Objektově orientovaný přístup

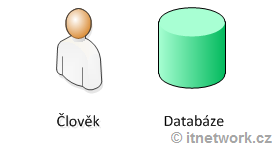
Jedná se o filozofii a způsob myšlení, designu a implementace, kde klademe důraz na **znovupoužitelnost**. Přístup nalézá inspiraci v průmyslové revoluci - vynález základních komponent, které budeme dále využívat (např. když stavíme dům, nebudeme si pálit cihly a soustružit šroubky, prostě je již máme hotové).

Poskládání programu z komponent je výhodnější a levnější. Můžeme mu věřit, je otestovaný (o komponentách se ví, že fungují, jsou otestovány a udržovány). Pokud je někde chyba, stačí ji opravit **na jednom místě**. Jsme motivováni k psaní kódu přehledně a dobře, protože ho po nás používají druzí nebo my sami v dalších projektech (přiznejme si, že člověk je od přírody líný a kdyby nevěděl, že se jeho kód bude znovu využívat, nesnažil by se ho psát kvalitně).

Znalosti, které jsme se doteď naučili, samozřejmě budeme používat dál. Náš kód budeme pouze jinak strukturovat a to do komunikujících objektů.

Jak OOP funguje

**Základní jednotkou je objekt**, který odpovídá nějakému objektu z reálného světa (např. objekt *člověk* nebo *databáze*).

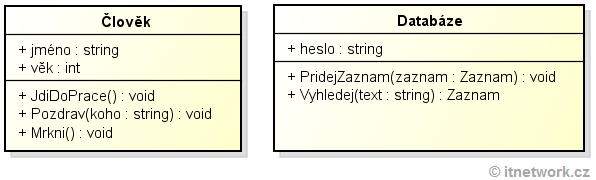


Objekt má své členské proměnné, vlastnosti a metody.

**Členské proměnné** objektu jsou data, která uchovává (např. u člověka jméno a věk, u databáze heslo). Jedná se o prosté proměnné, se kterými jsme již stokrát pracovali. Někdy o nich hovoříme jako o vnitřním stavu objektu.

**Vlastnosti** velmi často se nám stává, že chceme mít kontrolu nad změnami nějakého členské proměnné objektu zvenčí. Budeme chtít členskou proměnnou nastavit jako read-only nebo reagovat na jeho změny. Vlastnosti se navenek chovají jako veřejné členské proměnné, ale interně jsou tvořeny množinou **přístupových metod** **get** (selektor) a **set** (modifikátor), které realizují čtení a zápis vlastností.

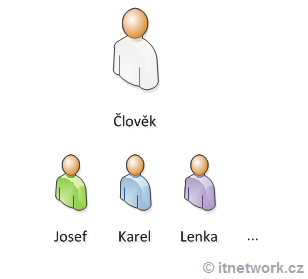
**Metody** jsou **schopnostmi**, které umí objekt vykonávat. U člověka by to mohly být metody: JdiDoPrace(), Pozdrav() nebo Mrkni(). U databáze PridejZaznam() nebo Vyhledej()). Metody mohou mít parametry a mohou také vracet nějakou hodnotu. Velmi dobře je známe, používali jsme např. metodu Split() na objektu string. String je vlastně třída, která reprezentuje v realitě nějaký text, se kterým pracujeme. Obsahuje metody, které řetězec umí vykonávat (kopírování, mazání, splitování...) a má také nějaké vlastnosti, např. Length, který značí jeho délku.

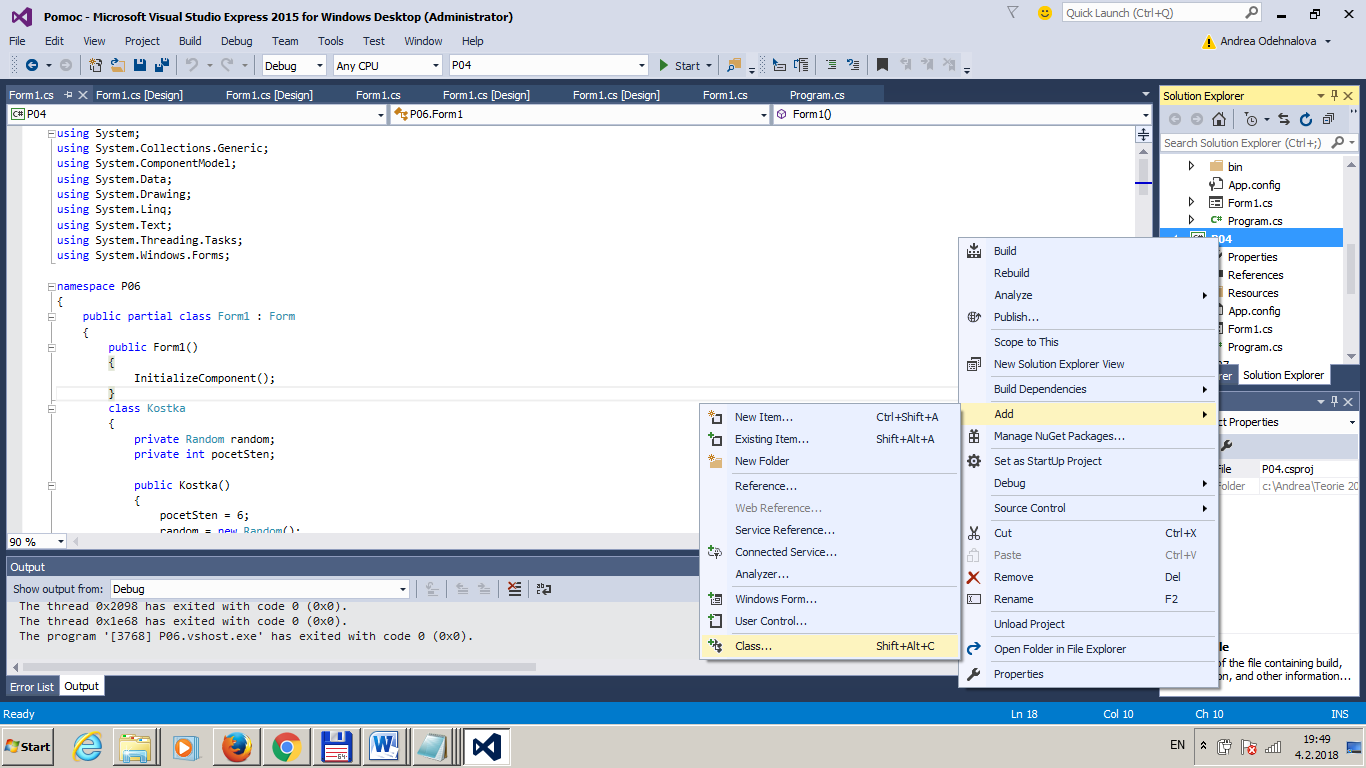


Třída

Třída je soubor příkazů. Třída je **vzor**, podle kterého se objekty vytváří. Definuje jejich členské proměnné, vlastnosti a metody.

Objekt, který se vytvoří podle třídy, se nazývá **instance**. Instance mají stejné **rozhraní** jako třída, podle které se vytváří, ale navzájem se liší svými daty. Mějme například třídu *Člověk* a od ní si vytvořme instance karel a josef. Obě instance mají jistě ty samé metody a členské proměnné, jako třída (např. jméno a věk) a metody (JdiDoPrace() a Pozdrav()), ale hodnoty v nich se liší (první instance má v členské proměnné jméno hodnotu "Karel" a ve věk 22, druhá "Josef" a 45).



