# Funkce

Funkce jsou nejdůležitější částí programování. Jedná se o samostatný kus kódu, který je nějak označen (často pouze jménem) a za úkol má vzít vstupní údaje, ty zpracovat a vrátit nějaké výstupní data. Funkce jsme již použité viděli - používají se ve WPF, v konzoli a dokonce jsme je již i volali.

Funkce slouží k zjednodušení kódu a přehlednosti. Funkce můžeme totiž opakovaně využívat. Pro funkce musíme definovat nové klíčové slova:

- `return` - velmi podobný `break`, slouží k ukončení funkce a navrácení hodnoty volání. Nemusíme určovat nic (viz. níže).
- `void` - znamená nic a reprezentuje návratový typ funkce, který nic nevrací.

Funkci deklarujeme pomocí návratového typu, jeho jména, kulatých závorek (kde mohou být parametry) a těla v tomto pořadí, např.:

```
bool PrintHeader() // funkce PrintHeader nepřijímá parametry a vrací návratem datový typ bo
{
    Console.BackgroundColor = ConsoleColor.Red;
    Console.ForegroundColor = ConsoleColor.White;
    Console.WriteLine("~~~~~~~~~~~~~~~~~~~");
    Console.WriteLine("~~~~   SSPŠ   ~~~~");
    Console.WriteLine("~~~~~~~~~~~~~~~~~~~");

    Console.ResetColor();

    return true;
}

PrintHeader(); // funkce zavolána, tady se návratová hodnota zahodí
bool success = PrintHeader(); // uloží se návratová hodnota
```

Pokud funkci určíme nějaký návratový typ mimo `void`, je nutné, aby každá cesta programu nějaký výstup daného datového typu vracela. Když použijeme `void`, nemůžeme vrátit jakoukoliv hodnotu a je nutné volat jen samotný return:

```
void DeleteFile() {
    if(fileNotExists) {
        return;
    }
```

```
    File.Delete();
}
```

Parametry určujeme tak, že dovnitř kulatých závorek umístíme datový typ a jméno parametru. Dané jméno je poté slovem, které se dá použít uvnitř funkce. Při více parametrech je určujeme stejným způsobem, ale rozdělujeme je čárkou (`,`).

```
bool PrintHeader(string name) // jeden parametr, jméno name, typ string.
{
    Console.BackgroundColor = ConsoleColor.Red;
    Console.ForegroundColor = ConsoleColor.White;
    Console.WriteLine("~~~~~~~~~~~~~~~~~~~");
    Console.WriteLine("~~~~    " + name + "    ~~~~"); // použijeme parametr
    Console.WriteLine("~~~~~~~~~~~~~~~~~~~");

    Console.ResetColor();

    return true;
}

PrintHeader("SSPŠ", false); // Nejde! Funkce má jeden parametr.
PrintHeader("SSPŠ");
```

```
bool PrintHeader(string name, bool reset)
{
    Console.BackgroundColor = ConsoleColor.Red;
    Console.ForegroundColor = ConsoleColor.White;
    Console.WriteLine("~~~~~~~~~~~~~~~~~~~");
    Console.WriteLine("~~~~    " + name + "    ~~~~");
    Console.WriteLine("~~~~~~~~~~~~~~~~~~~");

    if(reset)
        Console.ResetColor();

    return true;
}

PrintHeader("SSPŠ", false);
```

V případě, že používáme funkce mimo top-level-statements, je nutné funkce označovat pomocí klíčového slova static. O tomto si povíme později. Např.:

```
static string GetHeader(string name)
{
    string output = "";

    output += "~~~~~~~~~~~~~~~~~~\n";
    output += "~~~~    " + name + "    ~~~~\n";
    output += "~~~~~~~~~~~~~~~~~~\n";

    return output;
}
```

---

Některým parametrům můžeme nastavit základní hodnoty, které se doplní tehdy, když jej neuveden u volání. Např.:

```
void Print(string header = "SSPŠaG")
{
    Console.WriteLine($" ~~~ {header}  ~~~");
}

Print(); // v "header" bude "SSPŠaG"
Print("CAJTHAML"); // v "header" bude "CAJTHAML"
```

