Obsah

Psaní kódu

Zde jsou uvedeny konvence dle doporučeni **Microsoftu**. Použitý zdroj:
https://msdn.microsoft.com/en-us/library/ff926074.aspx

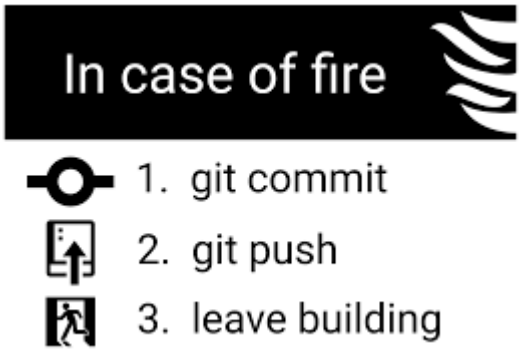
> Tyto konvence se případně uzpůsobí potřebám Fullcom.

Verzování

Pro správu zdrojových kódů se používá nástroj GIT.
Zde jsou nějaké základní pravidla:

1. Projekt je rozdělen do dvou základních větví: *
   - **dev**: čistě vývojářská část, kam se commitují změny dle aktuálního vývoje
   - **master** (případně **prod**): produkční/release verze - toto by měl být komplet jdoucí již do reálné instalace k zákazníkovi
2. Je žádoucí, aby se jednotlivé programátorské úkony rozdělili na menší jednotky vhodné pro commit do GITu - důvodem je lepší práce při ReviewCode a také to umožní vývojáři lépe utřídit myšlenky/práci na menší samostatné celky
3. Každý commit musí být srozumitelně popsán ve smyslu co bylo učiněno, není vhodné dávat příliš obecné popisky jako např. „Malá úprava kódu“ či „Drobné změny“
4. Při ukončení práce (typicky na konci pracovní doby) je nutné **vždy udělat PUSH** na centrální repozitář - důvod je jasný = v případě poruchy vývojářského počítače se nepřijde o rozpracovanou práci
5. **Nikdy nedělat PUSH do některé z hlavních větví (dev, master), pokud projekt není možné buildnout!** Nastane-li situace, že je projekt rozpracovaný, ale nezkompilovatelný, tak **push** udělat do nějaké vlastní dočasné větve

*) *Bod 1) ještě není zcela finální, zde bude třeba promyslet postupy...*

> V případě požáru postupovat následovně :)
>
> 
>

Rozvržení

- používat výchozí nastavení editoru ve Visual Studiu: „smart“ odsazení, čtyři mezery pro odsazení, „tab“ ukládat jako mezery
- na jeden řádek psát pouze jediný příkaz
- na jeden řádek psát pouze jedinou deklaraci
- vždy alespoň jeden prázdný řádek mezi definicemi metod a definicemi vlastností
- v rámci bloku kódu používat **prázdný řádek pouze vyjímečně**, je-li potřeba takových prázdných řádků z důvodu čitelnosti v kódu více, tak se jedná o nevhodně napsaný kód
- dodržovat odsazovaní následujících řádků - automaticky anebo za použití klávesy „tab“
- používat složené závorky tak jak je uvedeno na této ukázce

```
if ((val1 > val2) && (val1 > val3))
{
    // nějaká další část kódu
}
```

Obecně je doporučováno psát krátké metody s maximálně 20 řádky kódu. Obdobně to platí o třídách, které by neměly být příliš rozsáhlé.
Důvodem je jak lepší testovatelnost, tak i jednodušší ladění - nalezení chyb.
Toto pravidlo si budeme muset nastavit např. v nějakém doplňku pro hlídání čistoty kódu.

Rozložení třídy

Každá třída bude dodržovat shodné rozložení metod a vlastností v uvedeném pořadí:

1. statické privatní členy
2. statické privátní vlastnosti
3. statické privátní metody
4. statické veřejné vlastnosti
5. statické veřejné metody
6. privátní členy
7. protected členy - zvážit jestli opravdu je nutné mít člen, doporučuji spíše propertu
8. delegáti
9. konstruktory
10. privátní vlastnosti
11. privátní metody
12. protected vlastnosti
13. protected metody
14. veřejné vlastnosti
15. veřejné metody
16. destruktor