Třídy by se měly programovat do samostatného souboru. Stiskněte pravé tlačítko myši na název projektu a přidejte soubor **Třída**. Třídu pojmenujte například Student.cs.

### Instance (= object)

* Na základě datového typu třída lze vytvořit její instanci
* Instanci typu třída označujeme jako **objekt**
* Objekt je vytvářen pomocí klíčového slova **new** – způsobí vyvolání konstruktoru

Např.:

Osoba osoba1 = new Osoba();

Zde je volán **implicitní konstruktor**, který má každá třída, aniž bychom jej museli programovat (implementovat).

* Každý objekt (instance) musí patřit k určité třídě. Nelze vytvořit objekt, aniž by nebylo definováno, jak má vypadat.
* Ze **třídy** může být vytvořen **libovolný počet objektů**. Podobně lze například podle výkresu vyrobit libovolný počet navržených součástek.
  + Všechny **instance** (objekty) patřící k jedné třídě mají **různá** svá **vlastní data (stav) –** členské proměnné, vlastnosti
  + Všechny **instance** (objekty) patřící k jedné třídě mají **stejné schopnosti** (**metody**)

## **Konstruktory, this**

**Konstruktory**

Konstruktory tříd jsou metody, jejichž pomocí se **vytvářejí objekty** příslušných tříd a **mohou inicializovat** data objektu – nastavit počáteční stav.

Konstruktor má **stejný název jako třída,** je vždy **public** a **nevrací žádnou hodnotu, ani void**.

Konstruktor má většinou **vícero přetížení**. Jednotlivá přetížení inicializují počáteční stav objektu různým způsobem.

Pokud nenadefinujeme žádný konstruktor, má třída k dispozici implicitní konstruktor, který objekt pouze vytvoří a data objektu ponechá **nulová**. Všechny členské proměnné jsou inicializovány na nulovou hodnotu.

Definujeme-li jakýkoli konstruktor, ztrácíme možnost používat implicitní konstruktor, pokud jej rovněž nenadefinujeme.

### this

Parametry konstruktorů mají často stejné názvy jako členské proměnné, které nastavují. Členské proměnné od parametrů odlišíme pomocí klíčového slova **this** - představuje aktuální objekt.

Např.:   
Následující třída Osoba má tři přetížení konstruktoru.

class Osoba

{

private String jmeno; //soukromá členská proměnná

private String prijmeni;

public Osoba()

{

}

public Osoba(string jmeno)

{

this.jmeno = jmeno;

}

public Osoba(string jmeno, string prijmeni)

{

this.jmeno = jmeno;

this.prijmeni = prijmeni;

}

}

Instanci třídy osoba vytvoříme pomocí **new**.

**OOP stojí na základních třech pilířích:**

* Zapouzdřenost
* Dědičnost
* Polymorfismus

Zapouzdřenost

Zapouzdření umožňuje **skrýt některé metody a členské proměnné** tak, aby zůstaly použitelné jen pro třídu zevnitř. Objekt si můžeme představit jako **černou skřínku** (blackbox), která má určité **rozhraní (interface)**, přes které jí předáváme instrukce/data a ona je zpracovává.

Nevíme, jak to uvnitř funguje, ale víme, jak se navenek chová a používá. Nemůžeme tedy způsobit nějakou chybu, protože využíváme a vidíme jen to, co tvůrce třídy zpřístupnil.

Příkladem může být třída *člověk*, která bude mít členské proměnné *datumNarozeni* a na jeho základě další členské proměnné: *plnolety* a *vek*. Kdyby někdo objektu zvenčí změnil *datumNarozeni*, přestaly by platit členské proměnné *plnolety* a *vek*. Říkáme, že vnitřní stav objektu by byl nekonzistentní. Toto se nám ve strukturovaném programování může klidně stát. V OOP však objekt zapouzdříme a členskou proměnnou  *datumNarozeni* označíme jako privátní, zvenčí tedy nebude viditelný. Naopak ven vystavíme metodu ZmenDatumNarozeni(), která dosadí nové datum narození do proměnné *datumNarozeni* a zároveň provede potřebný přepočet věku a přehodnocení plnoletosti. Použití objektu je bezpečné a aplikace stabilní.

Zapouzdření tedy donutí programátory používat objekt jen tím správným způsobem. **Rozhraní (interface) třídy rozdělí na veřejně přístupné (public) a vnitřní strukturu (private)**.

### Modifikátory přístupu

K nastavení přístupu k členských proměnných, vlastnostem a metodám slouží modifikátory přístupu:

* **private** - soukromé  
  k datům mají přístup pouze metody dané třídy.  
  Podle toho, které metody implementujeme, si **řídíme přístup k členským proměnným** (zda lze stav objektu pouze zjišťovat (číst) nebo i měnit (zapisovat)
* **protected -** chráněnéjsou přístupné pouze z metod danétřídy a z metod jejich **potomků**
* **public -** veřejnéjsou přímo přístupné z jiných tříd.  
  Nelze řídit úroveň přístupu, můžeme odkudkoliv měnit stav objektu, proto se pro členské proměnné příliš nepoužívá.

Modifikátory přístupu se uvádějí u členských proměnných, vlastností i metod

### Klíčové slovo static

**OOP umožňuje definovat členské proměnné a metody na samotné třídě**. Těmto prvkům říkáme **statické** (někdy třídní) a jsou **nezávislé na instanci**.

Pokud je uvedeno klíčové slovo **static**, musím daný člen volat přímo na úrovni třídy, nikoliv z instance typu.  Statická data se alokují pouze jednou a sdílejí všechny instance objektů téhož typu. Statické metody mohou operovat pouze se statickými daty.

Když je třída definována jako statická, nedají se vytvářet její instance klíčovým slovem new, a může obsahovat pouze statické členy.

**Příklad 1**: Vytvořte program, který bude mít 3 TextBoxy. Do kterých bude zadávat jméno, pozdrav a počet stěn kostky. Program Vás **pozdraví**, oznámí vám, jakou **kostkou házíte** a jaké **číslo** padlo. Např: *„Ahoj Pavel – kostka s 6 stěnami – hodil jsi číslo 5“.* Pokud nevyplníte počet stěn bude nastaven automaticky na 6. Vytvořte **třídu Zdravice** s **veřejnou členskou proměnnou** text pozdravu a **metodou** vracející text pozdravu a jméno. Dále vytvořte **třídu Kostka**, která bude obsahovat dva přetížené **konstruktory**, které budou nastavovat počet stěn na kostce. Jeden konstruktor bude mít nastaven počet stěn na 6 a druhý podle vstupu z TextBoxu. Dále bude obsahovat soukromou členskou proměnnou random a počet stěn a veřejnou **metodu** VratpocetSten, která vrací počet stěn a veřejnou **metodu** Hod, která vrátí náhodné celé číslo v intervalu <1, počet stěn>. Dále bude obsahovat přetíženou **metodu** **ToString**, která vrátí větu “*kostka s 6 stěnami*”

Vytvoříme třídu – **class Zdravice.** Název třídy píšeme vždy velbloudí notací bez mezer a na rozdíl od proměnných název začíná velkým písmenem. Název je samozřejmě také bez diakritiky, tu v programu používáme maximálně uvnitř textových řetězců, nikoli v identifikátorech.

Podle této třídy později vytvoříme **objekt zdravice**, který nás bude umět pozdravit.

Nyní si do třídy Zdravice přidáme metodu **Pozdrav(),** bude veřejně viditelná a bude mít návratovou hodnotu string a parametr string.

