Programovací jazyk C#

Bázové třídy

Ing. Marek Běhálek
Katedra informatiky FEI VŠB-TUO
A-1018 / 597 324 251
http://www.cs.vsb.cz/behalek
marek.behalek@vsb.cz

Náplň kapitoly

- V této kapitole budete seznámeni se základní sadou knihoven.
  - Třída Systém.Object
  - Vstup a výstup
    - Práce s konzolou
    - Práce se soubory
    - Práce s řetězci
  - Práce s kolekcemi
  - Reflexe
  - Vlákna

Třída System.Object (1)

- Tato třída je kořenem všech typů .NET včetně hodnotových a odkazových typů.
- Není-li v deklaraci uveden typ, předpokládá se, že daný typ bude dědit z Object.
- Všechny třídy z něj dědí implicitně a mohou využívat jeho metody.
- Metody třídy System.Object
  - `string ToString()` - vrací textové vyjádření objektu;
  - `int GetHashCode()` - vrací hash hodnotu, ta může být použita například pro uložené do struktury Map;

Třída System.Object (2)

  - `bool Equals(Object obj)` – standardně porovnává odkazy dvou objektů, ovšem u některých typu bývá přetížená a porovnává hodnoty. Například u třídy `string`;
  - `bool ReferenceEquals(object objA, object objB)` - statická metoda, rozhoduje, zda se dva odkazy odkazují na stejnou instanci třídy;
- operátor porovnání == Ve většině případů znamená porovnání odkazů. Může být přetížen;
- `void Finalize()` - destruktor, který je volán v okamžiku, kdy je z dynamické paměti mazán referent;
- `Type GetType()` - vrací instanci třídy System.Type;
- `object MemberWiseClone()` - vytváří mělkou kopii objektu.

Vstup a výstup – Práce s konzolou (1)

- Jednoduchý vstup a výstup programu zprostředkovává třída *System.Console*. Tato třída obsahuje tyto veřejně přístupné vlastnosti:
  - `Error (System.IO.TextWriter)`
  - `Out (System.IO.TextWriter)`
  - `In (System.IO.TextReader)`
- Některé veřejné metody jsou:
  - `int Read()` - čte další znak ze standardního vstupu nebo -1 pro konec vstupu.
  - `string ReadLine()` - čte celý řádek ze standardního vstupu.
  - `Write()` - tato přetížená metoda vypíše hodnotu parametru. To může být text, číslo, ...
  - `WriteLine()` - podobně jako Write vypíše hodnotu parametru následovanou koncem řádku.

Vstup a výstup – Práce s konzolou (2)

- Příklad:

```
int i=3;
Console.WriteLine("i= {0}", i);
```

- Existuje celá řada formátovacích znaků. Například:
  - C, c –měna;
  - D, d- desítkový zápis;
  - E, e - exponenciální zápis;
  - F, f - zápis s pevnou desetinnou čárkou;
  - G, g - obecný zápis;
  - N, n – číslo;
  - X, x - hexadecimální zápis;
- Existuje celá řada různých formátovacích znaků, například pro data.

Vstup a výstup – Práce s konzolou (3)

- Formátování je ovlivněno nastavením kulturního prostředí.

```
CultureInfo us = new CultureInfo("en-US");
System.Threading.Thread.CurrentThread.CurrentCulture = us;
Console.WriteLine("{0:C}", 1000);

CultureInfo cz = new CultureInfo("cs-CZ");
System.Threading.Thread.CurrentThread.CurrentCulture = cz;
Console.WriteLine("{0:C}", 1000);
```

  - Výstup programu

$1,000.00
1 000,00 Kč

Vstup a výstup – Práce s konzolou (4)

- Formátování celé řady dalších prvků je ovlivněno nastavením kulturního prostředí.
- Obdobně můžete ovlivnit například i výstup data.
  - `DateTime.Now.ToString("D", us);`
    - D – formátovací znak pro datum;
    - Instance třídy CultureInfo, kromě jiného, obsahuje vlastnost DataTimeFormat.