Níže uvedený kód avšak fungovat nebude - nepovinné parametry musí být na konci všech parametrů a může jich být více. Důvod pro to, že nemohou být v pořadí je takový, že bychom nevěděli při zadání hodnoty, zda se myslí parametr se základní hodnotou a nebo nějaký následující.

```
void Print(string header = "SSPŠaG", string footer)
{
    Console.WriteLine($" ~~~ {header}  ~~~");
    Console.WriteLine($" ~~~ {footer}  ~~~");
}

Print();
Print("footer");
Print("header", "footer");
```

# Funkce v konzoli

Funkce můžeme začít používat ihned v konzolových aplikacích. Níže je uveden kód, který reprezentuje, jak by mohla vypadat správně rozdělená aplikace. Všímejte si především přehlednosti kódu:

```
static void Main(string[] args)
{
    bool running = true;

    while(running)
    {
        running = Process();
    }
}

static bool Process()
{
    Console.Clear();
    PrintHeader();

    uint status = ProcessMenu();

    switch(status)
    {
        case 1:
            {
                // ProcessUsers();
                break;
            }
    }

    return status != 9;
}

static uint ProcessMenu()
{
    PrintMenu();
```

```
    string selection = Console.ReadLine();

    uint number;
    if (!uint.TryParse(selection, out number))
        return 99;

    return number;
}

static void PrintMenu()
{
    Console.WriteLine("1) Users");
    Console.WriteLine("2) Products");
    Console.WriteLine("3) Inventory");
    Console.WriteLine("9) Exit");

    Console.WriteLine("");

    Console.Write("Selection: ");
}

static void PrintHeader()
{
    Console.BackgroundColor = ConsoleColor.Red;
    Console.ForegroundColor = ConsoleColor.White;
    Console.WriteLine(" ~~ MANAGMENT ~~ ");

    Console.ResetColor();

    Console.WriteLine("");
}
```

# Objektově orientované programování

Objektově orientované programování je jeden ze způsobu návrhu a psaní aplikací. Odborně se tomuto říká programovací paradigmata a existují další paradigmata:

- **objektově orientované programování** (OOP)
- datově orientované programování
- funkcionálně orientované programování

Jeden jazyk může podporovat více paradigmat. OOP je nejvíce používané paradigma, protože má nejméně nevýhod. Pro pochopení dalších paradigmat se OOP považuje za nejdůležitější a proto se ho učíme.

Myšlenky OOP jsou takové, že obyčejné programy jsou nepřehledné a těžko se spravují. My pomocí tříd a dalších funkcionalit rozdělíme kód na menší části, které často reprezentují reálné objekty. Tyto balíčky se poté znovu opakovaně používají.

Základy OOP se skládají z:

- tříd — `class`
- instancí / objektů — `instance` / `object`
- metod — `method`

Ty si nyní popíšeme.

# Třídy a instance

Třída je nějaká šablona či formulář. Každá třída má jméno a vlastnosti, které popisuji, co se uvnitř nachází. Často mají nějaký význam k reálným objektům (obchod, myš, hráč, kostka, ...). Vlastnosti jsou vlastně proměnné a každá věc, která z této třídy vytváří se, je bude mít uvnitř. Stejně jako proměnné musí mít jméno, datový typ a nyní může být modifikátory.

Třídu vytváříme pomocí klíčového slova class. Např.:

```
class User // jméno třídy
{
    // Vlastnosti:
    string name; // vlastnost name datového typu string
    string email;
    string password;
    ushort age;
    string role;
}
```

Po definici třídy se daný název třídy stane datovým typem a můžeme je používat při definici proměnných či na jiných místech.

V našich programech budeme psát třídy a vlastnosti pouze anglickými názvy. Názvy musí být konzistentní (PascalCase, camelCase, snake_case a další).

Rozdíl mezi třídami a strukturami je velmi viditelný v tom, že struktury nelze upravovat např. v cyklu a přímo a musíme je ukládat do proměnných. Třídy lze rovnou na místě upravovat a je s nimi obecně méně problémů.

---

Instance jsou již vyplněné šablony a reprezentuji již reálný objekt, který má vlastnosti. Např. máme-li třídu Kostka, tak instancí této třídy reprezentujeme již fyzický stav této kostky,

kterou máme. To znamená, že můžeme mít více kostek a každá je jasně popsaná a má své proměnné.