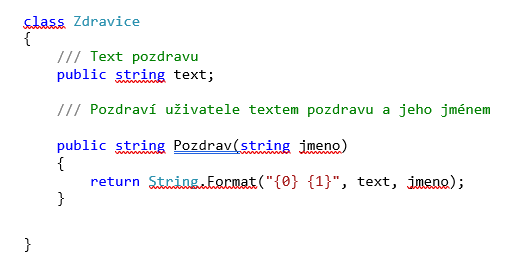
Deklarace metody v C# je tedy následující:

[modifikátor přístupu] [návratový typ] [JmenoMetody]([parametry])

Před metodou bude tzv. modifikátor přístupu, v našem případě **public** (veřejná). Kdybychom modifikátor vynechali, C# by ji chápal jako private (soukromá). Metoda bude vracet řetězec znaků - string. Dále bude následovat samotný název metody, metody píšeme stejně jako třídy velbloudí notací s velkým počátečním písmenem. Závorka s parametry je povinná, i když metoda neobsahuje žádný parametr. Naše metoda bude obsahovat parametr string jmeno.

Dále do třídy přidáme veřejnou členskou proměnnou pro text pozdravu.

Naše třída bude nyní vypadat takto:



Komentáře budeme psát nad název třídy a nad název každé členské proměnné a metody. K jejich zápisu použijeme trojlomítko "///", VS nám poté vygeneruje XML, do kterého vše popíše. Vyplatí se nám to ve chvíli, když na třídě používáme nějakou metodu, její popis se nám zobrazí v našeptávači.

Nyní vytvoříme obsluhu tlačítka.

Vytvoříme **instanci** třídy Zdravice. Bude to tedy ten objekt zdravice, se kterým budeme pracovat. Objekty se ukládají do proměnných, název třídy slouží jako datový typ. Instance má zpravidla název třídy, jen má první písmeno malé. Deklarujme si proměnnou a následně v ní založme novou instanci třídy Zdravice:

Zdravice zdravice = **new** Zdravice();

Při vytvoření nové instance se zavolá tzv. **konstruktor**. To je speciální metoda na třídě, proto při vytvoření instance píšeme ty prázdné závorky, jelikož voláme tuto "vytvářecí" metodu. Konstruktor zpravidla obsahuje nějakou inicializaci vnitřního stavu instance (např. dosadí výchozí hodnoty do proměnných). My jsme v kódu žádný konstruktor nedeklarovali, C# si proto vytvořil tzv. implicitní bezparametrický konstruktor. Vytvoření instance objektu je tedy podobné volání metody.

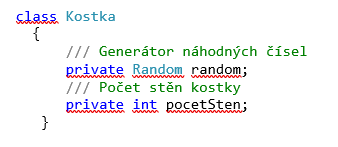
Jelikož v proměnné nyní máme opravdu instanci třídy Zdravice, můžeme instanci nechat pozdravit. Zavoláme na ni metodu *Pozdrav* a to jako zdravice.Pozdrav(„Pavel“).

Zdravice zdravice = new Zdravice();

zdravice.text = textBox2.Text;

MessageBox.Show(zdravice.Pozdrav(textBox1.Text));

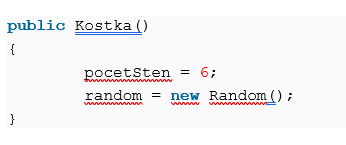
K projektu si přidejme novou class s názvem Kostka. Do třídy přidejte soukromou členskou proměnnou pocetSten. Dále bude kostka potřebovat tzv. generátor náhodných čísel. Naše třída bude mít nyní 2 členské proměnné: *pocetSten* typu int a *random* typu Random, kde bude náhodný generátor. Při návrhu třídy tedy nastavíme vše na private a v případě, že něco bude opravdu potřeba vystavit, použijeme public.



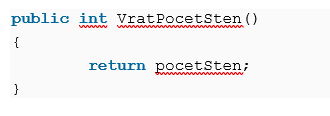
Až doposud jsme neuměli zvenčí nastavit jiné členské proměnné než public, protože např. private nejsou zvenčí viditelné. Již jsme si říkali něco málo o konstruktoru objektu. Je to metoda, která se **zavolá ve chvíli vytvoření instance objektu**. Slouží samozřejmě k nastavení vnitřního stavu objektu a k provedení případné inicializace. Kostku bychom nyní vytvořili takto:

Kostka kostka = **new** Kostka();

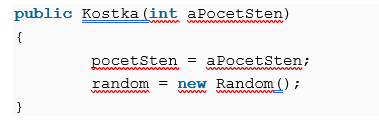
Právě Kostka() je konstruktor. Protože v naší třídě žádný není, C# si dogeneruje prázdnou metodu. My si však nyní konstruktor do třídy přidáme. Deklaruje se jako metoda, ale **nemá návratový typ** a musí mít **stejné jméno jako je jméno třídy**, v našem případě tedy Kostka. V konstruktoru nastavíme počet stěn na pevnou hodnotu a vytvoříme instanci třídy Random. Konstruktor bude vypadat následovně:

****

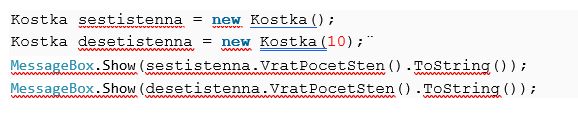
Pokud kostku nyní vytvoříme, bude mít v členské proměnné *pocetSten* 6 a v *random* bude vytvořená instance generátoru náhodných čísel. Není dobré členský proměnné nastavit na public, protože nebudeme chtít, aby nám někdo mohl již u vytvořené kostky měnit počet stěn. Přidáme do třídy tedy metodu VratPocetSten(), která nám vrátí hodnotu členské proměnné *pocetSten*. Docílili jsme tím v podstatě toho, že je členská proměnná read-only (členská proměnná není viditelný a lze ho pouze číst metodou, změnit ji nelze). Nová metoda bude vypadat asi takto:



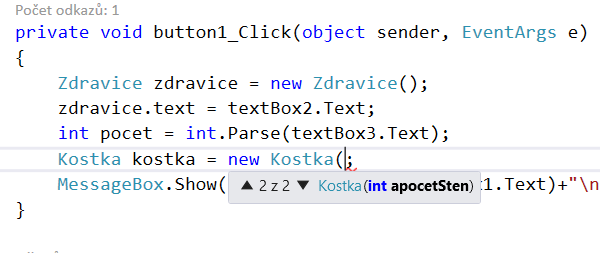
My bychom ale chtěli, abychom mohli u každé kostky při vytvoření specifikovat, kolik stěn budeme potřebovat, přidáme další konstruktor s parametrem.

****

Zkusme si nyní vytvořit 2 instance kostky, každou jiným konstruktorem:



C# nevadí, že máme 2 metody se stejným názvem, protože jejich parametry jsou různé. Hovoříme o tom, že metoda Kostka (tedy zde konstruktor) má **přetížení (overload)**. Toho můžeme využívat i u všech dalších metod, nejen u konstruktorů. Visual Studio nám přetížení u metod přehledně nabízí (ve chvíli, kdy za název metody napíšeme levou závorku), variantami metody si můžeme listovat pomocí šipek. Tohoto pomocníka nazval MS IntelliSense. V nabídce vidíme naše 2 konstruktory:



Mnoho metod v .NETu má hned několik přetížení, zkuste se podívat např. na metodu Remove() na stringu. Je dobré si u metod projít jejich přetížení, abyste neprogramovali něco, co již někdo udělal před vámi.

