů. Měli bychom být obeznámeni se třídami `File`, `StreamReader` a `StreamWriter`.

# Struktura a třída (teorie OOP)

## Úvod

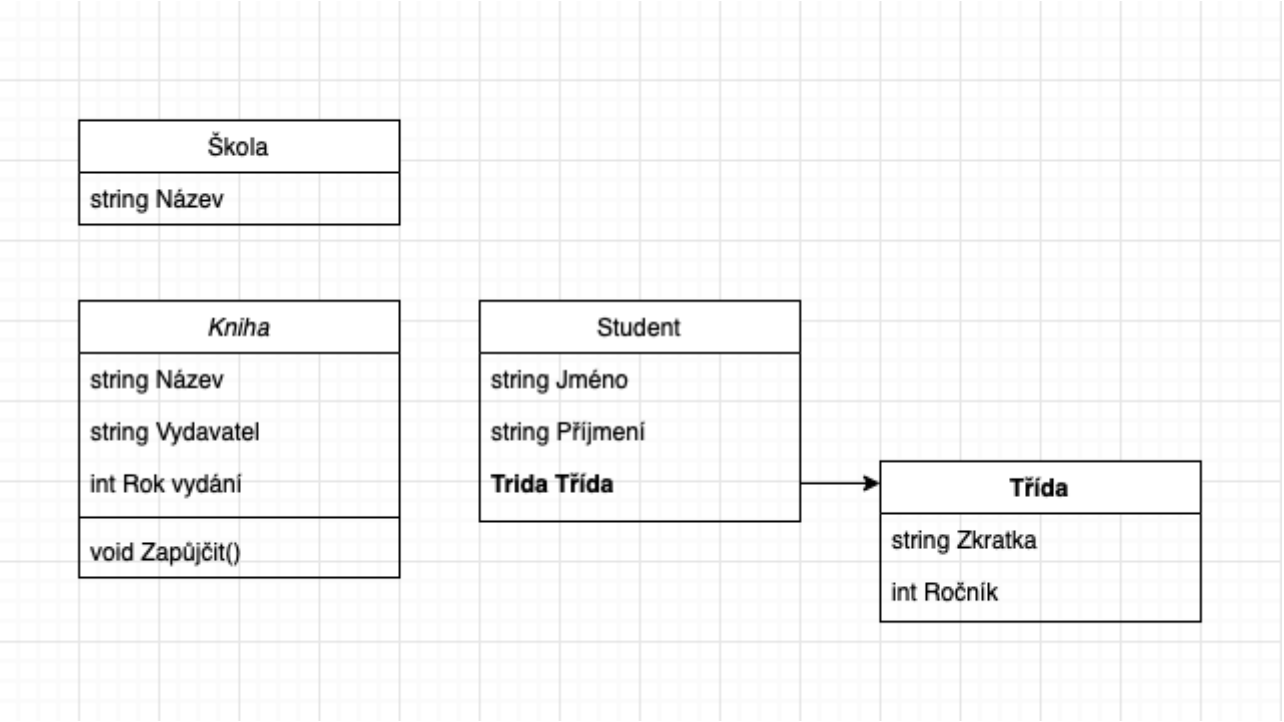
Jazyk C# je stejně jako mnoho ostatních programovacích jazyků založen na objektech, které jsou jeho základními stavebními kameny. Je proto velmi důležité chápat takto jazyk už od samého začátku. Co to ale ten objekt vlastně je? Než si na tuto a řadu dalších otázek odpovíme, uděláme si pro lepší pochopení látky malý výlet do historie a ujasníme si některá fakta.

## Objektově orientované programování (OOP)

Nikoho asi nepřekvapí, že vývoj softwaru je velmi mladé odvětví. Porovnejme si ho například se stavebnictvím. Domy stavíme již tisíce let, naproti tomu programovat jsme začali teprve nedávno. S tím souvisí i fakt, že do teď pořádně nevíme, jak software dobře navrhnout a hlavně jak efektivně řídit vývoj velkých projektů. Tak, jako se v průběhu času měnili požadavky na software, museli přicházet i nové metodiky a paradigmata, které si kladli za cíl, jak stále složitější programy vytvářet, řídit a především udržovat. Jedním takovýmto paradigmatem je i objektově orientované programování, které patří do základní výbavy každého dobrého programátora.

