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)

Revision #1
Created 2025-05-21 13:09:30 UTC by Magdalena Dobešová
Updated 2025-05-21 13:31:10 UTC by Magdalena Dobešová