Ukážeme si ještě, jak jde obejít nepraktický název členské proměnné u parametrického konstruktoru (v našem případě aPocetSten). Problém je samozřejmě v tom, že když napíšeme:

**public** Kostka(**int** pocetSten)

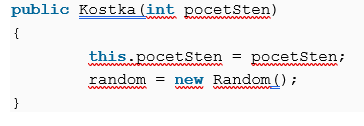
{

pocetSten = pocetSten;

random = **new** Random();

}

C# neví, kterou z proměnných myslíme, jestli parametr nebo členskou proměnnou. V tomto případě přiřazujeme do parametru znovu ten samý parametr. VS nás na tuto skutečnost dokonce upozorní. Uvnitř třídy se máme možnost odkazovat na její instanci, je uložena v proměnné **this**.



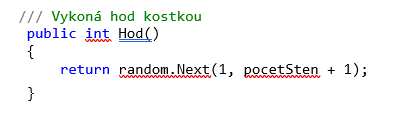
Pomocí this jsme specifikovali, že levá proměnná *pocetSten* náleží instanci, pravou C# chápe jako z parametru. Máme tedy 2 konstruktory, které nám umožňují tvořit různé hrací kostky.

Náhodná čísla

Definujme na kostce metodu *Hod()*, která nám vrátí náhodné číslo od 1 do počtu stěn. Je to velmi jednoduché, metoda bude public (půjde volat zvenčí) a nebude mít žádný parametr. Návratová hodnota bude typu int. Náhodné číslo získáme tak, že na generátoru zavoláme metodu Next(). Ta má několik přetížení:

* Next(): Varianta bez parametru vrací náhodné číslo v celém rozsahu datového typu int;
* Next(Do): Vrací nezáporná čísla menší než mez Do. random.Next(100) tedy vrátí číslo od 0 do 99.
* Next(Od, Do): Vrátí náhodné číslo v zadané mezi, přičemž Od do intervalu patří a Do již ne. Tedy náhodné číslo od 1 do 100 by bylo random.Next(1, 101);

Pro naše účely se nejlépe hodí třetí přetížení, píšeme tedy:



Dejte si pozor, abyste netvořili generátor náhodných čísel v metodě, která má náhodné číslo vracet, tedy že by se pro každé náhodné číslo vytvořil nový generátor. Výsledná čísla pak nejsou téměř náhodná nebo dokonce vůbec. Vždy si vytvořte jednu sdílenou instanci generátoru (např. do privátního členské proměnné pomocí konstruktoru) a na té potom metodu Next() volejte.

**Příklad 2**: Vytvořte třídu **Zbozi**, která bude obsahovat členské prvky název produktu, cenu produktu, datum výroby, počet dní minimální trvanlivosti. Vše nastavte v konstruktoru.

Vytvořte metodu **MinTrvanlivostDny**, která vrací kolik dní zbývá do konce minimální trvanlivosti.

Vytvořte metodu **NovaCena**, která nastaví cenu zboží. Pokud zbývají do konce minimální trvanlivosti **3 dny a méně,** zlevněte zboží o **25%.**  
Pokud je minimální trvanlivost **překročena**, zlevněte o **50%**.  
Pokud je minimální trvanlivost **překročena** o více než **10 dní** nastavte cenu na 0

Vytvořte metodu **Vypis**, která vypíše název produktu, cenu produktu, datum výroby, počet dní minimální trvanlivosti, kolik dní zbývá do konce minimální trvanlivosti. Pokud cena zboží je 0, místo ceny napište zboží je **neprodejné.**

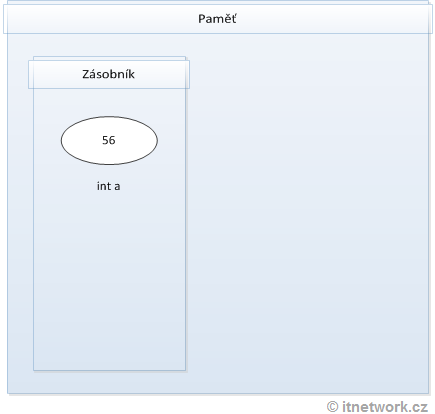
Překrývání metody ToString()

Kostka je téměř hotová, ukažme si ještě jednu užitečnou metodu, kterou ji přidáme a kterou budeme hojně používat i ve většině našich dalších objektů. Řeč je o metodě ToString(), o které jsme se již zmínili a kterou obsahuje **každý** objekt, tedy i nyní naše kostka. Metoda je určena k tomu, aby vrátila tzv. **textovou reprezentaci instance**. Hodí se ve všech případech, kdy si instanci potřebujeme vypsat nebo s ní pracovat jako s textem. Tuto metodu mají např. i čísla. Již víme, že v C# funguje implicitní konverze, jakmile tedy budeme chtít vypsat číslo nebo kterýkoli jiný objekt, C# na něm zavolá metodu ToString() a vypíše její výstup. Pokud si děláme vlastní třídu, měli bychom zvážit, zda se nám takováto metoda nehodí. My si ji ke kostce přidáme, bude vypisovat, že se jedná o kostku a vrátí i počet stěn. Metodu ToString() nemůžeme jen tak definovat, protože je již definována. Musíme ji tedy přepsat, resp. **překrýt**. K překrytí použijeme klíčové slovo **override**:

Referenční a hodnotové datové typy

Popíšeme si odlišnost mezi referenčními datovými typy (objekty) a hodnotovými (např. int). Zopakujme si pro jistotu ještě jednou, co jsou to hodnotové typy. Obecně jsou to **jednoduché struktury**, např. jedno číslo, jeden znak. Většinou se chce, abychom s nimi pracovali co **nejrychleji**, v programu se jich vyskytuje **velmi mnoho** a zabírají **málo místa**. V anglické literatuře jsou často popisovány slovy light-weight. Mají **pevnou velikost**. Příkladem jsou např. int, float, double, char, bool a další.

Aplikace (resp. její vlákno) má operačním systémem přidělenou paměť v podobě tzv. zásobníku (stack). Jedná se o velmi rychlou paměť s přímým přístupem, její velikost aplikace nemůže ovlivnit, prostředky jsou přidělovány operačním systémem. Tato malá a rychlá paměť je využívána k ukládání lokálních proměnných hodnotového typu. Proměnnou si v ní můžeme představit asi takto:



Na obrázku je znázorněna paměť, kterou může naše aplikace využívat. V aplikaci jsme si vytvořili proměnnou *a* typu int. Její hodnota je 56 a uložila se nám přímo do zásobníku. Kód by mohl vypadat takto:

**int** a = 56;

Můžeme to chápat tak, že proměnná *a* má přidělenu část paměti v zásobníku (velikosti datového typu int, tedy 32 bitů), ve které je uložena hodnota 56.

Vytvoříme si jednoduchou třídu, která bude reprezentovat uživatele nějakého systému.

**class** Uzivatel

{

**public** **int** vek;

**public** **string** jmeno;

**public** Uzivatel(**string** jmeno, **int** vek)

{

**this**.jmeno = jmeno;

**this**.vek = vek;

}

**public** **override** **string** ToString()

{

**return** jmeno;

}

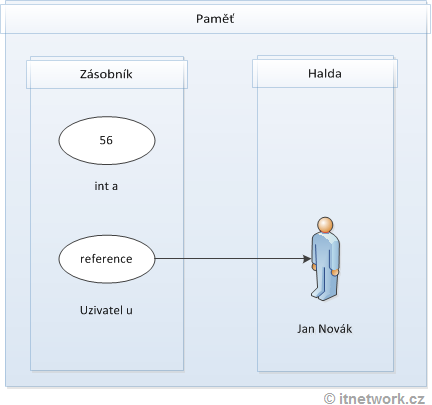
}

Třída má 2 jednoduché veřejné členské proměnné, konstruktor a přetížený ToString(), abychom uživatele mohli jednoduše vypisovat.