> Počátky objektově orientovaného programování sahají do sedmdesátých let minulého století, kdy Steve Jobs (zakladatel společnosti Apple) poprvé navštívil firmu Xerox. Tamní představitelé společnosti mu ukázali tři revoluční produkty. Byli jimi počítačová myš, první grafické uživatelské rozhraní a objektově orientované programování. Všechny tyto tři vynálezy později převratným způsobem změnili celé odvětví a do jisté míry je používáme i dnes.

Objektově orientované programování (OOP – object oriented programming) je paradigma (rozuměj způsob myšlení nebo vzor), které si klade za cíl do programování přenášet objekty a vztahy z reálného světa. Pokud bychom programovali například informační systém pro školní knihovnu, měli bychom objekty jako je **kniha**, **student**, **třída** nebo **škola**. Objekt kniha by obsahoval vlastnosti jako je **název**, **rok vydání** nebo **žánr** a objekt student kromě svých základních identifikačních vlastností může zahrnovat i metodu pro zapůjčení konkrétní knihy. Vše lépe ilustruje diagram níže.




*Jednoduchý diagram zobrazující možné třídy (business entity) pro informační systém školní knihovny.*

Obsah jednotlivých objektů (přesněji řečeno jejich rozhraní) je čistě na nás. Objekty a vztahy mezi nimi jsou pro člověka lépe představitelné a dovolují tak poměrně snadno řešit danou problémovou doménu. **V odborném jazyce bychom řekli, že OOP vnáší do naší práce více abstrakce.**

> Kromě pojmu **objekt** se můžeme v OOP setkat i s výrazem **instance**. **Objekt a instance jsou synonyma** a znamenají tudíž totéž.

# Třída a struktura jako základ objektu

Instance neboli objekt vzniká vždy na základě nějaké třídy nebo struktury. V minulé sekci jsme si základní myšlenku OOP ilustrovali na příkladu s knihovnou. Pojďme se tedy nyní podívat, jak by vypadala výroba obyčejného objektu knihy:

Třída Kniha

```
class Kniha
{


}
```

Jak již bylo zmíněno výše, základem každého objektu je vždy **třída** (`class`) nebo struktura. Strukturu nyní ponecháme stranou a zaměříme se pouze na třídu. V ukázce jsme definovali novou třídu se jménem `Kniha` a prozatím jsme její tělo ponechali prázdné. I takováto prázdná třída by

bohatě postačila jako startovní bod pro vznik nového objektu:

Výroba objektu (neboli instance) ze třídy

```
Kniha mojeKniha = new Kniha();
```

Tímto příkazem jsme vytvořili nový objekt třídy `Kniha` se jménem `mojeKniha`. Odborně bychom řekli, že jsme **vytvořili instanci třídy Kniha**. Pojďme se nyní blíže podívat na celý zápis. Začali jsme definicí proměnné se jménem `mojeKniha`. Samotná třída `Kniha` je zde datovým typem podobně,  jako například typy `int` nebo `string`. Kdykoliv tedy vytvoříme novou třídu, tvoříme tím i náš **vlastní nový datový typ**. O proměnné `mojeKniha` bychom tedy mohli říct, že je datového typu `Kniha`. Za operátorem přiřazení (`=`) poté následuje klíčové slovo `new` a poté volání tzv. konstruktoru. **Konstruktor** je speciální typ metody, který nám po svém zavolání vrací onu instanci neboli objekt. Pokud se ale blíže podíváme na implementaci naší třídy `Kniha` tak zjistíme, že zde žádný konstruktor definovaný nemáme. Jak je tedy možné, že konstruktor voláme? Kompilátor si totiž daný konstruktor vytvoří automaticky sám, pokud ve třídě žádný nenajde.

