Řešené příklady v C#
aneb C# skutečně prakticky

Josef Pirkl

Obsah

1 Řešené příklady v C# aneb C# skutečně prakticky

Dostává se Vám do rukou publikace, při jejíž tyorbě byl brán zřetel na maximální jednoduchost a praktičnost při každodenním použití.

Prvotním impulsem, který vedl k napsání této knihy, bylo poskytnout jakýsi *"lexikon",* který programátor vezme do ruky, když mu bude něco nejasné, když nebude zrovna vědět, jak to či ono implementovat, když bude pouze tušit, že snad takto nějak by to přece mohlo být, že snad takto nějak to již někdy dělal, jakpak to jen ale bylo...?

Podařilo-li se mi tento úkol splnit, musí čtenář posoudit sám. Doufám, že mu bude tato publikace sloužit k jeho plné spokojenosti.

1.1 Rozsah knihy

Je zřejmé, že žádná kniha tohoto „pragmatického rozsahu" není schopná obsáhnout základy jazyka C#, popsat *.NET* model i s jeho metodami, vývojové prostředí *Visual Studio* či samotného C# a k tomu ještě přinést již zmíněný lexikon příkladů a postupů.

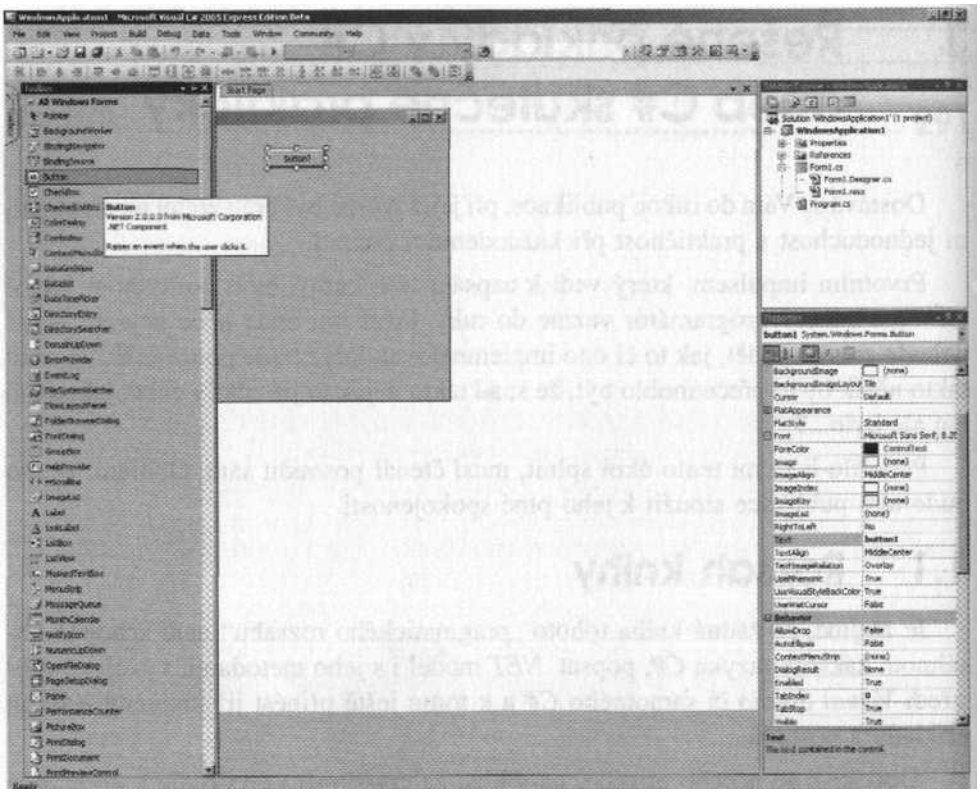
Dotknu-li se něčeho nového, například zabrousím-li ve výkladu k *překrytým* **(override)** metodám, pokusím se na dvou třech řádcích pojem vysvětlit. Zmiňuji se o tom proto, že někomu by se mohl zdát výklad některých termínů příliš stručný - je to však proto, že z hlediska účelu této knihy jsou tyto výrazy uvedeny spíše navíc a předpokládá se, že čtenář má již určitou zkušenost s programováním i se znalostí, prací a použitím objektového programovacího jazyka.

1.2 Verze .NET s příklady

Většina příkladů byla testována v prostředí *Microsoft Visual Studio (C#)* s knihovnou *.NET Framework 1.1*. Lze předpokládat, že velká část z nich bude funkční i v dalších verzích *.NET* knihovny.

Ukázky, které mají za svým titulkem poznámku *„(NET2.0)"* byly testovány v beta verzi *Microsoft C# Express 2005 Beta,* která již nese novější verzi knihovny. *.NET 2.0. Microsoft C# Express* si lze pak volně stáhnout na adrese:

http://lab.msdn.microsoft.com/express/vcsharp/default.aspx Příklady *.NET 2.0* nebudou na starší verzi *NET 1.1* pracovat.



1.3 Typografické konvence

Rezervovaná slova jazyka *C#* a názvy objektů, struktur, rozhraní, výčtů apod. náležející přímo knihovně *.NET* budou uvedeny **tučným písmem.**

Odkazy na názvy vlastních proměnných, čísel, řetězců a názvů technologií budou zvýrazněny *kurzívou.*

Ukázkové kódy pak budou odlišeny tímto podbarvením.

1.3.1 Překlady termínů

Kniha se snaží užívat české termíny, doprovázené alespoň v prvním případě anglickým překladem. Překlad všech důležitějších termínů lze nalézt zde:

bezpečnost	- security
datový proud (tok)	- stream
fronta	- queue
hašovací kód	- hash code
jmenný prostor	- namespace
komplet	- assembly
ladění	- debugging
výčet	- enumeration
omezení	- constraint
poklepání myší	- doubleclick
poskytovatel	- provider
průhlednost	- opacity
přetypování	- typecasting
sezení	- session
schránka	- clipboard
událost	- ebeny
uživatelský prvek	-usercontrol
vázání dat (vazby)	- databinding
vlastnost	- property
(neúplné třídy v C#)	- partial types
(šablony typů v C#)	- generics
zásobník	- stack
zdroj	- resource

1.4 Některá pravidla pro použité zdrojové kódy

Hned na začátku bude definováno, jakým způsobem se bude se zdrojovým kódem pracovat. To se týká zejména použití jmenných prostorů a určení výstupních „nástrojů" **pro** vrácené výsledky z metod.

1 4.1 Pravidla pro užití jmenných prostorů (namespace)

Pro ilustraci předpokládejme tuto deklaraci, která připojuje objekt **XmlDocu-ment,** jenž představuje dokument ve formátu *XML:*

```
XmlDocument doc = new  XmlDocument();
```

V případě, že nebude v hlavičce připojena deklarace **using** pomocí:

```
using System.Xml;
```

oznámí kompilátor při kontrole chybu.