**int** a = 56;

Uzivatel u = **new** Uzivatel("Jan Novák", 28);

Proměnná *u* je nyní referenčního typu. Podívejme se na novou situaci v paměti:



Vidíme, že objekt (proměnná referenčního datového typu) se již neukládá do zásobníku, ale do paměti zvané **halda**. Je to z toho důvodu, že objekt je zpravidla **složitější** než hodnotový datový typ (většinou obsahuje hned několik dalších členských proměnných) a také zabírá více místa v paměti.

Zásobník i halda se nacházejí v paměti RAM. Rozdíl je v přístupu a velikosti. Halda je prakticky neomezená paměť, ke které je však přístup složitější a tím pádem pomalejší. Naopak zásobník je paměť rychlá, ale velikostně omezená.

**Proměnné referenčního typu jsou v paměti uloženy vlastně nadvakrát**, jednou v zásobníku a jednou v haldě. V zásobníku je uložena pouze tzv. **reference**, tedy odkaz do haldy, kde se poté nalézá opravdový objekt.

Můžete se ptát, proč je to takto udělané. Důvodů je hned několik, pojďme si některé vyjmenovat:

1. Místo ve stacku je omezené.
2. Když budeme chtít použít objekt vícekrát (např. ho předat jako parametr do několika metod), nemusíme ho v programu předávat jako kopii. Předáme pouze malý hodnotový typ s referencí na objekt místo toho, abychom obecně paměťově náročný objekt kopírovali.
3. Pomocí referencí můžeme jednoduše vytvářet struktury s dynamickou velikostí, např. struktury podobné poli, do kterých můžeme za běhu vkládat nové prvky. Ty jsou na sebe navzájem odkazovány referencemi, jako řetěz objektů.

Založme si 2 proměnné typu int a 2 proměnné typu Uzivatel:

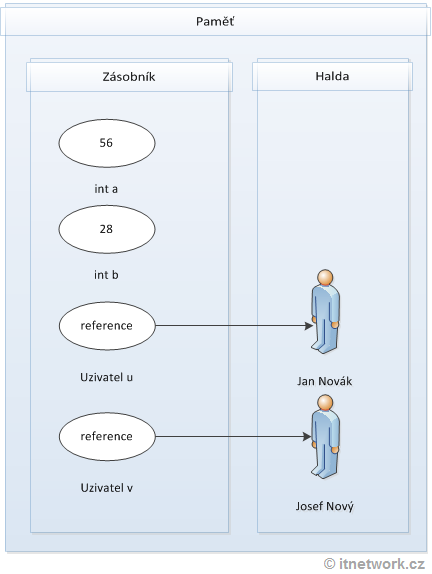
**int** a = 56;

**int** b = 28;

Uzivatel u = **new** Uzivatel("Jan Novák", 28);

Uzivatel v = **new** Uzivatel("Josef Nový", 32);

Situace v paměti bude následující:



Nyní zkusme přiřadit do proměnné *a* proměnnou *b*. Stejně tak přiřadíme i proměnnou *v* do proměnné *u*. Hodnotový typ se v zásobníku jen zkopíruje, u objektu se zkopíruje pouze reference (což je vlastně také hodnotový typ), ale objekt máme stále jen jeden. V kódu vykonáme tedy toto:

**int** a = 56;

**int** b = 28;

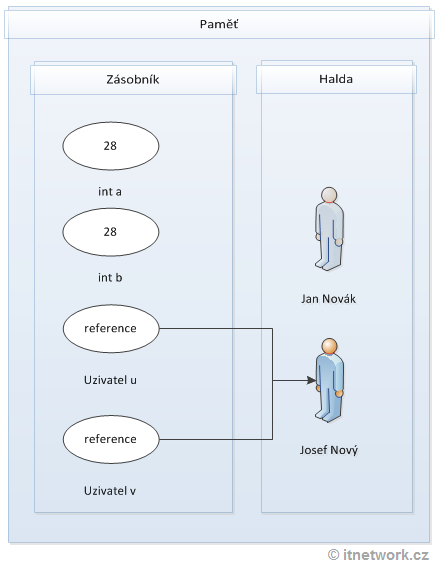
Uzivatel u = **new** Uzivatel("Jan Novák", 28);

Uzivatel v = **new** Uzivatel("Josef Nový", 32);

a = b;

u = v;

V paměti bude celá situace vypadat následovně:



Přesvědčme se o tom, abyste viděli, že to opravdu tak je Nejprve si necháme všechny čtyři proměnné vypsat před a po změně.

// založení proměnných

**int** a = 56;

**int** b = 28;

Uzivatel u = **new** Uzivatel("Jan Novák", 28);

Uzivatel v = **new** Uzivatel("Josef Nový", 32);

MessageBox.Show("číslo a = " + a.toString());

MessageBox.Show("číslo b = " + b.toString());

MessageBox.Show("jméno uživatele u = " + u.toString());

MessageBox.Show("jméno uživatele v = " + v.toString());

// přiřazování

a = b;

u = v;

MessageBox.Show("číslo a = " + a.toString());

MessageBox.Show("číslo b = " + b.toString());

MessageBox.Show("jméno uživatele u = " + u.toString());

MessageBox.Show("jméno uživatele v = " + v.toString());

Na výstupu programu zatím rozdíl mezi hodnotovým a referenčním typem nepoznáme:

číslo a = 56

číslo b = 28

jméno uživatele u = Jan Novák

jméno uživatele v = Josef Nový

číslo a = 28

číslo b = 28

jméno uživatele u = Josef Nový

jméno uživatele v = Josef Nový

Nicméně víme, že zatímco v *a* a *b* jsou opravdu 2 různá čísla se stejnou hodnotou, v *u* a *v* je ten samý objekt. Pojďme změnit jméno uživatele *v* a dle našich předpokladů by se měla změna projevit i v proměnné *u*. K programu připíšeme:

// změna

v.jmeno = "John Doe";

MessageBox.Show("jméno uživatele u = " + u.toString());

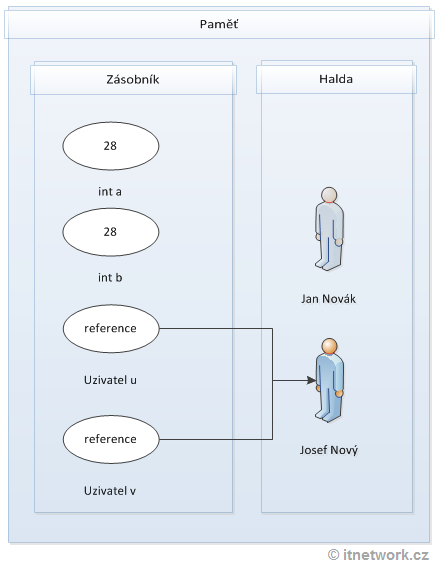
MessageBox.Show("jméno uživatele v = " + v.toString());

Změnili jsme objekt v proměnné *v* a znovu vypíšeme *u* a *v*:

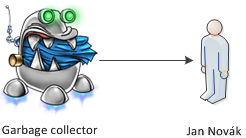
jméno uživatele u = John Doe

jméno uživatele v = John Doe

Spolu se změnou *v* se změní i *u*, protože proměnné ukazují na ten samý objekt. Jestli se ptáte, jak vytvořit opravdovou kopii objektu, tak nejjednodušší je objekt znovu vytvořit pomocí konstruktoru a dát do něj stejná data. Nyní se zaměřme se na Jana Nováka.