Instanci v C# známe již ze struktur a vytváříme jí stejně pomocí klíčového slova new. Při vytvoření můžeme určit nějaké vlastnosti a nebo ke všem přistupovat:

```
class User
{
    public string name;
    public string email;
    public string password;
    public ushort age;
    public string role;
}

User user = new User();
user.name = "John";

Console.WriteLine(user);
Console.WriteLine(user.name);
Console.WriteLine(user.password);
Console.WriteLine(user.age);
```

Všechny vlastnosti musí být prozatím označené pomocí `public`, proč to tak je, si vysvětlíme později.

# Metody

Metody jsou funkce uvnitř tříd. Jedná se o stejné definice jako dříve. Metody však mají přístup rovnou k dané instanci a mohou přistupovat na dané data z instance.

Metody se používají na akce, které se dějí na dané instanci. Metodami určujeme akce, které se s instancí dějí, např. pohyb, změnou a čehokoliv dalšího. Používáme je proto, aby věci, které s objektem souvisí, byly právě definovány u objektu a ne nikde v kódu.

Ukázková definice metod:

```
class User
{
    public string name;
    public int age;

    public void Age()
    {
```

```
        age++;
    }

    public string Introduce()
    {
        return $"Hello, I am {name} and I am {age} years old!";
    }
}

User user = new User();
user.name = "John";

Console.WriteLine(user.Introduce()); // "Hello, I am John and I am 0 years old!"

user.Age();
Console.WriteLine(user.Introduce()); // "Hello, I am John and I am 1 years old!"

user.Age();
user.Age();
user.age = 50;
Console.WriteLine(user.Introduce());  // "Hello, I am John and I am 50 years old!"
```

V metodách můžeme stejně definovat parametry, základní hodnoty parametrů, návratové typy a další. Všechny metody můžeme označovat modifikátory, aktuálně musíme pomocí `public`.

---

Uvnitř tříd často používáme klíčové slovo `this`, které referuje na aktuální instanci. Nejčastěji ho používáme v případě kolize parametru s proměnnou, např.:

```
class User
{
    public string name;

    public string Greet(string name)
    {
        // this.name = vlastnost == "John"
        // name = parametr == "Greg"
        return $"Hello, I am {this.name}, nice to meet you {name}!";
    }
```

```
}

User user = new User();
user.name = "John";
Console.WriteLine(user.Greet("Greg"));
```

Uvnitř metod můžeme volat další metody:

```
public void Sign()
{
    this.signature += this.userFirstName;
}

public void Fill(string userFirstName)
{
    this.userFirstName = userFirstName;

    this.Sign();
}
```

# Modifikátory

Modifikátory modifikují určitou část kódu. Prozatím se pro nás jedná o vlastnosti, metody a třídy. Nějaké modifikátory jsme již viděli a nyní si je popíšeme. Modifikátory zapisujeme před definicí dané věci. Metody, vlastnost i třída může mít několik modifikátorů najednou.

# Modifikátor viditelnosti

Určuje, odkud je daná vlastnost či metody viditelná ze třídy. Máme tři hodnoty:

- **public** — viditelná odkudkoliv, i mimo třídu
- **private** — viditelná pouze uvnitř třídy
- **protected** — viditelná ve třídě, a třídách, které z této třídy dědí (viz. dědičnost)

Privátní vlastnosti a metody se používají tehdy, když nechceme, aby viděl daný stav kdokoliv z venku. Např.:

```
class Car {
    public string licencePlate;
    public string color;

    private List<Item> items; // Jsou k dispozici jen v této třídě
```

```
    public void AddItem(Item item)
    {
        items.Add(item);
    }

    private void RemoveItem(Item item)
    {
        items.Remove(item);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");

        Car car = new Car();
        car.licencePlate = "ABC-123";
        car.color = "red";

        // car.items = new List<Item>(); // NEJDE
        // car.RemoveItem(new Item()); // NEJDE
    }
}
```

Když modifikátor viditelnosti neurčíme, znamená to, že je daná věc automaticky privátní.

# Konstruktor

Konstruktor je speciální metoda. Tato metoda se volá, když vytváříme instanci třídy. Samotné kulaté závorky za jména třídy při vytváření je volání tohoto konstruktoru. Když konstruktor neexistuje, nezavolá se žádná metoda.

Konstruktor tedy můžeme mít různé parametry. Konstruktory používáme většinou pro nastavení správného stavu instance tak, aby byla již při vytvoření ve správném tvaru. Např. můžeme předávat různé parametry, které ihned nastavíme a tedy si uživatel může být jistý, že instance bude funkční.