Druhou možností, jak jmenný prostor určit, a to již bez použití příkazu **using,** je jmenným prostorem přímo specifikovat objekt. Pak by deklarace vypadala takto:

Tento způsob sice ušetří jedno použití **using,** nicméně znamená nutnost pokaždé prefixovat objekt příslušným jmenným prostorem, což se vztahuje i na každý jiný prvek, např. výčet, který se ve jmenném prostoru nachází.

Proto tato publikace používá první způsob s tím, že bude pokaždé zdůrazněno, je-li nutné připojit některý další jmenný prostor. Výjimku tvoří příklady pro *.NET 2.0* - zde bude použita celá specifikace jmenného prostoru, nebude-li řečeno jinak.

1.4.2 Výstupy pro příklady

Pro ilustraci výstupů mohou být v kódech použity tyto elementy:

l) MessageBox.Show() - zobrazuje údaj do dialogového boxu

2) listBoxl - prvek **ListBox** pro výpisy rozsáhlejších informací J

3) textBoxl - případně i prvek **TextBox**

Dále je nutné poznamenat, že některé příklady se mohou zdát na první pohled nesmyslné a že by bylo lepší řešit je zcela jinak. Příkladem může být např. kód uvedený u příkazu **continue:**

```
for ( int i = C; i < 3; i++ )
{
  if { i == 1 ) continue; MessageBox.Show( "Krok č. " + i.ToString());
}
```

U tohoto kódu se vždy zobrazí dialog pouze jednou - potom se smyčka opustí. Pak by bylo samozřejmě jednodušší vypsat dialog přímo, bez použití smyčky. Pro ilustraci příkazu **continue** však tento kód vyhovuje - i když testování v reálném životě vypadá často složitěji. To by však navíc vyžadovalo i popis proměnných, které do testování logické podmínky **continue** vstupují, což by čtenáře zbytečně obtěžovalo - a to není cílem této knihy.

1.4.3 Použití znaků…. "v kódu

Tři znaky teček („..."), které budou použity uvnitř kódu, znamenají „libovolný další jiný kód".

1.4.4 Upozornění pro začínající programátory!

Jazyk C# stejně jako jazyky *C/C++* rozlišuje velikost písmen! Lze proto použít kód:

```
int Int = 10;
```

a jeho výsledkem je deklarace a inicializace proměnné *Int* datového typu **int.**

2 Něco ze základních postupů C#

V této kapitole si ukážeme některé základní pravidla práce s jazykem *C#.*

2.1 Přiřazení hodnot proměnným

Začneme deklaracemi a přiřazením hodnot proměnným.

2.1.1 Deklarace proměnné typu int s inicializací

int představuje velmi často používaný celočíselný typ.

```
int i = 5;
```

Následně je demonstrativně provedeno přiřazení nové hodnoty do této proměnné.

2.1.2 Deklarace a přiřazení proměnné typu float s inicializací

Typ **float** obecně vyhovuje pro požadavky na uložení čísla s desetinnou čárkou. Aby kompilátor věděl, že desetinné číslo je přiřazováno jako **float,** musí mu to být řečeno znakem „f" za přiřazovanou hodnotou. Jinak je číslo chápáno jako **double.**

```
float fValue = 56.75f;
```

2.1.3 Deklarace proměnné typu bool s inicializací

bool je logická proměnná, která může nabývat stavů *Ano/Ne* - **(true/false).**

```
bool bValue = false; bValue = tree;
```

Následně je demonstrativně provedeno přiřazení nové hodnoty do této proměnné.

2.1.4 Přiřazení a změna řetězcové hodnoty

Řetězcová hodnota představuje text. Tento text je uvozen znakem uvozovek („).

```
sValue = "Změna hodnoty";
```

Následně je demonstrativně provedeno přiřazení nové hodnoty do této proměnné.

2.1.5 Přiřazení hexadecimální hodnoty (šestnáctková soustava)

```
int i = '\x0201';
```

Přiřazovaná hexadecimální hodnota představuje desítkově číslo *513.*

2.1.6 Přiřazení prvku z výčtu

```
enum Vyska {
  mala, stredni, vysoka }
private void buttonl.Click(object sender, System.EventArgs e) { Vyska vyska = Vyska.stredni;
  }
```

Bázový typ výčtu (jeho prvků) jsou implicitně **int** hodnoty. Hodnoty prvků jsou číslovány vzestupně, přičemž pomocí

znaménka „=" je možné za prvkem přiřadit explitně konkrétní **int** hodnotu. Pro možnost přiřazení jiného typu nežli **int** lze výčet deklarovat takto:

```
enum Vyska: byte
  {...}
```

2.1.7 Přiřazení prázdné reference

Nadsazeně si lze toto přiřazení představit jako nastavení nuly do číselného typu **(int, double).** Říká, že reference *íb* nemá přiřazenou hodnotu - nikam neukazuje.

2.2 Větvení programu

2.2.1 if

Základní větvící příkaz. Logický výraz musí být v závorkách. Spojování jednotlivých podmínek výrazu se provádí pomocí logických operátorů.

```
if ( ( i > 3 ) && ( i < 10))
  MessageSex.Show( "Splněn" );
else
  MessageSex.Show( "Nesplněno");
```

Testuje, zda je i větší nežli 3 a menší nežli 10. Nejpoužívanější logické operátory pro spojování výrazů:

&& - logické A

|| - logické NEBO

! - negace výrazu (uvádí se před výraz)

22.2 switch

Umožňuje přehledně rozvětvit kód dle více testovaných hodnot jednoho výrazu.

Výraz za **switch** musí být v závorce. Každá alternativa čase musí být ukončena explicitním přenosem řízení mimo příkaz **switch** nebo na některé jiné návěstí c**ase** v témže příkazu **switch.** To může obstarat příkaz **break, goto, continue, return, throw.**

Blok **default** je proveden, když hodnota nebyla mezi čase hodnotami naleze- na. Tento **default** blok je nepovinný.

```
int i = 2:
switch (i)
{
   case1: MessageBox.Show( "Jedna" ); break;  case2 :
MessageBox.Show ( "Dvě" ); break;
 case3 : MessageBox.Show( "Tři" ); break;
 default : MessageBox.Show( "Jiná" ); break;
}
```

2.2.3 goto

Přímý odskok na jiné místo v rámci metody. Místo, kam se tímto příkazem přenese řízení, je označeno *návěstím,* za kterým je dvojtečka.

```
int i = 1;

zvysit:
MessageBox.Show( ( i++ ).ToString() );
```

2.2 Smyčky

2.3.2 Příkaz while

Cyklus s testem podmínky na počátku. Vykonává daný počet opakování po dobu, po kterou je splněna úvodní podmínka. V případě nesplnění této podmínky se smyčka opouští.

```
int i = 1;
bool bWhile = true;
while ( bWhile )
{
  MessageBox.Show( i.ToString());
i++;
bWhile = ( i < 4 );
}
```

Tento kód zobrazí 3x příslušný dialog.