Co se s ním stane? "Sežere" ho tzv. Garbage collector.



Garbage collector a dynamická správa paměti

Paměť můžeme v programech alokovat staticky, to znamená, že ve zdrojovém kódu předem určíme, kolik jí budeme používat a jakého jsou datového typu. Ale máme i datové typy jaké je například pole, kdy před spuštěním přesně nevíme, kolik paměti budeme potřebovat. Velikost paměti alokujeme až za běhu programu. V tomto případě hovoříme o **dynamické správě paměti**.

Garbage collector je program, který běží paralelně s naší aplikací, v samostatném vlákně. Občas se spustí a podívá se, na které objekty již v paměti nevedou žádné reference. Ty potom odstraní. Poslední věc, o které se zmíníme, je tzv. hodnota **null**. Referenční typy mohou, na rozdíl od hodnotových, nabývat speciální hodnoty a to null. Null je klíčové slovo a označuje, že reference neukazuje na žádná data. Když nastavíme proměnnou *v* na null, zrušíme pouze tu jednu referenci. Pokud na náš objekt existuje ještě nějaká reference, bude i nadále existovat. Pokud ne, bude uvolněn GC. Změňme ještě poslední řádky našeho programu na:

// změna

v.jmeno = "John Doe";

v = **null**;

Výstup:

jméno uživatele u = John Doe

jméno uživatele v =

Vidíme, že objekt stále existuje a ukazuje na něj proměnná *u*, v proměnné *v* již není reference.

**Příklad 3**: Založ třídu **Retez** s členskou proměnnou string **veta**. Vytvoř **konstruktor** s jedním parametrem. Vytvoř metodu **PocetSlov**, která spočítá počet slov v řetězci veta, kde každé slovo je odděleno právě jednou mezerou. Vytvoř metodu **Vypis**, která vypíše větu. Vytvoř metodu **Smaz**, která z řetězce **veta** smaže zadaný znak jako parametr.

Založ instanci na třídě **Retez** a zavolej metodu **Vypis a PocetSlov,** Zavolej metodu **Smaz** a znovu **Vypis** . Návod video

**Příklad 4**: Vytvořte program, který bude mít 4 TextBoxy. Do kterých bude zadávat jméno a datum narození dvou lidí. Vytvořte třídu **Clovek**, soukromý členský prvek **jméno**, **narozen**, **konstruktor** bude nastavovat jméno a datum narození. Vytvořte metodu **NastavJmeno** a **VypisJmeno**. Vytvořte metodu **Vek**, která bude vracet věk člověka, metodu **Plnolety**, která bude vracet bool a metodu **Starsi**, kde parametrem bude druhý člověk a metoda bude vracet string, kdo je starší. Dále bude obsahovat přetíženou **metodu** **ToString**, která vrátí větu např: „Petr má 10 let a není plnoletý“. Založte dvě instance třídy a vypište. Změňte jméno druhého člověka a vypište jeho jméno. Zjistěte, kdo je starší.

### Vlastnosti

Velmi často se nám stává, že chceme mít kontrolu nad změnami nějakého členského prvku objektu zvenčí. Budeme chtít členský prvek nastavit jako read-only nebo reagovat na jeho změny. Vytvořme si následující třídu Student, která bude reprezentovat studenta v nějakém informačním systému.

**Příklad pro třídu Student**

**class** Student

{

**public** **string** jmeno;

**public** **bool** muz;

**public** **int** vek;

**public** **bool** plnolety;

**public** Student(**string** jmeno, **bool** pohlavi, **int** vek)

{

**this**.jmeno = jmeno;

**this**.muz = pohlavi;

**this**.vek = vek;

plnolety = **true**;

**if** (vek < 18)

plnolety = **false**;

}

**public** **override** **string** ToString()

{

**string** jsemPlnolety = "jsem";

**if** (!plnolety)

jsemPlnolety = "nejsem";

**string** pohlavi = "muž";

**if** (!muz)

pohlavi = "žena";

**return** String.Format("Jsem {0}, {1}. Je mi {2} let a {3} plnoletý.", jmeno, pohlavi, vek, jsemPlnolety);

}

}

Třída je velmi jednoduchá, student se nějak jmenuje, je nějakého pohlaví a má určitý věk. Podle tohoto věku se nastavuje členská proměnná plnolety pro pohodlnější vyhodnocování plnoletosti na různých místech systému. K uložení pohlaví používáme hodnotu bool, zda je student muž. Konstruktor dle věku určí, zda je student plnoletý. Metoda ToString() je nám vypíše všechny informace. Pomocí konstruktoru si nějakého studenta vytvořme:

Student s = **new** Student("Pavel Hora", **true**, 20);

label1.Text = s.ToString();

Výstup: Jsem Pavel Hora, muž. Je mi 20 let a jsem plnoletý.

Vše vypadá hezky, ale členské proměnné jsou přístupné jak ke čtení, tak k zápisu. Objekt tedy můžeme rozbít například takto (hovoříme o nekonzistentním vnitřním stavu):

Student s = **new** Student("Pavel Hora", **true**, 20);

s.vek = 15;

s.muz = **false**;

label1.Text = s.ToString();

Výstup: Jsem Pavel Hora, žena. Je mi 15 let a jsem plnoletý.

Určitě musíme ošetřit, aby se plnoletost obnovila při změně věku. Když se zamyslíme nad ostatními členskými proměnnými, není nejmenší důvod, abychom je taktéž umožňovali modifikovat. Student si za normálních okolností asi jen stěží změní pohlaví nebo jméno. Bylo by však zároveň vhodné je vystavit ke čtení, nemůžeme je tedy pouze nastavit jako private. Před tím jsme používali metody, které sloužily ke čtení privátních členských proměnných. Jejich název jsme volili jako VypisJmeno() a podobně. Ke čtení vybraných členských proměnných vytvoříme také metody a členské proměnné označíme jako privátní, aby se nedaly modifikovat zvenčí. Třída by nově vypadala např. takto (je vynechán konstruktor a ToString()):

**class** Student

{

**private** **string** jmeno;

**private** **bool** muz;

**private** **int** vek;

**private** **bool** plnolety;

...

**public** **string** VratJmeno()

{

**return** jmeno;

}

**public** **bool** VratPlnoletost()

{

**return** plnolety;

}

**public** **int** VratVek()

{

**return** vek;

}

**public** **bool** Muz()

{

**return** muz;

}

**public** **void** NastavVek(**int** hodnota)

{

vek = hodnota;

// přehodnocení plnoletosti

plnolety = **true**;

**if** (vek < 18)

plnolety = **false**;

}

}

Metody, co hodnoty jen vracejí, jsou velmi jednoduché. Nastavení věku má již nějakou vnitřní logiku, při jeho změně musíme totiž přehodnotit členskou proměnnou plnolety. Zajistili jsme, že se do proměnných nedá zapisovat jinak, než my chceme. Máme tedy pod kontrolou všechny změny členské proměnné a dokážeme na ně reagovat. Nemůže se stát, že by nám někdo vnitřní stav nekontrolovaně měnil a rozbil.