> Konstruktory budou látkou dalšího ročníku a toto je v tuto chvíli vše, co o nich v potřebujeme vědět.

Pro úplnost si ještě ukážeme celý kód viz ukázka níže:

Výroba objektu "mojeKniha"

```
using System;
□□□□□
public class Program
{
□public static void Main()
□{
□□Kniha mojeKniha = new Kniha();
□}
}

class Kniha
{

}
```

Samotná třída `Kniha` může být vytvořena na úrovni namespacu (viz ukázka) nebo může být oddělena do samostatného `.cs` souboru. V praxi se nejčastěji setkáme s druhou variantou, protože programy mohou obsahovat klidně i stovky tříd. Jejich členěním do samostatných souborů (často pojmenovaných stejně jako třída samotná - např. `Kniha.cs`) dosáhneme lepší organizovanosti našeho kódu. Na pořadí nebo způsobu, v jakém definujeme třídy nezáleží, pokud dodržíme

pravidlo, že všechny vytváříme ve stejném namespacu.

Když už nyní máte představu, co to objekt je a jak ho vytvoříme, asi se sami sebe ptáte, k čemu je toto všechno vlastně dobré? Užitečnost objektů si tedy nyní budeme demonstrovat na malé ukázce:

Třída s veřejnými proměnnými

```
class Kniha
{
    public string nazev;
    public int rokVydani;
}
```

Naše instance kniha1 a kniha2 nyní umožňují, aby do nich byl uložen název knihy a její rok vydání. Data v těchto instancích jsou na sobě nezávislá a každá instance může mít svá unikátní data. Musíme si dát pouze pozor na to, abychom k samotným datům objektů přistupovali právě přes jejich instance (proměnné kniha1 a kniha2) a nikoliv přes název jejich třídy (Kniha)! Až se naučíme pracovat s konstruktory, můžeme tento zápis ještě více zkrátit. **Samotná užitečnost objektů tedy tkví v enkapsulaci neboli zapouzdření dat.**

> Enkapsulace je jeden ze čtyř základních pilířů OOP.

Kdybychom OOP neznali, museli bychom náš kód zamořit velkou spoustou nepřehledných proměnných, jako je to vidět na následující ukázce:

Výroba knih bez znalosti OOP

```
kniha1Nazev = “Robinsoe Crusoe”;
kniha1rokVydani = 1797;
kniha2nazev = “Válka s mloky”;
kniha2rokVydani = 1940;
// a tak dále...
```

Naše třída zatím obsahuje pouze název a rok vydání, takže použití několika proměnných jako na ukázce výše se nyní nezdá jako špatný nápad. V reálné aplikaci by ale naše modelová třída `Kniha` mohla obsahovat desítky dalších členů (např. jazyk, autora ilustrací, počet stran atd.) a nemuseli bychom být omezeni pouze na knihy dvě ale i tisíce. S takovými čísli bez použití OOP by byla složitost naší aplikace neúnosná.

# Závěr

Objektově orientované programování (**OOP**) patří mezi základní znalosti každého programátora. **Objekty** tvoříme vždy na základě obecných předpisů (tzv. **tříd** - `class`). Z jedné třídy může vzniknout mnoho objektů, které jsou na sobě nezávislé a mají své vlastní data a vnitřní stav (enkapsulace).

# Objektově orientované programování

## Úvod

V této kapitole si ukážeme, jak vytvářet a pracovat s objekty.

## Třídy

Jsou základním stavebním kamenem objektového programování, kde jsou definované části, které se soustředí na konkrétní úkoly. Výhodou objektového programování je přehledné rozdělení programu na menší celky, jejich možnou recyklace napříč různými projekty (pokud budu mít třídu, která dokáže spočítat objem těles pro program pro Matematiku, mohu tuto třídu recyklovat i pro program pro Fyziku), program se zároveň stává přehlednější a tím se dodržují programovací konvence.

Třída je obecný předpis nějakého celku, například člověk je tvořený různými vlastnostmi (má nějakou výšku, hmotnost, barvu vlasů a nějaký den se narodil) a má nějaké funkce (umí mluvit, rozpoznat barvy). Z takového předpisu lze tvořit objekty / instance, což jsou již konkrétní lidé vycházející z obecného předpisu.