2.3.3 Příkaz do...while

Velmi podobné smyčce **while,** avšak test se provádí až na konci - to znamená, že smyčka se v každém případě (není-li jinak násilně přerušena) provede alespoň jednou.

```
int i = 1;
bool = bWhile;
MessageBox.Show( i.ToString () );
bWhile =( i < 4 );
```

Tento kód zobrazí 3x příslušný dialog.

2.3.4 Příkaz foreach

Jazyky C ani *C++* tento druh smyčky nativně neobsahovaly - zavádí ji až C#. Umožňuje procházet prvky kolekcí nebo polí.

```
string[] aArray = new strstring[]{"Jedna","Dva", "Tři"};
foreach ( string sValue in aArray ) MessageBox.Show( sValue };
```

Vypíše postupně všechny tři prvky pole.

Pozn.: Aby kolekce mohla být ve **foreach** použita, musí implementovat rozhraní **IEnumerable** - to s sebou nese povinnost implementace metod tohoto rozhraní.

2.3.5 continue prušení smyčky -> skok na začátek smyčky

```
for ( int i = 0; i < 3; i++ )
 {
   if (i==1 ) continue;
   MessageBox.Show( "Krok č." + i.ToString() );
 }
```

Vypíše dialog s hodnotou „0" a pak s hodnotou „2". Výpis „1" je vynechán.

2.3.6 break - přerušení smyčky -> úplné ukončení myčky

```
 for ( int i = 0; i < 3; it++ )
{
if ( i == 1 ) break;
MessageBox.Show("Krok č. " + i.ToString() );
}
```

Vypíše pouze dialog s hodnotou „0". Pak je smyčka přerušena a pokračuje se až za ní.

Pozn.: Příkaz break se často používá k ukončení alternativ čase v příkazu switch.

2.3.7 Implementace nekonečné smyčky

```
while ( true )
{
 ....
break;
}
```

Zobrazuje dva asi nejpoužívanější způsoby pro vytvoření nekonečné smyčky. Smyčka se opouští pomocí rezervovaného slova break.

2.4 Volání metod a předávání parametrů

Metody a jejich volání tvoří jednu z nejdůležitějších oblastí, které je nutné zvládnout.

pozn.: Metoda je název pro funkci, která je deklarována v rámci objektu.

2.4.1 Předávání parametrů metodám

Správná deklarace pro funkci s parametry je tato:

```
public TestHash( string item1, string item2 ) Pozor! Nelze použít tvar: public TestHash (string item1, item2 )
```

2.4.2 Vyvolání metody s parametry a vrácení výsledku metody

```
int GetSum( int _iParam1, int _iParam2 )
{ re turn  _iParaml + _iParam2);
    private void buttonl_Click(object sender, System.EventArgs e)
  }
int iResult  = GecSum( 10, 20 );
MessageBox.Show( iResult.ToString() );
                                .....
```

Vypíše hodnotu *„30"*.

2.4.3 Předání pole jako parametru

```
void write_array( string [ ] array )
{
string sText = "";
for ( int i = 0 ; i < array.Length; i++) sText = sText + " " + array[il.ToString();
MessageBox.Show( sText  );
}
private void buttonl_Click(object sender, System.EventArgs e)
 {
  string[] sArray = { "Ahoj", "člověče" };     write_array ( sArray );
}
```

Předání parametrů přes pole a jejich výpis uvnitř metody. Lze tak poměrně jednoduše předat více parametrů, aniž je nutné každý zvlášť deklarovat v hlavičce metody.

2.4.4 Vyvolání metody s parametry a vrácení hodnoty přes parametr (out)

```
void GetSum ( int _iParam1, int _iParam2, out int _iResult )
_iResult = _iParaml + _iParam2;
pTivate void buttonl_Click( object sender, System. Ever.tArgs e)
{
    int iResult;
    GetSum( 10, 20, out iResult );
    MessageBox.Show(iResult.ToString() );
}
```

Vyvolá metodu *GetSum()* s parametry *10* a *20,* provede jejich součet a ten vrátí přes parametr typu **out.**

Pozor! Nese-li metoda parametr **out,** musí být s indikací tohoto parametru i vyvolána - čili, i před příslušným parametrem ve volání metody musí být rezervované slovo **out** použito.

Výsledek volám je uložen do proměnné *iResult* (zde hodnota *30).*

2.4.5 Vyvolání metody s parametry a vrácení hodnoty přes parametr (ref)

```
void GetSum{ int .iPararsl, int _iParam2, ref int _iResult   )
   {
   _iResult = „iParaml + _iParam2;
  private void buttonl_Click(object sender, System.EventArgs e)
{
int iResult = -1;

Get.Sum( 10, 20, ref iResult );
MessageBox.Show ( iResult.ToString());
```

Vyvolá metodu *GetSum()* s parametry *10* a *20,* provede jejich součet a ten vrátí přes parametr typu **ref.**

Pozor! Nese-li metoda parametr **ref,** musí být s indikací tohoto parametru i vyvolána - čili i před příslušným parametrem ve volání metody musí být rezervované slovo **ref** použito.

Rozdíl oproti volání **out** je v tom, že hodnota **ref** parametru musí být navíc inicializována před prvním použitím v **ref** volání metody.

Výsledek volání je uložen do proměnné *iResult* (zde hodnota *30).*

2.4.6 Volání statické metody deklarované v jiném jmenném prostoru

Máme-li nový jmenný prostor s objektem *MyMath,* jenž nese jednu statickou metodu *MyMath.Sum():*

```
namespace MyMathNtemeSpace
public class MyMath
{
static public int Sum( irnt iValue1, int iValue2 )
    retun ( iValuel + iValue2 );
     }
```

pak pro volání metody ze základní jednotky použijeme: using MyMathNameSpace;

3 Přetěžování operátorů

V jazyce C# je možné, podobně jako v *C++,* přetížit určité operátory dané třídy, což vlastně znamená, že si definujeme jejich nový význam - a to opět v rámci té a té třídy, v nichž je přetěžujeme.

Názorně: řekněme například, že máme jednoduchou třídu *MyInt,* která eviduje jednu **int** hodnotu. Pokud použijeme operátor „+" mezi dvěmi instancemi tříd *Mylní,* kompilátor oznámí chybu, protože neví, jak tyto instance sečíst. Abychom si pomohli, můžeme v takovém případě „přetížit" operátor „+" ve třídě *MyInt* a určit, co bude vráceno při použití konstrukce *„MyInt + MyInt".*

Přetěžovat v rámci třídy lze unární operátory (+,-,! atd.), binární operátory (%, &,<<,>> atd.) a srovnávací operátory (= =, >, < atd.).