Metodám k navrácení hodnoty se říká **gettery** a metodám pro zápis **settery**. Pro editaci ostatních členských proměnných bychom udělali jednu metodu EditujStudenta, která by byla podobná konstruktoru. Jméno, věk a podobně by se tedy měnily pomocí této metody, tam bychom mohli např. kontrolovat, zda hodnoty dávají smysl, opět bychom odchytili všechny pokusy o změnu na jediném místě.

Když použijeme gettery a settery nehovoříme o členských proměnných, ale o **vlastnostech**. Syntaxe vlastnosti je velmi podobná členské proměnné:

**public** **string** Jmeno { **get**; **set;** }

Zprvu to vypadá, jako bychom deklarovali členskou proměnnou. Jméno vlastnosti je však velkým písmenem, jedná se totiž o metodu (přesněji 2 metody). Ve složených závorkách poté specifikujeme, které metody si přejeme vygenerovat. Za vlastností nepíšeme středník! V ukázce výše se vygeneruje setter i getter, vlastnost půjde tedy normálně číst i modifikovat:

label1.Text = objekt.Jmeno; // číst

objekt.Jmeno = "Jan Malý"; // zapisovat

Jediný rozdíl oproti členské proměnné je zvenčí v tom, že počáteční písmeno je velké. C# ve skutečnosti vygeneruje privátní členskou proměnnou a k němu dvě metody, které podle kontextu volá (pozná dle situace, zda čteme nebo zapisujeme).

Pokud si přejeme, aby se v getteru nebo setteru dělo více, než jen načtení/zápis hodnoty, můžeme si ho definovat ručně. Ukažme si to na našem příkladu s plnoletostí, která se musí po změně věku přehodnotit:

**private** **int** vek;

**public** **int** Vek

{

**get**

{

**return** vek;

}

**set**

{

vek = **value**;

// kontrola plnoletosti

Plnolety = **true**;

**if** (vek < 18)

Plnolety = **false**;

}

}

Zprvu je nutné si vytvořit privátní členskou proměnnou vek s malým písmenem, ve kterém bude hodnota ve skutečnosti uložena. **V getteru a setteru poté pracujeme s touto členskou proměnnou, pokud použijete v get{} nebo set{} Vek, program se zacyklí!**. Není možné definovat jen getter nebo setter, buď se oba vygenerují samy, nebo oba definujeme ručně. Pro přístup k zadané hodnotě je nám v setteru k dispozici klíčové slovo **value**. Takto se v C# do verze 3.0 musely definovat všechny vlastnosti, až poté Microsoft zavedl tzv. autoimplementaci a zkrácený zápis, jaký jsme si uvedli výše. U drtivé většiny vlastností totiž v metodách nepotřebujeme žádnou logiku. S Vek nyní pracujeme opět stejně, jako s členskou proměnnou, jen s velkým písmenem. Nenápadné přiřazení do věku vnitřně spustí další logiku k přehodnocení vlastnosti Plnolety:

objekt.Vek = 15; // nyní se změní i plnoletost

Stejně můžeme pochopitelně implementovat i getter a například něco někam logovat.

Upravíme si naši třídu Student tak, aby používala vlastnosti. Vypadala by takto:

**class** Student

{

**public** **string** Jmeno { **get**;}

**public** **bool** Muz { **get**; }

**public** **bool** Plnolety { **get**; }

**private** **int** vek;

**public** **int** Vek

{

**get**

{

**return** vek;

}

**set**

{

vek = **value**;

// kontrola plnoletosti

Plnolety = **true**;

**if** (vek < 18)

Plnolety = **false**;

}

}

**public** Student(**string** jmeno, **bool** pohlavi, **int** vek)

{

Jmeno = jmeno;

Muz = pohlavi;

Vek = vek;

}

**public** **override** **string** ToString()

{

**string** jsemPlnolety = "jsem";

**if** (!Plnolety)

jsemPlnolety = "nejsem";

**string** pohlavi = "muž";

**if** (!Muz)

pohlavi = "žena";

**return** String.Format("Jsem {0}, {1}. Je mi {2} let a {3} plnoletý.", Jmeno, pohlavi, Vek, jsemPlnolety);

}

}

To je o hodně lepší, že? Vlastnosti budeme odteď používat stále, umožňují nám totiž objekty dokonale zapouzdřit. V .NET jsou **všechny veřejné členské proměnné tříd vlastnosti** (např. nám známá vlastnost Length na stringu), platí pravidlo, že **co jde ven je vlastnost**, co se používá jen **uvnitř je privátní členské proměnné**. Veřejná členská proměnná se příliš nepoužívá.

Student s = **new** Student("Pavel Hora", **true**, 20);

s.Vek = 15;

// s.Muz = false; // tento řádek nyní způsobí chybu a musí být odebrán

label1.Text = s.ToString();

Výstup: Jsem Pavel Hora, muž. Je mi 15 let a nejsem plnoletý.

### Shrnutí vlastnosti:

Vynecháním jedné z metod dostaneme **read only** respektive **write only** vlastnost. Předávaná hodnota do metody set je v jejím těle reprezentována klíčovým slovem **value**.

Vlastnosti poskytují programátorovi při práci s objekty větší komfort než členské proměnné.

Např.:

– vlastnost **Jmeno**, která je pro čtení i zápis:

public string Jmeno { get; set; }

– vlastnost **Hmotnost**, která přístupná pouze pro čtení, uvedeme pouze selektor get:

public double Hmotnost { get;}

– vlastnost **Vyska** pro čtení i zápis, která neumožní zapsat nesmyslnou hodnotu a poskytuje hodnotu ve tvaru celého čísla (přestože uchovává i desetinná místa).

Pokud chceme takto kontrolovat zápis, případně čtení – musíme **implementovat** **selektor get**, případně **modifikátor set**. K tomu potřebujeme soukromou pomocnou členskou proměnnou (v tomto případě vyska):

private double vyska;

public double Vyska

{

get

{

return Math.Round(vyska);

}

set

{

if (value <= 0) MessageBox.Show("Výška musí být kladné číslo");

else vyska = value;

}

}

Pokud ve Visual Studiu ve třídě deklarujeme členskou proměnnou a klikneme na ní pravým tlačítkem, můžeme vygenerovat vlastnost a tudíž ji nemusíme psát ručně. (1. položka menu – Rychlé akce a refaktoring)

private int jmeno;

public int Jmeno { get => jmeno; set => neco = value; }

**PŘÍKLADY:**

1. Založ třídu **Retez** s členskou proměnnou string **veta (**veta je retezec slovo, slova jsou oddělena právě jednou mezerou**)** a členskou proměnnou int **cislo**  
   Nastav vlastnost Veta, která upraví větu tak, že bude začínat velkým písmenem a končit právě jednou tečkou.

Vytvoř **konstruktor** s jedním parametrem, který nastaví vlastnost **Veta** a členskou proměnnou **cislo** na 1.  
Vytvoř **konstruktor** se dvěma parametry, který nastaví vlastnost **Veta** a členskou proměnnou **cislo**.  
  
Vytvoř metodu **ToString**, která vypíše větu.   
  
Vytvoř metodu **Zkrat**, která zkrátí řetězec Veta na daný počet slov, podle členské proměnné **cislo**.

**Příklad** v textBoxu je ahoj jak se máš, co děláš cislo = 4

Po aplikaci metody Zkrat a Vlastnosti Veta

Ahoj jak se máš.