Zde je ukázka předpisu člověka s jeho vlastnostmi a funkcemi v programovacím jazyce C#

```
class Human
{
  public string name;
  public DateTime dateOfBirth;
  public void Mluv()
  {
    Console.WriteLine("Ahoj");
  }
}
```

V ukázce výše je vidět jak se třída vytváří. V jejím těle je několik proměnných a jedna funkce.

Aby bylo možné využít proměnné a funkce, je potřeba vytvořit z této třídy objekt.

Tímto způsobem jsme vytvořili objekt "adam" ze třídy Human

```
Human adam = new Human();
```

Naše třída Human obsahuje i několik proměnných, které jsou v současné době nenaplněné. Pro každou proměnnou je důležité vytvořit pravidlo na způsob ukládání takových dat.

> **Třída** - předpis, **Instance** - objekt vytvořený z předpisu

**# Konstruktor**

Jedním ze způsobů jak uložit do proměnných hodnoty je použití konstruktoru, konstruktor je předpis událostí, které se provedou v případě vytváření objektu. Chová se a vypadá jako metoda, ale nedokáže vracet žádnou hodnotu. Pokud chceme využít konstruktor, musí se jmenovat jako jeho třída ve které je použitý.

Využití konstruktoru pro získání jména a následně nastavení data narození na aktuální čas

```
class Human
{
  public string name;
  public DateTime birthDay;
  public Human(string name)
  {
    this.name = name;
    birthDay = DateTime.Now;
  }
}
```

Pokud teď budeme vytvářet instanci třídy, budeme muset vložit argument.

Nyní je uloženo v objektu adam v proměnné name hodnota "Adam" a nastaven datum na aktuální čas vytvoření

```
Human adam = new Human("Adam");
```

**# Vlastnosti**

Je člen který se chová podobně jako proměnná, ale na rozdíl od proměnné se vlastnost definuje pomocí bloku, který obsahuje proceduru *Get* nebo *Set*, případně jednu z nich. Používají se jako veřejné datové členy, u kterého chceme ovlivnit možnost čtení, nebo zápisu.

> Vlastnost se vždy označuje velkým písmenem nebo CamelCase

Deklarace vlastnosti: 1. řádek vlastnost umožňuje jak zápis, tak čtení 2. řádek vlastnost navenek umožňuje pouze čtení a uvnitř třídy je možné nastavit

```
class Human
{
  public string Name { get; private set }
  public string Data { get; set; }
}
```

**# Get & Set**

Jedná se o proceduru vlastnosti, nebo také přistupující objekty vlastnosti. Tvoří blok pro inicializaci vlastnosti. Díky těmto objektům je možné provádět různé úkony v průběhu čtení, nebo zápisu dat do této vlastnosti, nebo je možné tyto přístupy omezit.

V tomto příkladu se zaznamenává čas přistupu a nastavení hodnoty vlastnosti "Data"

```
class Human
{
  public DateTime lastView;
  public DateTime lastEdit;
  public string data;
  public string Name { get; private set; }
  public string Data
  {
    get
    {
      lastView = DateTime.Now;
      return data;
    }
    set
    {
      lastEdit = DateTime.Now;
      data = value;
    }
  }
}
```

Pro fungování tohoto řešení je potřeba, aby vedle vlastnosti byla vytvořená totožná proměnná. Procedura *Get* musí vracet nějakou hodnotu a procedura *Set* musí přijmout a uložit hodnotu. Pokud by jsme použili pouze jednu proměnnou, došlo by k zacyklení. Pokud budu zapisovat nějaká data, zavolá se *Setter*, v tomto settru je ale právě zápis do proměnné a tedy znovu se bude volat *Setter* a tímto způsobem dojde k zacyklení.