Komentáře

- komentáře psát **vždy česky s diakritikou a pravopisně správně**
- umísťovat komentáře na samostatný řádek, nesmí být na konci řádku za příkazem či deklarací
- začátek komtáře začína vždy velkým písmenem
- konec komentáře je ukončen tečkou
- nevytvářet bloky komentářů pomocí hvězdiček (/** */)
- mezi značku komentáře (//) a samotný komentář vždy vložit jednu mezeru

```
// Toto je poznámka k této části kódu. A zde následuje
// další řádek.
```

Blok IF / ELSE

Pro použití podmínek v kódu jsou tato pravidla:

1. jednořádkový zápis - ten je povolen pouze pokud je podmínka k vyhodnocení relativně jednoduchá (maximálně 3 operátory) a současně bude na základě splnění podmínky vykonán pouze jediný příkaz příkaz:

```
if (value > 150 || (!active && dateCreate > DateTime.Now.AddDays(-10))
callSomeMethod(value);
// Anebo
if (value < 150) throw new ApplicationException("Value is less 150");
```

2. víceřádkový zápis - v případě, že bude na základě podmínky vykonáno více příkazu, anebo se v těle podmínky nacházejí složitější konstrukce (např. ternární operátory), tak je nutné vždy tělo podmínky zapsat odděleně do složených závorek:

```
if (value > 150 || (!active && dateCreate > DateTime.Now.AddDays(-10))
{
  callSomeMethod(value);
  finnishAndSave();
}

if (value > 150 || (!active && dateCreate > DateTime.Now.AddDays(-10))
{
  callSomeMethod(value);
  if (value > 9999) throw new ApplicationException("Value overflow");
}
```

3. pokud je vyhodnocovaná podmínka komplikovanější, tak použít pomocnou proměnnou s vhodným názvem:

```
var allowSaveRecord = data.Active == true && data.EndTime == null;
allowSaveRecord = allowSaveRecord && user.IsAdmin;
if (allowSaveRecord)
{
  callSomeMethod(value);
  finnishAndSave();
```

```
}
```

Smyčky FOR / FOREACH / WHILE

Veškeré smyčky musí být řešeny formou bloku kódu umístěného samostatně ve složených závorkách:

```
for(var i=; i<10l i++)
{
  callSomeMethod(i);
  finnishAndSave();
}

var a = 50;
while(a > )
{
  callSomeMethod(a);
}
```

Pravidla pro datové typy

String

V případě krátkých string proměnných spojovat řetězec za pomocí operátoru „**+**“:

```
string displayName = nameList[n].LastName + ", " + nameList[n].FirstName;
```

Pokud jsou řetězce dlouhé anebo pokud je zpracováváme ve smyčce, tak použít vždy třídu **StringBuilder**:

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = ; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
```

Implicitní deklarace

- Používat implicitní deklaraci proměnné pomocí klíčového slova **var** v případě, že jde o lokální proměnnou u které je zřejmé z pravé strany o jaký typ půjde. Případně když není podstatné o jaký datový typ jde.

```
// Je-li typ proměnné zřejmý z kontextu, tak použít var v deklaraci.
var var1 = "This is clearly a string.";
var var2 = 27;
```

```
var var3 = Convert.ToInt32(Console.ReadLine());
```

- Nepoužívat klíčové slovo **var** v případě, že z kontextu není jasno o jaký datový typ jde:

```
int var4 = ExampleClass.VratNejakyVysledek();
```

- Nepoužívat v názvu proměnné název datového typu, tato informace může být zavádějící:

```
// Pojmenování inputInt je matoucí.
// Vstupem je řetězec.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Používat implicitní typ **var** ve smyčkách:

```
var syllable = "ha";
var laugh = "";
// Implictní var pro for
for (var i = ; i < 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}
// Implictní var pro foreach
foreach (var ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

- Nepoužívat klíčové slovo **var** pro typy **dynamic**

Datové typy "unsigned"

Obecně platí, že použití **int** je vhodnější než **unsigned** typy. Použití **int** je běžné v C# a usnadňuje to interakci s ostatními knihovnami.

Pole - Array

Používat zkrácenou syntaxi při inicializaci pole:

```
// Preferovaný zápis. Všimněte si, že nelze použít var místo string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };

// Pokud používáme explicitní "instancionování", tak je možné použít var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// Pokud se určí velikost pole, musí se inicializovat prvky jeden po druhém.
```

```
var vowels3 = new string[5];
vowels3[] = "a";
vowels3[1] = "e";
// Atd.
```

Delegáti

Použít stručnou syntaxi pro vytvoření instance.