Podobným způsobem lze také přetížit implicitní a explicitní konverzi (třídy) na daný typ třídy.

3.1.1 Přetížení operátoru „+" ve vlastní nové třídě

```
class MyInt
{
  int value;   public MyInt( int value)
{
  this.value = value;
}
  public static MyInt operator+ (Mylnt left, MyInt right )
  return( new MyInt ( (left.value + right.value) * 2 ) );
}
  public override string ToString()
  return Convert.ToString( value );
}
}
```

Volání:

```
MyInt il = new MyInt ( 10 );
MyInt i2 = new MyInt ( 20 );

MyInt result = il + i2;
MessageBox.Show (result.ToString() );
```

zobrazí text *60* - dvojnásobek součtu *10+20.*

pozn.: Ve třídě je vidět způsob přepsání metody *MyInt.ToString()* pro snadný textový výstup.

3.1.2 Přetížení operátoru pro implicitní přetypování na typ int

Vyjdeme-li z předchozí třídy *MyInt,* pak, aby mohlo být použito:

```
int  i = result
```

musí být operátor **(int)** rovněž ve třídě přetížen v této metodě:

```
public static implicit operátor int( Mylnt obj )
{
  return( obj.value );
}
```

To způsobí, že do proměnné i (typu **int)** bude uložena hodnota *60.*

Pozn.: Implicitní konverzi používáme, když se nová hodnota „vejde" do cílové a nehrozí ztráta dat. Typickým příkladem je převod hodnoty typu **int** do **long -long** má dost prostoru, aby **int** hodnotu pojalo.

3.1.3 Přetížení operátoru pro explicitní přetypování na typ int

Vyjdeme-li opět z předchozí třídy *Mylnt,* pak aby mohlo být použito:

```
int i = (int)result;
```

musí být operátor **(int)** rovněž ve třídě přetížen v této metodě:

```
public static explicit operátor int i (MyInt obj )
{
   retum i(obj, value );
}
```

To způsobí, že do proměnné i (typu **int)** bude uložena hodnota *60.*

Pozn.: Explicitní konverzi používáme, když se nová hodnota „nevejde" do cílové a hrozí ztráta dat (např. velký **long** do menšího **int).** Také se používá pro konverze mezi instancemi tříd. Je-li definováno implicitní i explicitní přetížení operátoru na stejný typ, kompilátor vyvolá chybu.

3.1.4 Přetížení operátorů true a false

Po přetížení operátora **true** a **false** lze instance třídy přímo podrobovat logickému testování.

```
class Mylnt
{
  public int value;
  public MyInt( int value )
  {
    this.value = value;
  }
}

public static bool operátor true ( MyIn obj )
{
   retum obj.value > 0;
}
ublic static bool operátor  false ( Mylnt obj )
{
retum obj.value <= 0;
}
}
```

```
MyInt i = new MyInt ( -100 );
MessageBox.Shcw ( "true" );
MessageBox.Show( "false" );
```

Zde vytvořená instance třídy *Mylnt* vrací **true,** je-li hodnota proměnné *instance třídy MyInt.value* větší než nula. V opačném případě je **false.**

Pozn.: Při přetěžování operátoru **true** je nutné přetížit i **false** - a naopak.

3.2 Indexery (indexers)

Indexer umožňuje odkazovat podobným způsobem jako pole přes index na hodnotu, kterou vrací. Lze tak přistupovat k instanci třídy, která ho implementuje.

3.2.1 Vytvoření jednoduchého indexem v rámci formuláře

```
public string this[ int iValue ];
{
return ( iValue.ToString());
}
privace void buttonl_Click(object sender, System.EventArgs e!)
  { MessageBox.Show( this[50] );
}
```

Tento *indexer* nemá sice žádný praktický účel, ukazuje však, jak indexer (zde v rámci formuláře) vytvořit a volat. V tomto příkladě převádí předanou hodnotu 50 na řetězec, který vrací, a jenž se následně zobrazuje. Obecně se pak *indexer* používá k přístupu ke členským polím či kolekcím třídy.

Pozn.: Na rozdíl od C++ se v *C#* za přetěžování operátoru [ ] nepovažuje. Přesto je tato část *zařazena* do této kapitoly.

4 Práce s instancí objektů

Třída v jazyce C#může mimo jiné poskytovat tzv. statické proměnné a metody, ke kterým je možné přistupovat přímo, aniž budeme cokoli deklarovat či vytvářet.

Chceme-li však pracovat s proměnnou, která je *instancí* datového typu **object,** musíme tuto proměnnou nejprve vytvořit - říkáme, že vytváříme instanci objektu.

Zde je nutné upřesnění. V případě, že je např. prvek **TextBox** vložen na formulář umístěním ve vývojovém prostředí (v případě C# obvykle *Visual Studio),* není potřeba vytvářet instanci prvku programově - integrované prostředí generuje automaticky odpovídající kód, který se volá při startu aplikace.

4.1 Dynamické vytvoření a zrušení instance objektů

4-1 • 1 Vytvoření nové instance objektu Timer

```
System.Timers.Timer  timer = new System.Timers.Timer();
timer.Interval = 500;
```

Instance objektů se vytvářejí přes operátor **new.** Pokud to pro vás bude přijatelnější, instance je vlastně totéž, co proměnná. Tak se pouze nazývají dynamicky vytvářené proměnné.
Pozn.: To může být v C# tak trošku zavádějící, neboť objekt je tu každý typ (i ten, který nese hodnotu - např. **int).**

4.1.2 Zrušení objektu Timer

```
System.Timers.Timer timer = new System.Timer();
timer.Interval = 500;
.....

timer.Dispose;
```

Instance objektů se v C# ruší automaticky. Není proto nutné rušit je manuálně a zatěžovat se navíc pamatováním na to, kdy a kde instanci uvolním. V určitých situacích to však může být výhodnější. Explicitní rušení prostředků se provádí přes přepsanou metodu **Dispose()** z rozhraní **IDisposable.** Důležitou podmínkou však je, aby daná třída toto rozhraní implementovala.

4.1.3 Použití klíčového slova this

U práce s instancemi je potřeba vysvětlit význam klíčového slova **this.** To odkazuje na vytvořenou instanci třídy, ve které je **this** v kódu použito.

Nejlépe to lze pochopit na příkladu. Předpokládejme, že máme jednoduchou třídu pro evidenci jedné číselné **int** hodnoty. Třída obsahuje konstruktor, který **int** předanou hodnotu ukládá do členské proměnné třídy:

```
class MyInt
{
  int value;
  public MyInt( int value )
  this.value = value;
}
```

K vytvoření instance této třídy použiji následující volání:

```
MyInt il = newMyInt( 10 );
```

Abych byl v rámci konstruktem **Mylnt()** schopen rozlišit, zda chci pracovat se členskou proměnnou třídy či se stejně pojmenovaným parametrem konstruktoru, použiji klíčové slovo **this.** V takovém případě pak přistupuji k vytvořené instanci, kterou slovo **this** uvnitř třídy zastupuje - čili k proměnné instance této třídy. Volání **this.value** použité v konstruktoru třídy je vlastně stejné, jako volání **il.value,** volané však mimo třídu, přímo na vytvořené instanci.