Chybný způsob použití vlastnosti a její procedury Get & Set, která vede k zacyklení

```
public string Data
{
  get
  {
    lastView = DateTime.Now;
    return Data;
  }
  set
  {
    lastEdit = DateTime.Now;
    Data = value;
  }
}
```

# Dědičnost

Vedle zapouzdření a polymorfismu je dědičnost dalším z pilířů OOP a slouží k recyklaci starých datových struktur, díky kterým vytvoříme nové podřazené datové struktury. Díky dědičnosti se také ušetří spousta repetitivní práce. Dědičnost v programování je podobná, jako dědičnost například u lidí. Naši předci měli nějaké společné rysy a ty dědíme, vycházíme tedy z nějakého předpisu a během života získáváme vlastní zkušenosti a vlastnosti, které mohou zdědit naši potomci.

Pokud budeme chtít vytvořit třídy různých typů uživatelů v případě bez možnosti dědičnosti vypadal by kód zhruba takto

Ani to nemusíte číst stačí se kouknou na délku kódu

```
class User
{
  public string name;
  public string password;
  public string position;
  public bool Log = false;
  public int age;
  public bool LogIn(string password)
  {
    if (this.password == password)
    {
      Log = true;
```

```
      return true;
    }
    else { return false;}
  }
  public void WriteFile()
  {
    ///
  }
}
class Admin
{
  public string name;
  public string password;
  public string position;
  public bool Log = false;
  public int vek;
  public bool LogIn(string password)
  {
    if (this.password == password)
    {
      Log = true;
      return true;
    }
    else { return false; }
  }
  public void WriteFile()
  {
    ///
  }
  public void DeleteFile()
  {

  }
  public void AddFile()
  {

  }
}
```

}
Ale "naštěstí" C# umí i dědičnost a tím dokážeme takovéto věci řešit mnohem efektivněji. Jak jsme si řekli, dědičnost je vlastně sdílení podobností a to přesně můžeme použít zde, kdy každý administrátor je zároveň i uživatelem, jen **Admin** umí trochu více věci.

Dědičnost v programování děláme pomocí znaku ' : '  (dvojtečka)

Zde je vidět značná úspora programování a tedy i času a zároveň se kód stává kratší (nemusí se stát přehlednějším)

```
class User
{
  public string name;
  public string password;
  public string position;
  public bool Log = false;
  public int age;
  public void LogIn(string password)
  {
    if (this.password == password)
    {
      Log = true;
      return true;
    }
    else{ return false; }
  }
  public void WriteFile()
  {
    ///
  }
}
class Admin : User
{
  public void DeleteFile()
  {

  }
  public void AddFile()
  {

  }
}
```

V ukázkách výše je vidět jasná úspora místa a program dělá to samé. V případě druhé ukázky **Admin** získal schopnosti a vlastnosti od **User**, tedy přihlásit se a psát do souboru, navíc má další vlastnosti které však **User** nemá. Vytváří se tím i určitá hierarchie.

> Každý **Admin** je **User**, ale né každý **User** je **Admin**.

Díky této hierarchii je možné **Admina**, přetypovat na **Usera**, ale nikoliv naopak. Při přetypování však **Admin** přijde o vše co uměl navíc. Jelikož každý vytvořený objekt, je zároveň referenčním datovým typem, je možné ho přetypovat, ale je potřeba dodržovat pravidla nadřazených a podřazených tříd.

V případě dědění může dojít k určité nejasnosti a to v případě konstruktoru nadřazené třídy. I na to je však myšleno, využívá se k tomu klíčové slovíčko již nám známá ' : ' a nové klíčové slovíčko **base**, které odkazuje na nadřazenou třídy. Je tedy nutné, aby oba měli konstruktor.

Tímto způsobem zajistíme, aby vše proběhlo v pořádku a třídy si mezi sebou předali data pro konstruktor

```
class User
{
  public string name;
  public User(string name)
  {
    this.name = name;
  }
}
class Admin : User
{
  public Admin(string name) : base(name)
  {
    this.name = name;
  }
}
```

**# Zapouzdření**

V některých případech však budu chtít omezit přístup k proměnným, vlastnostem nebo funkcím třídy, včetně podřazených tříd, nebo jen umožnit přístupu dědicům. K tomu je potřeba znát **modifikátory přístupu**.