```
// Za prvé, ve třídě, definovat typ delegáta a metodu, která
// má odpovídající podpis.

// Definice typu.
public delegate void Del(string message);

// Definice metody s odpovídajícím podpisem.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// V hlavní metodě vytvořit instanci delegáta Del.

// Preferovaný způsob: vytvoření instance delegáta pomocí zkráceného zápisu.
Del exampleDel2 = DelMethod;

// Toto je vytvoření instance delegáta nezkrácenýcm zápisem.
Del exampleDel1 = new Del(DelMethod);
```

Vyjímky

try/catch

Používat **try/catch** pro odchycení většiny vyjímek v aplikaci.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

try/finaly - using

Pokud chceme ošetřit vyjímku nad objektem implementujícím rozhrani **IDisposable**, tak použijeme doporučený způsob s klíčovým slovem **using**:

```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

...místo **try/finally**:

```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

Operátory "&&" a "||"

Pro zvýšení výkonu vyhnutí se vyjímkám při porovnávání vždy používáme zdvojené operátory **&&** a **||** místo **&** a **|**. Tím se vyhneme zbytečnému porovnávání v případě, že je již vyhodnocení podmínky jasné.

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());

// Pokud je dělitel 0, tak druhá část podmínky by vyvolala vyjímku.
// Operátor "&&" zajistí, že v případě nesplnění první části podmínky se již
druhá nebude provádět.
// Pokud bycho použili pouze operátor "&" tak se vždy vyhodnotí
// obě části podmínky.
if ((divisor != ) && (dividend / divisor > ))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
```

```
}
```

Operátor "New"

Používat vždy zkrácenou formu vytváření instance za pomocí implicitní definice klíčovým slovem **var**.

```
var instance1 = new ExampleClass();
```

Používat „inicializátory“ při konstrukci objektu pro jeho jednodušší vytvoření:

```
// Objekt inicializátor.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Standardní inicializace objektu.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Eventy - zpracování událostí

Pokud se definují události, které není nutné později odstraňovat, tak používat „lambda“ výrazy.

```
// Tradiční použití má nevýhodu "rozdrobenosti" kódu na více různých míst.
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

...a při použití lambda:

```
// Použití lambda výrazu je kratší a čitelnější než tradiční zápis.
// Ihned na místě je vidět co daná metoda dělá.
public Form2()
{
    // Zde se použije lambda výraz pro definici handleru.
    this.Click += (s, e) =>
        {
            MessageBox.Show(((MouseEventArgs)e).Location.ToString());
        };
}
```

Static members

Volejte statické členy pomocí názvu třídy: **ClassName.StaticMember**. Tato praxe činí kód čitelnější tím, že je jasně vidět volání statického členu.
Nelze, ale volat statický člen bázové třídy s názvem odvozené třídy. I když toto zápis poněkud komplikuje, tak kód je čitelnější a není zavádějící do budoucna, kdy například můžete odvozené třídě přidat statického člena se stejným názvem.

Linq

- Používat smysluplné názvy pro dotaz vracející proměnné. Následující příklad používá **seattleCustomers** pro zákazníky, kteří se pocházejí ze Seattlu.

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- Použít aliasy, pro jasné zviditelnění kam daná vlastnost patří. Používat *Pascal zápis*:

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Přejmenovat názvy vlastností, pokud ve výsledku budou jejich názvy nejednoznačné.

```
// Zde existuje jak cust.Name tak i dist.Name.
// Přejmenováním na "CustomerName" jasně dáváme najevo o co jde.
var localDistributors2 =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorID = dist.ID };
```

- Použít implicitní psaní datového typu pomocí klíčového slova **var** v přiřazování z dotazu:

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- Používat podmínky ke zúžení výběru dotazu před ostatními částmi, aby se zajistilo optimalizované zpracování těchto dat:

```
var seattleCustomers2 = from cust in customers
                        where cust.City == "Seattle"
                        orderby cust.Name
                        select cust;
```

- Používat spíše syntaxi **from** než syntaxi **join** pro spojení více položek.

```
// Použití spojení dvou kolekci (master/detail) k průchodu po detail
```

```
položkách.
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

- Vždy zarovnávat/odsazovat klauzule *linq* dotazu, tak jek je uvedeno v těchto příkladech.