4.2 Práce s konstruktory

Konstruktor je volán těsně před vytvořením vlastní instance datového typu. Třída může deklarovat konstruktor, ale nemusí - v takovém případě je použit konstruktor implicitní. Konstruktor má stejné jméno jako třída, může jich být i více, existuje-li požadavek na inicializaci instance s rozdílnými parametry.

4.2.1 Konstruktor s jedním parametrem

```
public class MyClass
{
  public MyClass ( int iValuel )
  {
  .....
  MessageBox.Show( iValuel.ToString() );
  }
  }
MyClass  m1 = new  MyClass( 10 );
```

Při vytvoření každé nové instance dochází k předání celočíselné hodnoty *„10".* Ta je v tomto případě zobrazena.

4.2.2 Statický konstruktor

Statický konstruktor je vyvolán pouze pro první vytvářenou instanci daného datového typu

```
public class myClass
{
    public MyClass (int iValue1)
  {
   .....
    MessageBox.Show(iValue1.ToString())
  }
  Static MyClass()
 {
 ........

MessageBox.Show( "První" );
 }
}
MyClass m1 = new MyClass( 10 ); MyClass m2 = new MyClass( 20 );
```

V tomto případě dochází k zobrazení textu *„První"* a "10" pro první vytvářenou instanci datového typu *MyClass;* a textu *"20"* pro druhou vytvářenou instanci.

4.3 Testy na typy objektů

Instance objektů určitého datového typu lze přetypovat na jiný datový typ. To lze tehdy, existuje-li mezi třídami instance přímá závislost: např. lze přetypovat **TextBox** na základního předka **object,** protože ho již „v sobě obsahuje"; lze přetypovat předanou instanci datového typu **object** na **TextBox,** vím-li, že instance byla vytvořena jako **TextBox.** Nelze však přetypovat **TextBox** na **DataGrid,** protože se jedná o zcela jiný nezávislý objekt.

K tomu, aby se mohlo správně a bez chyb přetypovávat, pomohou následující příklady.

4.3.1 Test na třídu instance objektu (či na předka objektu)

K testování, zda daná instance je dané třídy, se používá operátor **is.**

```
private void listBoxl.DoubleClick(object sender, System.EventArgs e)
{

 bool b = ( sender is ListBox); MessageBox.Show( b.ToString () );

 }
```

Tento příklad vrací **true,** je-li předaná instance v parametru *sender* třídy **List Box** (je-li odvozená z této třídy).

4.3.2 Test na třídu instance objektu a její přetypování na zvolenou třídu

```
private void listBoxl.DoubleClick(object sender, System.EventArgs e)
{
  if ( sender is ListBox)
  ListBox  li = (ListBox)sender;
}
```

Testuje, zda je předaná instance objektu v parametru *sender* typu **ListBox;** pokud ano, provádí přetypování instance na typ **ListBox.**

4.3.3 Výpis typu instance objektu

```
private void button1_Click(object sender, System.EventArgs e)
{
  /* pokud neni Button, tak... */
  if( !( sender is Button ))
  MessageBox.Show( "Prvek není typu Button, ale " + sender.GetType() .ToString());
  return;
}
/* je-li Button, pokračuj zde */
Button button = (Button)sender;
.....
}
```

Testuje, zda parametr *sender* není typu **Button** nebo potomek této třídy. Pokud není, vypíše hlášení, do něhož specifikuje, o jakou třídu objektu se tedy jedná.

4.4 Přetypování

Přetypování umožňuje, aby se s instancí jednoho typu objektu zacházelo jako **s** instancí jiného typu objektu. To znamená, že objekt je deklarován jako jeden typ, pracuje se s ním jako s jiným typem. Toto však nelze provést u každé z kombinací typů objektů. Lehce lze přetypovat potomka na předka, dobře akceptované jsou také přetypování mezi numerickými typy (v jazyce C# je však stejně každý typ reprezentován jako objekt - což je určitou odlišností od jiných konvenčních objektových jazyků).

4.4.1 Explicitní přetypování double na int

```
double d = 35.6; int i = (int)d;
```

Uloží do celočíselné proměnné i **double** hodnotu *d* a to s přetypováním na typ **int.** Uloženo bude *35* - **int** samozřejmě nepracuje s desetinnými místy. Toto přety-pování na **(int)** se nazývá explicitní a mohli jsme si ho dobře všimnout při tvorbě explicitního přetypování u operátorů.

Bez tohoto přetypování by kompilátor ohlásil chybu - jde o konverzi, při níž 1 se ztrácí informace, a proto je vyžadováno, aby ji programátor vyjádřil explicitně. Všechny „zužující" konverze musí být v *C#* explicitně předepsány.

4.4.2 Přetypování parametru sender na třídu Button

```
private void button1_Click(object sender, System.EvemArgs e)
Button b = (Button)sender;
MessageBox.Shcw( b.Name );
```

Parametr *sender* nese odkaz na objekt, který událost vyvolal - v případě tohoto kódu se jedná o událost, která obsluhuje stisk tlačítka. Vím-li tedy, že se jedná např. o tlačítko, mohu přímo přetypovat předaný typ **object** na **Button** a dostat se k jeho vlastnosti **Button.Name,** kterou samotný typ **object** neposkytuje. Tento příklad vypíše název stisknutého tlačítka *„button1".*

4.5 Deklarace vlastností (properties)

Vlastnosti (properties) zastupují přístup k proměnným třídy. Programátor tedy přes vlastnosti k členským proměnným třídy přistupuje; čili čte, popřípadě upravuje jejich hodnotu. Vlastnost je tedy jakýsi „prostředník" mezi programátorem a proměnnou uvnitř třídy.

Po pravdě řečeno, programátor je schopen vyjít i bez použití vlastnosti a to tak, že deklaruje proměnné jako veřejně přístupné a pak přistupuje přímo k nim. Tato technika se však nedoporučuje.

Velkou výhodu použití vlastností vidím osobně v tom, že po změně proměnné mohu jako programátor říci, že má dojít i k následné úpravě další proměnné či skupiny proměnných, že mohu okamžitě překreslit upravené grafické vyjádření instance objektu, jedná-li se o grafický objekt, apod. Pokud budu přistupovat přímo k proměnným instance objektu, musím na vše toto myslet a vystavuji se riziku opomenutí a vzniku chyby. Rovněž i pozdější úprava kódu je pro mě jako pro programátora náročnější, musím sledovat všechny tyto závislosti, které jsem mohl včlenit do obsluhy pomocí vlastností.

```
public int Valuel get { return this.iValuel; }
if ( value > 10 ) value - 10;
this. iValuel = value;
doAct ion ();
private void doAction( ) { /* volitelné následné akce */ }
```

V rámci třídy MyClass je vidět nová vlastnost Valuel, která podporuje čtení hodnoty (get), tak i její úpravu (set). Vlastní hodnota je potom uložena v privátní(=z venku třídy nevidelné) proměnné iValuel.