> Public - veřejná (přístup mimo třídu včetně dědiců)

>

Private - soukromá (přistup pouze uvnitř třídy)

Protected - chráněná (přístup pouze uvnitř třídy a dědicům)

Static - statická (přístup pouze statickým prvkům a nelze je volat z instance)

# Maturita 2023

Tento dokument slouží jako podpůrný materiál pro maturanty v roce 2023. Obsahuje užitečné informace ke složitějším maturitním otázkám. Koukejte si to pořádně pročíst!

# Otázka 1: Césarova šifra

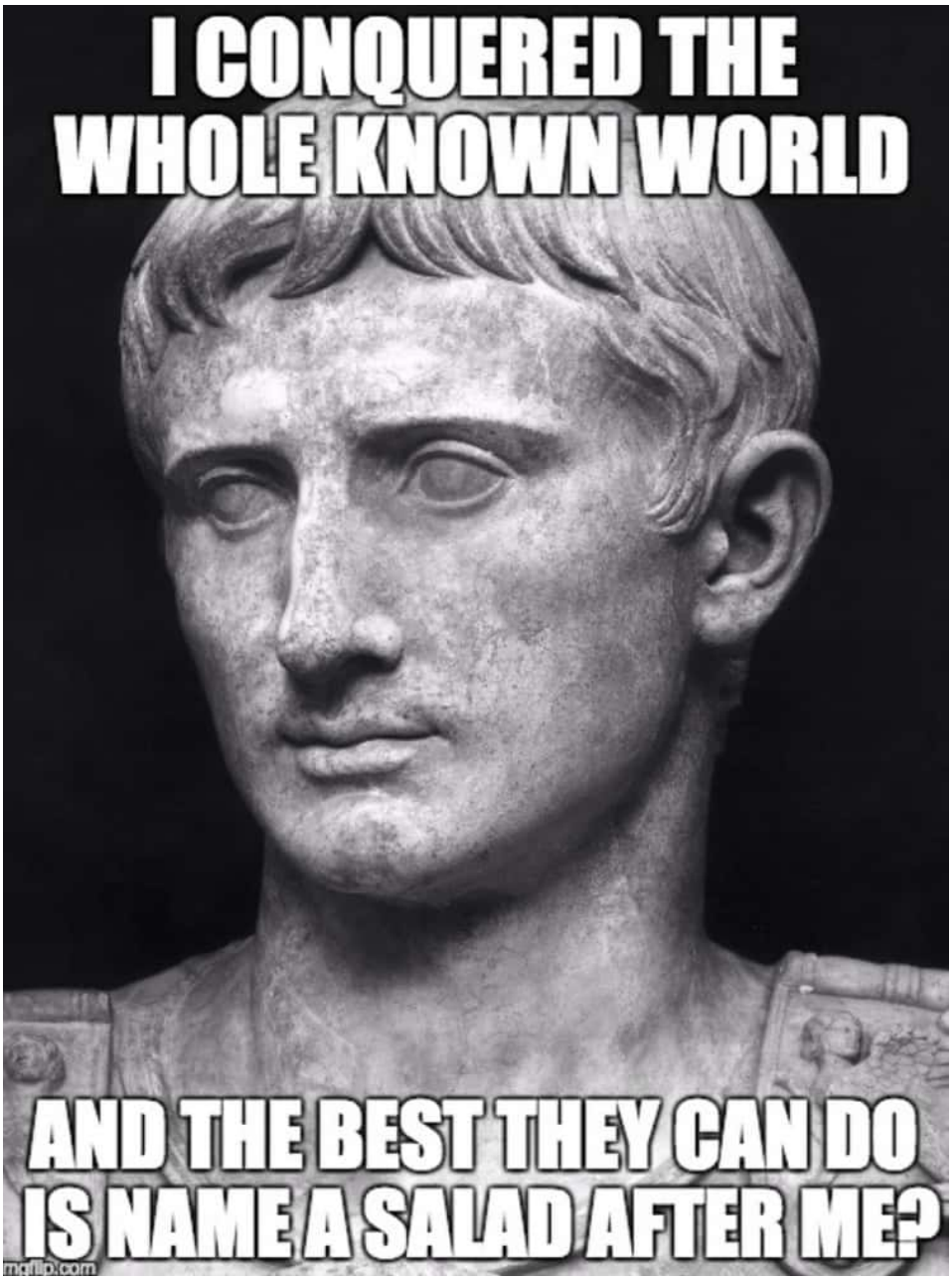
`Cézarova šifra` je starověká metoda kódování zpráv pomocí posunu písmen o určitý počet míst v abecedě. Například, pokud si zvolíme posun o 3 místa, písmeno `A` se nahradí písmenem `D`, písmeno `B` písmenem `E` atd. Tento způsob kódování se používal již v době Římské říše. V programovacím jazyce C# můžeme implementovat Cézarovu šifru jednoduše pomocí cyklu a jednoduchých matematických operací.