Konstruktory mohou mít modifikátory a jejich jméno vždy odpovídá jménu třídy. Např.:

```
class Tower
{
```

```
    public string type;
    public Location position;
    public uint range;

    public Tower(string type, Location position, uint range) // konstruktor se třemi parame
    {
        this.type = type;
        this.position = position;
        this.range = range;
    }
}

// Definice Location

Tower tower = new Tower("bomb", new Location(10, 4), 4);
Console.WriteLine(tower.range); // 4
Console.WriteLine(tower.type); // "bomb"
```

Třída může mít více konstruktorů, musí mít však různé parametry:

```
class Tower
{
    public string type;
    public Location position;
    public uint range;

    public Tower(string type, Location position, uint range)
    {
        this.type = type;
        this.position = position;
        this.range = range;
    }

    public Tower(Location position, uint range)
    {
        this.type = "basic";
        this.position = position;
        this.range = range;
    }
```

```
    public Tower(Location position)
    {
        this.type = "basic";
        this.position = position;
        this.range = 1;
    }
}

new Tower("spike-generator", new Location(1, 2), 3);
new Tower(new Location(5, 2), 5);
new Tower(new Location(12, 5));
```

---

Veřejný konstruktor znamená, že instanci ze třídy můžeme vytvořit kdykoliv. Když bychom vytvořili privátní konstruktor, znamenalo by to, že daný konstruktor lze pro vytvoření instance použít pouze uvnitř dané třídy (např. v metodě).

# Slovníky

Slovníky jsou datová struktura, která je velmi podobná listu. List ukládá věci za sebou a můžeme je libovolně přidávat a odebírat. List ukládá informace pod indexy (celé číslo). Slovníky oproti tomu místo ukládání informací pod čísly, je ukládají pod jakýmkoliv jiným datovým typem. Klíče uvnitř slovníku musí být unikátní, hodnoty nikoliv.

Ukládáme tedy pár klíče a hodnoty (key-value pair). Slovníky jsou připravené v knihovně `System.Collections.Generic`, kterou musím použít (pomocí `using`). Slovníky tvoříme pomocí `Dictionary<>`, kde do špičatých závorek dáváme dva datové typy, jeden pro klíče, jeden pro hodnotu. Na slovníku máme různé metody. Např.:

```
Dictionary<string, string> translations = new Dictionary<string, string>();

translations.Add("ahoj", "hello"); // přidání páru hodnoty "ahoj" a "hello"
translations.Add("čau", "hello");

Console.WriteLine(translations["ahoj"]); // získáme hodnotu, která je uchována pod klíčem "
```

Když se pokusíme do slovníku přidat již existující hodnotu, tak nastane výjimka. Proto máme k dispozici metodu `TryAdd`, která se pokusí přidat hodnotu s klíčem, pokud klíč existuje, tak se nic nestane. Další metody a funkčnosti:

---

```
translations.Remove("čau"); // odstraní ze slovníku hodnotu s klíčem
translations.Remove("xdlol"); // odstranění neexistující nevyhodí výjimku

translations.Clear(); // vyčistí celý slovník, nic v něm nezůstane

Console.WriteLine(translations.ContainsKey("ahoj")); // obsahuje klíč?
Console.WriteLine(translations.ContainsValue("hello")); // obsahuje hodnotu?
Console.WriteLine(translations.Count); // počet hodnot
Console.WriteLine(translations.Keys); // pole klíčů
Console.WriteLine(translations.Values); // pole hodnot
```

---

Použití slovníků:

```
class User
{
    string name;
    int id;

    public User(string name, int id)
    {
        this.name = name;
        this.id = id;
    }
}

Dictionary<int, User> users = new Dictionary<int, User>(); // pár čísla a uživatele
users.Add(1, new User("Pepa", 1));
```

```
Dictionary<List<int>, List<List<string[]>>> users =
              new Dictionary<List<int>, List<List<string[]>>>();
users.Add(
     new List<int> { 32, 1, 4 },
     new List<List<string[]>> {
         new List<string[]> {
             new string[] { "Yoko" }
```

```
        }
    });
```

# Vlákna

Vlákno je výpočetní jednotka programu. Používáme je pro paralelní přístup k programování, tedy, aby náš program využíval všechny možnosti počítatče. Paralelní programování je však velmi náročné a pro většinu věcí nám stačí vlákno jedno. S jedním vláknem jsme všichni již seznámení. Když nám běží kód v konzoli, tak mám jen jedno vlákno. Jedinou možností, co s tímto vláknem můžeme teď dělat, je to, že jej můžeme na nějakou dobu zastavit. Proto využíváme metodu `Sleep`, která uspí aktuální vlákno na určitý počet milisekund (1000ms=1s):

```
Thread.Sleep(50);
```