V bloku set je před změnou testováno, zda nová hodnota, reprentovaná pomocí value, není větší než 10. Pokud ano, sníží se nová hodnota na 10. Volání doAction() pak ilustruje i možnost zpracování následných akcí, které se provedou bezprostředně po změně proměnné.

```
MyClass myclass = new MyClass();
myclass.Valuel = 20;
MessageBox.Show(myclass.Value1.ToString());
```

Tento příklad zobrazí hodnotu *10* (v set části dochází ke snížení dle zadané Podmínky).

5 Práce s událostmi

Událost (event) představuje místo, které je vyvoláno pro obsloužení určité akce - typickým příkladem je reakce na stisk tlačítka. Obsluhu události musí mít třída implementovánu (lze samozřejmě vytvořit událost vlastní).

Pro události se v jazyce C# vytváří objekt typu **delegate,** který událost zastupuje. To mimo jiné umožňuje přiřadit pomocí operátoru += na danou událost více reakcí. Operátor -= ji může zase odebrat.

5.1.1 Dynamické přidělení události

Událost lze přidělit přímo v návrhu formuláře v editoru vývojového prostřed (pro C# obecně Visual Studio) - pak je událost s prvkem svázána staticky (což uz(můžeme programově změnit); alternativou je dynamické přidělení události přímo za běhu programu - např. až po startu formuláře, po stisku tlačítka, apod.

Přidělí-li se na obsluhu události např. dvě reakce, jsou také dvě (po sobě) vyvolány

```
private void resizeProc( object sender, System.EventArgs e )
MessageBox.Show("Resize");
}
this.Resize += new EventHandler (resizeProc)j;
```

Pokud se provede:

```
this.Resize -= new EventHandlen (resizeProc);
```

a událost není zatím přiřazena, nic se neděje.

5.1.2 Přidělení více reakcí na jednu událost

Dynamické přiřazení dvou metod pro obsluhu stisku tlačítka buttonl:

```
private void Clickl ( object sender, EventArgs e )
MessageBox.Show ("1" );
private void Click2 (object sender, EventArgs e)
{
  MessageBox.Show( "2" );
}
buttonl.Click += new EventHandler( Clickl );
buttonl.Click += new EventHandler( Click2 );
```

Po stisku tlačítka buttonl se zobrazí dialog s „1" a pak s „2".

5.1.3 Odstranění jedné reakce na událost (jsou-li již přiděleny)

Dynamické přiřazení dvou metod pro obsluhu stisku tlačítka *buttonl s* následnou ukázkou odebrání metody *Click1:*

```
private void Clickl( object sender, EventArgs e }
KessageBox.Show( "1" );
private void Click2 (object sender, EventArgs e)
MessageBox.Show ("2" );
burttonl.Click += new EventHandler ( Clickl ); button1.Click += new EventHandler ( Click2 );
buttonl.Click. - = Clickl;
```

Metoda je z obsluhy události odstraněna pomocí operátoru -=. Zobrazí se pouze dialog s „2".

5.1.4 Ovlivnění výsledku přes událost

Často se události používají i ve smyslu notifikačním - v prototypu metody události je tento parametr předán referencí (odkazem).

Obsluha vlastní metody události pak tuto proměnnou buď ponechá nebo upraví. Podle toho, co je v tomto parametru vráceno, se pak chová program, který metodu události vyvolal.

```
Public delegate void CalcHandler( double d1, double d2, ref double result ); public class Suma public Suma( double dl, double d2 )
{
  this.d1 = d1;
  this.d2 = d2;
}
  Public double doCalc()
{
```

6.1. 1 Obsluha obecné (nekonkrétní) výjimky

```
int i = 10;
int i = 0;
try
{
        i= i / i1;
}
        catch
{
        MessageBox.Show( "Nastala vzjíimka  chyba");
}
```

Předpokládám-li, že výjimka může nastat a neznám přesně její typ (=třídu), výjimka pro mě nemá konkrétní podobu, nevím co očekávám, jen tuším místo, kde by mohly nastat problémy, definuji blok try...catch. Ten libovolnou výjimku zachytí a umožní obsloužit.

6.1.2 Obsluha konkrétní výjimky

```
int i = 10;
int i = 0;
try
{
 i = i / i1
}
catch(System.DivideByZeroExceptions)
{
MessagBox..Show( "'Nastalo dělení nulou." );
 catch
 {
MessagBox..Show( "'Nastala vyjímka chyba." );
 }
```

Vim-li předem, který typ výjimky by mohl nastat, obsloužím její typ přímo, úvěrem je možné obsloužit všechny ostatní výjimky. Zobrazí hlášení *„Nastalo dělení nulou."*.

6.1.3 Získání bližších informací o výjimce

```
int i = 10;
  int i = 0;
double result= ( d1 + d2 );
AfLerCalc( d1, d2, ref result );
return (result);
double dl;
double d2;

public event CalcHandler AfterCalc;
private void AfterCalcProc( double dl, double d2, ref double result )
{
MessageBox.Show( Scring.Format( "Událost proběhla - {0}, {1}", dl, d2 ) );
resuit *= 2; }
private void buttonl_Click(object sender,: System.EventArgs e) { Suma s = new Suma( 10, 50 );
s.AfterCalc += new CalcHandler( AfterCalcProc);
double d = s.doCalc();
MessageBox.Show ( String.Format("Výsledek = {0}", d ) );
```

V tomto příkladě je při *Suma.doCalc()* vyvolána událost *AfterCalcProc.* Ta má parametr *result,* který je typu ref, což umožňuje přes změnu tohoto parametru v metodě události ovlivnit jeho hodnotu zpět do *Suma.doCalc()*.

6 Práce s výjimkami

Výjimka představuje objekt pocházející z předka Exception. Výjimka "obaluje" chybu. Je vytvořena při vzniku chyby (je-li výjimka vyvolána) a jako taková předána do místa obsluhy, kde je zachycena v části **catch**. Objekt výjimky pak popisuje chybu, která vznikla, její typ, případně i další informace důležité pro její obsluhu.
Pro názornost: v případě, že např. programátor provede dělení nulou, je systémem interně vyvolána předdefinovaná systémová výjimka DivideByZeroExcepition která by se měla obsloužit. Obsluhou výjimky v jakékoli formě se zabráni pádu programu a zobrazení implicitního, často konečnému uživateli nesrozumitelného chybového hlášení.

```
try
{
i = i / i1; }
catch ( Exception el )
{
MessageBox.Show( "Nastala výjimka - chyba * + e1.StackTrace );
```

Proměnné *el* budou předány bližší informace o vzniklé výjimce - např. originální chybový text, místo jejího vzniku. Tento příklad řetězcově vypisuje náhled na zásobník v okamžiku vzniku výjimky.