Ukázka implementace šifry

```
string message = "Tajna zprava"; // zpráva, kterou chceme zašifrovat
int shift = 3; // počet míst, o které se mají posunout písmena v abecedě
string encryptedMessage = ""; // inicializace prázdného řetězce pro zašifrovanou zprávu

foreach (char letter in message)
{
    int asciiCode = (int)letter; // převod písmene na ASCII kód
    int shiftedAsciiCode = asciiCode + shift; // posunutí ASCII kódu o zvolený počet míst
    char shiftedLetter = (char)shiftedAsciiCode; // převod posunutého ASCII kódu zpět na
písmeno
    encryptedMessage += shiftedLetter; // přidání posunutého písmene do zašifrované zprávy
}

Console.WriteLine(encryptedMessage); // výpis zašifrované zprávy
```




*Nějaký humřík na uvolnění :D*

# Otázka 3: Vigenèrova šifra

`Vigenèrova šifra` je typ šifrování, který používá `klíč` složený z řetězce písmen k zakódování zprávy. Každé písmeno zprávy je posunuto o určitý počet míst na základě odpovídajícího písmene v klíči. Tento posun se opakuje pro každé písmeno v zprávě, dokud není celá zpráva zakódována.

Vigenèrova šifra je podobná Cézarově šifře, ale namísto pevného posunu používá proměnlivý posun pro každé písmeno v zprávě, což ji činí mnohem bezpečnější.

# Otázka 5: Metoda Monte Carlo

Metoda `Monte Carlo` v programování je způsob, jak přibližně vypočítat výsledek nějakého problému, aniž bychom museli použít přesné matematické rovnice. **Namísto toho používáme náhodná čísla a statistickou analýzu, abychom získali odhad výsledku**. Tento přístup se používá v mnoha různých oblastech, kde je těžké nebo nemožné najít přesné matematické řešení.

Jako příklad si ukážeme program, který aproximuje hodnotu čísla `π` pomocí náhodně vygenerovaných bodů v čtverci a kruhu:

Metoda Monte Carlo

```
using System;

class MonteCarloPi
{
    static void Main(string[] args)
    {
        int n = 1000000; // počet náhodných bodů
        int count = 0;

        Random rnd = new Random();

        for (int i = 0; i < n; i++)
        {
            double x = rnd.NextDouble(); // náhodná hodnota x od 0 do 1
            double y = rnd.NextDouble(); // náhodná hodnota y od 0 do 1
            if (x * x + y * y <= 1) // bod je uvnitř kruhu
            {
                count++;
            }
        }

        double pi = 4.0 * count / n; // aproximace hodnoty pi

        Console.WriteLine("Approximation of pi: " + pi);
    }
}
```

> Aproximace je matematický postup, při kterém se snažíme najít hodnotu nebo řešení daného problému, aniž bychom museli použít přesné matematické vztahy nebo algoritmy. Namísto toho používáme jednodušší a rychlejší metody, které nám umožňují dosáhnout přibližného výsledku, který je dostatečně přesný pro naše účely.