Založ instanci na třídě **Retez** s jedním parametrem a zavolej metodu **ToString** a metodu **Zkrat**. Nastav novou větu a zavolej **ToString**

Založ instanci na třídě **Retez** se dvěma parametry a zavolej metodu **Zkrat** a **ToString**.

Návod video

1. Vytvořte třídu **Student**, která bude obsahovat:

* Veřejná vlastnost **Jméno** pro čtení
* Veřejná vlastnost **Příjmení** pro čtení i zápis
* Vlastnost **Datum narození** pro čtení i zápis. Vlastnost naprogramujte tak, aby se při uložení vždy kontrolovalo, zda se náhodou nejedná o budoucí datum, pokud ano, vypíše se varovné hlášení „Budoucí datum“.
* Vlastnost **Známka** pro čtení i zápis.  
  Naprogramujte vlastnost tak, aby se do vlastnosti uložila hodnota 0, pokud uživatel zadá jinou hodnotu než 1, 2, 3, 4, 5. Dále zajistěte, aby se při čtení této vlastnosti vypsalo varovné hlášení „Neproběhla klasifikace“, pokud obsahuje hodnotu 0.

Třída bude mít jediný **konstruktor**, který nastaví jméno, příjmení a datum narození

Třída bude obsahovat metody:

* **Vek** – spočítá věk studenta
* **Sleva** – zjistí, zda má daný student nárok na slevu v dopravních prostředcích, pokud je mu méně než 26 let.
* Přepsanou metodu **ToString**, která vrátí text se všemi informacemi o Studentovi.

Vytvořte **instanci** třídy Student a vyzkoušejte v programu. Vytvořte List alespoň šesti studentů. Vypište studenty, kteří jsou plnoletí. Vypište průměr známek klasifikovaných studentů

**7.** Vytvořte třídu **NakladniAuto**, která bude obsahovat:

* Soukromou členskou proměnnou **spz**
* Soukromou členskou proměnnou **nosnost** (v tunách),
* Vlastnost **Hmotnost nákladu** (v tunách) – pouze pro čtení inicializovaný na 0

Třída bude obsahovat jediný **konstruktor**

* Nastaví **spz** a **nosnost** na požadované hodnoty

Třída bude obsahovat metody:

* **NalozNaklad** – která naloží požadovaný náklad a nedovolí přesáhnout nosnost. Pokud při naložení celého nákladu je překročena nosnost, metoda umožní naložit náklad částečně, do výše nosnosti a navíc vypíše zprávu, kolik tun nákladu nebylo naloženo.
* **VylozNaklad** – vyloží požadovanou hmotnost nákladu a nedovolí vyložit více, než je naloženo. Pokud je požadováno více, metoda vyloží veškerý náklad a vypíše, kolik tun požadovaného písku chybí.
* Přepsanou metodu **ToString** – vrátí text ve tvaru „Nákladní auto ABC123 má nosnost 3 t a má naloženo 1,5 t “.

Vytvoř instanci třídy a nakládej a vykládej náklad. A zobraz vlastnost Hmotnost nákladu

**8.** Vytvořte třídu **Atlet**, která bude obsahovat:

* Vlastnost **Jméno** bude pro čtení i zápis a bude upravovat jméno tak, aby vždy začínalo velkým písmenem.
* Soukromou členskou proměnnou **únava** (int), bude inicializován na hodnotu 0.

Třída bude obsahovat jediný **konstruktor**

* Nastaví požadované **Jméno**

Třída bude obsahovat metody:

* **Behej** – atlet uběhne zadanou vzdálenost, což se projeví na únavě – únava se zvýší o 10 jednotek na každý uběhnutý km. Únava nikdy nesmí přesáhnout 200 jednotek – při překročení únavy člověk odmítne běhat a sdělí, že je příliš unavený.
* **Spi** – spaním se snižuje únava o 100 jednotek za každou hodinu spánku. Únava nesmí klesnout pod hodnotu 0, dalším spánkem se únava již nesnižuje.
* **ZjistiUnavu**

Vytvořte **instanci** atleta, vyzkoušejte vlastnost Jméno a nechte jej střídavě běhat a spát, vypisujte únavu.

**9.** Vytvořte třídu **Auto**, která bude obsahovat:

* Soukromou členskou proměnnou **znacka** (Škoda Scala, Renault Megane, Ford Mondeo,…)
* Soukromou členskou proměnnou **spotřeba na 100 km**
* Soukromou členskou proměnnou **ujetoCelkem** – celková ujetá vzdálenost během všech jednotlivých jízd (km)
* Soukromou členskou proměnnou **okamzikRozjezdu** – čas rozjezdu
* Soukromou členskou proměnnou **dobaJizdyCelkem** – celková doba všech jednotlivých jízd
* Vlastnost **Jede** pouze pro čtení – uchovává informaci, zda auto jede či ne

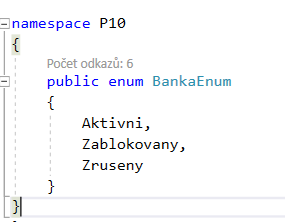
Třída bude mít jediný **konstruktor**, bude nastavovat **značku a spotřebu na 100 km**, celkovou ujetou **vzdálenost** a celkovou **dobu** jízdy nastaví na nulu a vlastnost **Jede** na hodnotu false.

Třída bude mít následující metody:

* **VratUjeteKm** – která vrací celkové kilometry během všech jízd
* **Rozjeď** – rozjede auto a uloží čas rozjezdu.
* **Zastav** – auto **zastaví,** zvýší celkovou ujetou **vzdálenost** o kilometry zadané jako parametr metody a zaktualizuje členskou proměnnou **dobaJizdyCelkem**
* **Celková spotřeba –** vypočte spotřebované palivo za celou dobu jízdy, metoda nebere zřetel na to, zda auto jede čí stojí.
* Přepsanou metodu **ToString** – vrátí **víceřádkový** text se všemi údaji, včetně stavu auta (informace zda jede, či stojí)

Vytvořte instanci třídy Auto a vyzkoušejte v programu.

**10.** Vytvořte třídu **Ucet**, který bude obsahovat jedinečné číslo účtu, které se bude pro nově vzniklý učet automaticky o jedničku zvětšovat (proměnná static). Dále bude obsahovat jméno majitele pro čtení i zápis. Bude obsahovat částku na účtu A status pro daný účet, což bude datového typu enum, nastavte v samostatném souboru na Aktivni, Zablokovany, Zruseny



Konstruktor nastaví jméno majitele, což bude ve vstupním parametru a nastaví číslo účtu.

**Metody**:

public void NastavStatus(BankaEnum status)  
public void Vklad(decimal castka)  
public void Vyber(decimal castka)

Metody Vloz a Vyber vloží nebo vybere z účtu peníze. Metody vypíši aktuální stav na účtu, číslo účtu a jméno majitele. Dále budou kontrolovat dostatek peněz na účtu. S účtem lze manipulovat jen pokud je aktivní. Nelze vybírat ani vkládat záporné částky peněz.

**11.** Založ třídu **Retez,** která má **vlastnost Retezec**, která jen při výpisu retezec upraví tak, že každý znak je oddělen právě jednou mezerou.

Bude obsahovat Metody:

Metoda **Samohlásky**, zjistí počet samohlásek v zadaném řetězci.

Metoda **Palidrom**, která vratí true, pokud řetězec je palidrom, čte se stejně zleva doprava jako zprava doleva (např slovo kajak, madam, Anna..)

Přepsanou metodu **ToString**, která vypíše Retezec, Samohlasky, Palidrom

Vytvořte instanci třídy a zavolejte metodu ToString