Použití výjimky typu **Exception** je možné, měla by však existovat snaha použít co nejspecializovanější typ, abychom už samotným typem výjimky co nejpřesněji popsali vzniklý problém (např. **ArithmeticException).**

Pozn.: Tato obsluha přes **catch** (typ_vyjimky) je univerzální, zachytí všechny výjimky vzniklé v *.NET* prostředí. Blok **catch** bez specifikace typu výjimky slouží k zachycení výjimek, které vzniknou i mimo prostředí *.NET*- např. v nezabezpečení ném kódu v C++.

6.1.4 Vyvolání výjimky

```
int i = 10; int i1 = 5;
try {  if (i1==S) throw new Exception( "Pětkou nelze dělit." );
catch ( Exception e1 ) {
MessageBox.Show ( "Chyba - " - e1.Message );
}
```

V případě, že dělitel výrazu je roven pěti, vyvolá výjimku s příslušným hlášením.
Pozn.: Příklad je nesmyslný, lze jej vyřešit i podmínkou, způsob vyvolání výjimky však ukazuje názorně.

7 Práce s konzolí - konzolová aplikace

používá se pro jednodušší aplikace s textovým výstupem v „dosovém" okně.

Při vytváření projektu konzolové aplikace se postupuje přes "New Project..." -> „ Console Application".

7.1.1 Výstup textu na konzoli

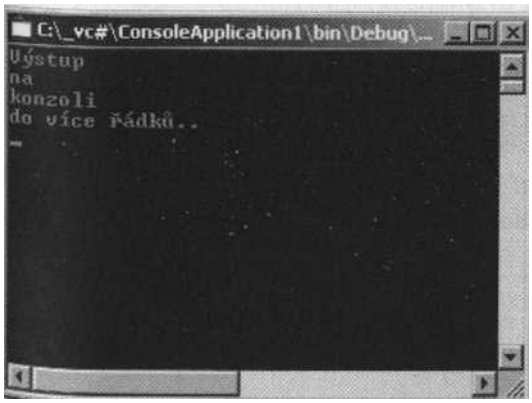
```
static vold Main(string[] args)
Console.WriteLine ( "Výstup na konzoli ...");
Console. Read ();
```

Vypíše na konzoli jeden řádek a čeká na stisk <*Enter*>.

7.1.2 Výstup na konzoli do čtyř řádků přes jeden příkaz

```
{
ConsoleWrite ("Vystup\nna\nkonzoli\ndo vice radku")
ConsoleRead();
}
```

Vypíše na konzoli pod sebe čtyři řádky a čeká na stisk <*Enter*>.



7.1.3 Formátovaný výstup na konzoli

```
double d = 55.555;
Ccnsole.WriteLine( "Hodnota je {0:N2} procenc.", d );
```

Lze používat stejné masky, které jsou přístupné u metody **String.Format().**

Tento příklad vypíše *,Hodnota je 55.56 procent".*

Pozn.: Příklady na hlubší použití formátovacích masek lze nalézt v kapitole „ *Práce s datovými typy "* - „ *Práce s řetězci".*

7.1.4 Určení výstupního proudu (pro XmlTextWriter)

```
System.Xml.XmlTextWriter writer = new System.Xml.XmlTextWriter( Console.Out );
writer. WriteStart.Document ();
writer.WriteStartElement ( "Text");
.......
writer.WriteEndDocument () ;
writer.Closse();
Console.Read();
```

Tímto způsobem lze zajistit, že jednoduchý výstup z instance objektu **XmLTextWriter** bude automaticky směřován na konzoli **(Console.Out).**

7.1.5 Vymazání konzole zvolenou barvou (.NET 2.0)

```
Console.BackgroundColor = ConsoleColor.Gray;
Console.Clear();
Console.ForegroundColor - ConsoleColor.Yellow;
Console.WriteLine( "Vystup na šedé barvě Zlutým písmem" );
```

.NET 1.1 nepodporoval vymazání konzole - za tímto účelem musely být celkem složitě volány funkce jádra *Windows API.* Nyní je tato možnost přímo podporována objektem **Console.**

7.1.6 Ukončení přes <Ctrl+C> - odchycení klávesy na konzoli (.NET 2.0)

```
static  void Main (string[] args)
{
Console.CancelKeyPress += new
ConsoleCancelEventHandler (Console.CancelKeyPress);
Console.WriteLine( "Čekání na ukončení přes CTRL-C );
wile (true) { };
}
static void Console.CancelKeyPress(object sender, CcmsoleCancelEvenLArgs e)
Console.WriteLine( "Stisk klávesy/CTRL+C" );
```

Konzolová aplikace čeká tak dlouho, dokud není vyvolána událost reakce na **Console.CancelKeyPress** - ta totiž představuje stisk kombinace kláves <*Ctrl+C*>.

8 Práce s komponentami z palety nástrojů

Paleta nástrojů (tzv. *toolbox)* se implicitně (v závislosti na zvoleném typu vytvářené aplikace) objevuje na levé straně vývojářského prostředí. Objekty, které nabízí, lze myší aktivovat a umístit přímo na formulář - programátor není tedy nucen k tomu, aby každý pryek vytvářel dynamicky (programově).

Komponenty lze rozdělit na:

a) vizuální - po spuštění aplikace jsou vidět přímo na formuláři **(TextBox, ListBox, TreeView)**

b) nevizuální - mají určitou skrytou činnost, jejich umístění na formuláři je viditelné pouze ve vývojovém stádiu, v běžícím programu však vidět nejsou **(Timer, EventLog)**

8.1 Společné pro všechny vizuální komponenty

Třída Control je společným předkem pro všechny vizuální komponenty. Proto je možné jím poskytované společné rysy využít u všech tčchto zděděných prvků (tříd).

Pro demonstraci bude použit objekt TextBox, pamatujte si však, že stejně tak je možné aplikovat tyto příkazy např. nad objektem ListView. Panel atd.

8.1.1 Změna pozice prvku přes Control.Left a Control.Top

```
textBoxl.Left = 10; textBoxl.Top  = 10;
```

Prvek bude přesunut na pozici [10,10].

8.1.2 Změna pozice prvku pres Control.Location

```
textBoxl.Location = new Point: ( 10, 10 i;
```

Další varianta pro změnu pozice. Prvek bude přesunut na pozici *[10, 10]*.