# Otázka 6: Eratostenovo síto

Eratostenovo síto je matematická metoda pro nalezení všech prvočísel menších nebo rovných zadanému číslu. Princip spočívá v tom, že se nejprve vytvoří seznam všech čísel od 2 do zadaného čísla a postupně se odstraňují čísla, která nejsou prvočísly.

Algoritmus funguje takto:

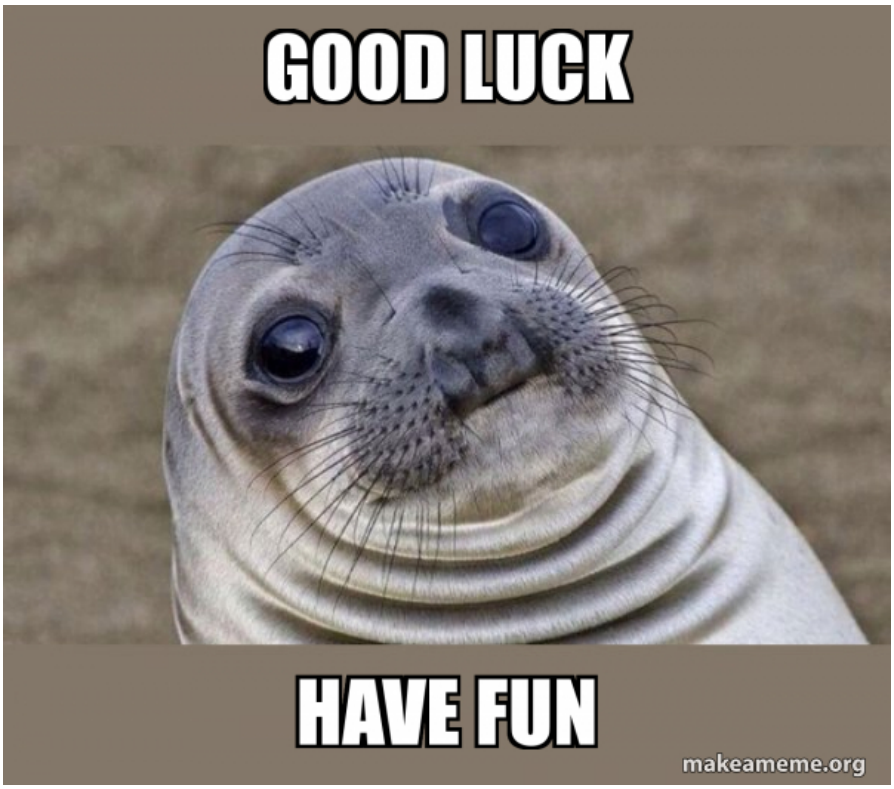
- Vytvoř seznam všech čísel od 2 do zadaného čísla.
- Začni s prvním prvočíslem (tj. číslem 2) a vyškrtni z seznamu všechna násobky tohoto čísla (tj. čísla 4, 6, 8 atd.).
- Přejdi na další nevyškrtnuté číslo v seznamu a opakuj krok 2 až do konce seznamu.
- Zbývající nevyškrtnutá čísla jsou všechna prvočísla menší nebo rovna zadanému číslu.

Například, pokud chceme najít všechna prvočísla menší nebo rovna číslu 30, postupujeme následovně:

- Vytvoříme seznam všech čísel od 2 do 30: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30.
- Vyškrtneme všechny násobky čísla 2 (tj. 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30).
- Přejdeme na další nevyškrtnuté číslo v seznamu (tj. 3) a vyškrtneme všechny jeho násobky (tj. 9, 15, 21, 27).
- Přejdeme na další nevyškrtnuté číslo (tj. 5) a vyškrtneme všechny jeho násobky (tj. 25).
- Postupujeme takto až do konce seznamu.
- Zbývající nevyškrtnutá čísla jsou všechna prvočísla menší nebo rovna číslu 30: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

# Závěr

Je důležité ke každé otázce znát alespoň teorii! Hodně štěstí!


GOOD LUCK
HAVE FUN
makeameme.org

*:-)*