# Dědičnost

Dědičnost je způsob, jak můžeme v programování zdědit funkcionality a vlastnosti nějaké jiné třídy a např. do nich přidat vlastní funkcionalitu, nebo nové vlastnosti. Velkou výhodou dědičnost je to, že nově děděná třída je vlastně pořád ta třída, ze které dědíme a tedy tak k ní můžeme v kódu přistupovat pomocí přetypování. Všechny třídy dědí ze třídy Object.

Dědění v kódu zapisujeme při deklaraci třídy tím, že za její jméno dáme dvojtečku a jméno třídy, ze které chceme dědit. V C# můžeme dědit pouze z jedné třídy. Např.:

```
class User
{
    public string name;
    public string password;
    public DateTime createdAt;
}

class Administrator : User // Administrátor dědí z uživatele
{
    public void ManageStuff()
    {
        Console.WriteLine($"{name} did manage stuff.");
    }
}

List<User> users = new List<User>();
User user = new User();
user.name = "Matěj Cajthaml";
```

```
Administrator administrator = new Administrator();
administrator.name = "Yoko Cajthaml";

users.Add(user);
users.Add(administrator);

foreach(User u in users)
{
    Console.WriteLine(u.name);
}

Console.WriteLine(user.ManageStuff()); // nelze!
Console.WriteLine(administrator.ManageStuff());
Console.WriteLine(administrator.name);

Console.ReadKey();
```

---

Samotné třídy se používají k zpřehlednění kódu a proto existuje i dědičnost. Místo toho, abychom stejné vlastnosti a metody opakovali v různých třídách. Nevýhoda dědičnosti je to, že při velkém zanořování dědičnosti (A > B > C > ...) začne být dědičnost a třídy nepřehledné.

---

Při dědičnosti se můžeme potkat s problémem ohledně konstruktorů. Konstruktory slouží k tomu, abychom měli jisté, co se bude v dané třídě nacházet již při vytvoření. V případě použití dědění však vůbec nadřazený konstruktor v případě nového konstruktoru nevoláme, a tím pádem je jen aktuální třída ve správném formátu. Proto máme k dispozici klíčové slovo base, které slouží k zavolání konstruktoru z třídy, ze které dědíme.

```
class Enemy {
    protected Location location;
    protected string name;

    public Enemy(Location location, string name) {
        this.location = location;
        this.name = name;
    }
}
```

```
class Zombie : Enemy {
    protected string type;
    protected int infected;

    public Zombie(Location location, string name, string type, int infected) : base(locatio
        this.type = type;
        this.infected = infected;
    }
}

Zombie zombie = new Zombie(new Location(0, 0), "Jyohn", "Villager", 0);
```

Volání base může zavolat konstruktor, který zavolá jiný nadřazený konstruktor, a tak dále.

# Abstrakce

Abstrakce nám dovoluje pracovat s třídami jako kdyby byly jiná třída. Můžeme přistupovat k nadřazené třídě, pokud jsme si jistí, že to daná třída vůbec je. Nebo naopak, můžeme přistupovat níže a volat metody, když si vůbec nejsme jistí reálným typem třídy.

Třídy mezi sebou umíme přetypovávat:

```
class User {}
class Administrator : User {}

User u1 = new User();
User u2 = new User();
Administrator a1 = new Administrator();

User adminAsUser = (User) a1;

// Co nastane?
Administrator userAsAdministrator = (Administrator) u1; // Nepůjde - objekt není Administra
Administrator adminBackAsAdmin = (Administrator) adminAsUser; // Funguje! Protože objekt je
```

Datové typy můžeme i zkontrolovat:

```
List<User> users = new List<User>();
User user = new User();
user.name = "Matěj Cajthaml";
```

```
Administrator administrator = new Administrator();
administrator.name = "Yoko Cajthaml";

users.Add(user);
users.Add(administrator);

foreach(User u in users)
{
    Console.WriteLine(u.name);

    if(u is Administrator)
    {
        ((Administrator)u).ManageStuff();
        // stejné jako:
        // Administrator admin = (Administrator)u;
        // admin.ManageStuff();
    }
}
```