8.1.3 Změna délky prvku přes Control.Width

```
textBoxl.Width = 100;
```

Délka prvku bude nastavena na *100* pixelů.

8.1.4 Změna délky prvku přes Control.Size

```
textBoxl.Size = new Size( 1000, textBoxl.Height );
```

Alternativa pro změnu rozměrů - zde nastavuje délku na 100, výšku ponechává původní.

8.1.5 Změna barvy písma a změna barvy prvku pozadí prvku

```
textBox1.ForeColor = Color.Red;
textBoxl.BackColor = Color.Yellow;
```

Nastavuje barvu popředí (barvu textu) na červenou a barvu pozadí na žlutou.

8.1.6 Změna barvy rodičovského objektu

```
if ( textBox1.Parent != null ) textBoxl.Parent.BackColor = Color.Red;
```

Zde je vidět přístup k rodičovskému objektu, na kterém je prvek *textBox1* umístěn a změna barvy pozadí tohoto rodiče.

8.1.7 Skrytí objektu na formuláři

Po změně vlastnosti Control.Visible je vyvolána událost Con-trol.VisibleChanged. Tu lze přepsat, např. následujícím způsobem:

```
Private void textBoxlVisibleChanged(object sender, System.EventArgs e)
MessageBox.Show ("'Visible=" + c.Visible.ToString());
```

Jako praktický význam lze uvést např. automatické zobrazení jiného pole po změně viditelnosti, apod.

8.1.8 Zákaz práce s objektem - objekt je viditelný, zákaz je graficky rozlišen

```
textBox1.Enabled = false;
```

Po změně vlastnosti **Control.Enabled** je vyvolána událost **Control.EnabledChanged.** Tu lze přepsat, např. následujícím způsobem:

```
private void textBoxl_EnabIedChanged(object.sender, System.EventArgs e)
  {
Control c = (Control)sender;
MessageBox.Show( "Enabled=" + c.Enabled.ToString() );
}
```

8.1.9 Reakce na vstup do prvku (podporuje-li ji, např. 1 textové pole)

```
private void textBoxl_Enter(object sender, System.EventArgs e)
{
  labell.Text = 'Vstup do prvku';
}
```

Obsluha této události umožňuje při vstupu do prvku kurzorem provést inicializační akce.

Pozn.: Vstup do vizuálního prvku na formuláři (klávesou, myší či programově) označujeme jako získání fokusu či zaostření. Obvykle je prvek i určitým zpúsobem graficky odlišen - např. širším rámečkem nebo zvýrazněným textem. Naopak, po kud se prvek stane neaktivním, ztrácí fokus, zaostření.

8.1.10 Reakce na výstup z prvku (podporuje-li ji, např. textové pole)

```
private void textBoxl_Leave(object sender, System.EventArgs e)
labell.Text = "výstup z prvku";
```

Událost je volána při opuštění prvku.

8.1.11 Reakce na získání zaostření prvku (fokusu)

```
private void textBox1.GotFocus(object sender, EvenLArgs e)
{
  label.Text = "Focus prvku";
}
```

Reakce je volána po zaostření prvku. Tato událost je vyvolána až po události Control.Enter.

8.1.12 Reakce na ztrátu zaostření prvku (fokusu)

```
private void textBox1.LostFocus(object sender, EventArgs e)
labell.Text = "Bez focusu";
```

Reakce je volána po ztrátě zaostření. Tato událost je vyvolána ještě před události Control.Leave.

8.1.13 Získání velikosti prvku jako typ Rectangle

```
Rectangle r = buttonl.ClienlRectangle;
```

Tato metoda je velmi využívána zejména ve vykreslování grafiky.

8.1.14 Control.SetStyle a přepsání notifikační metody Control.OnNotifyMessage()

Je-li nad prvkem formuláře nastaven atribut ControlSty-les.EnableNotifyMessage, je pro každou zprávu, která je obsluhována v proceduře okna (WndProc), volána notifikační metoda OnNotifyMessage. Tuto metodu lze Překrýt a upravit chování pro specifickou zprávu.

```
private void buttonl_Click{object sender, EventArgs e)
{
this.SetStyle( ControlStyles.EnableNotifyMessage, true );
}
protected override void OnNtotifyMessage(Message m);
{
base.OnNotifzMessage (m);
if ( m.Msg == '\x201' )
listBoxl.Items.Add( m.ToString()); }
```

Tento kód nejprve po stisku tlačítka aktivuje notifikační metodu (ta je implicitně neaktivní). Metoda je přepsána a mimo původní význam testuje, zda se vyskytla zpráva *WM_LBUTTONDOWN* (stisk levého tlačítka myši) a pokud ano vypisuje informaci o této zprávě do ListBoxu.

Pozn.: Control.SetStyle() se používá zejména k úpravě volání způsobu vykreslování přes metodu Paint() + použití vykreslovacích bufferů.

8.1.15 Změna kurzoru

```
this.Cursor - Cursors.WaitCursor;   buttonl.Cursor = Cursors.Default;
```

Nastaví kurzor hodin nad celým formulářem (mimo tlačítko *button1,* kde ponechá kurzor implicitní).

8.1.16 Načtení kurzoru ze souboru

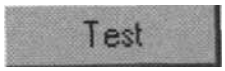
```
Cursor myCursor = new Cursor(@ "c:\lano.cur");
if ( myCursor != null ) this.Cursor = mzCursor;
```

Z externího souboru *c:\lano.cur* načítá kurzor a aktivuje ho.

8.2 Button

Klasické tlačítko.

8.2.1 Změna textu na tlačítku



8 2.2 Změna textu i obrázku na tlačítku s definicí jeho zarovnání

```
button1.Text = "Text";
button1.TextAlign = ContentAlignraent.TopCenter;
button1.Image = Image.FromFile( @"c:\calc.bmp" ); button1.ImageAlign = ContentAlignment.BottomCenter;
```

Určuje se zarovnání textu i zarovnání obrázku (který byl načten ze souboru).

8.2.3 Změna barvy tlačítka po najetí myší a změna na Flat styl (.NET 2.0)

```
buttonl.FlatAppearance.MouseOvnerBackColor; buttonl.FlatStyle = FlatStyle.Flat;
```

Přes Button.FlatAppearance lze nastavit způsob vykreslení tlačítka, zde je jako styl použit Flat (ploché).

8.2.4 Zkrácení dlouhého textu tlačítka a doplnění znaky "..."(.NET 2.0)

```
buttonl. Autoelipsis = true;
button1.Text = 'Toto je delší tex";
```

Delší text, který by jinak přesahoval, je zkrácen a ukončen znaky „...".

8.3 ComboBox

Seznam hodnot, v jeden okamžik může být viditelná jedna hodnota.

8.3. 1 Naplnění hodnot do ComboBoxu s aktivací první hodnoty

```
comboBox1.Items.Clear();
comboBox.Items.Add("Hodnota 1");
```