Vstup a výstup – Práce s konzolou (5)

- Speciální znak @
  - Umožňuje vypsat řetězec, tak jak je. Není použito žádné „formátování“.

```
Console.WriteLine(@"Tento řetězec se
na obrazovce objeví tak, jak se v
programu
zadá. Navíc může obsahovat i zpětná
lomítka, tedy tato: \");
```

  - Explicitně definuje, že následuje řetězec.

```
int @if = -1;
```


Vstup a výstup – Práce se soubory (1)

- Pro práci ze soubory je určen jmenný prostor: `System.IO`.
- Celá řada tříd pro práci se soubory.
  - Pro zápis a čtení ze `streamu`.
    - System.IO.StreamReader
    - System.IO.StreamWriter
    - System.IO.TextReader
    - System.IO.TextWriter
  - Třídy reprezentující čtený soubor či paměť.
    - System.IO.FileStream
    - System.IO.MemoryStream
  - Pro práci se soubory System.IO.File a System.IO.Directory

Vstup a výstup – Využití třídy File

- Třída System.IO.File implementuje celou řadu metod pro práci se souborem.
  - Delete
  - Create
  - Copy
  - …
- Čtení a zápis dat z respektive do souboru
  - ReadAllLines
  - ReadAllBytes
  - …
  - Příklad:

```
string text = File.ReadAllText(
   fileName, Encoding.GetEncoding(1250));
```


Vstup a výstup – Práce s textovým souborem

- Čtení dat ze souboru:

```
FileStream fs = new FileStream(
  "soubor.txt", FileMode.Open);
StreamReader sr = new StreamReader(fs);
while(sr != null)
  Console.WriteLine(sr.ReadLine());
sr.Close();
```

- Zápis dat do souboru:

```
FileStream fs = new FileStream(
  "soubor.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
for(int i=0; i<text.GetLength(0); i++)
  sw.WriteLine(i);
sw.Close();
```

Vstup a výstup – Práce s binárním souborem (1)

- Čtení dat ze souboru:

```
BinaryReader br = new BinaryReader(
  File.Open("soubor.bin", FileMode.Open));
try {
  while(true)
  Console.WriteLine(br.ReadInt32());
} catch (EndOfStreamException) {}
finally { br.Close(); }
```

- Zápis dat do souboru:

```
BinaryWriter bw = new BinaryWriter(
  File.Create("soubor.bin", FileMode.Create));
for(int i=0; i<text.GetLength(0); i++)
  bw.Write(i);
bw.Close();
```


Vstup a výstup – Práce s binárním souborem (2)

- Pokud chcete uložit nějaký složitější objekt do souboru, je nutná jeho serializace.

```
[Serializable]
class Osoba {
  private string jmeno;
  private string prijmeni;
  ...
  public Osoba(sring jmeno, string prijmeni) {... }
}
```

- Pro zápis pak můžete použít:

```
using System.Runtime.Serialization.Formatters.Binary;
Osoba karel = new Osoba("Karel","Novak");
FileStream fs = new FileStream("soubor.srd",FileMode.Create);
BinaryFormatter output = new BinaryFormatter();
output.Serialize(fs, karel);
```

- Pro čtení:

```
FileStream fs2 = new FileStream("soubor.srd", FileMode.Open);
Osoba precetena = (Osoba)output.Deserialize(fs2);
```

Vstup a výstup – Práce s řetězci

- C# nabízí široký rozsah prvků zpracování řetězců. Podporuje
  - měnitelné i neměnitelné řetězce;
  - formátování řetězců;
  - porovnávání řetězců;
  - místní nastavení;
  - kódování řetězců;
  - vyhledávání v řetězcích;
  - náhrady regulárními výrazy;
  - kvantifikátory (??, *?, +?, {n, m}?);
  - pozitivní a negativní dopředné vyhledávání;
  - podmínečné vyhledávání.

Vstup a výstup – Třída System.String (1)

- Typ string představuje neměnitelnou sekvenci znaků a je aliasem třídy System.String. Řetězce mají metody:
  - porovnávání; připojování; vkládání; převádění; kopírování; formátování; indexování; spojování; rozdělování; doplňování; ořezávání; odstraňování; nahrazování;prohledávání.
- System.String je odkazovým typem.
- Řetězce jsou neměnitelné (nelze je po vytvoření změnit).
- Metody měnící řetězec ve skutečnosti vytváří řetězec nový.
  - Měnitelný řetězec lze vytvořit třídou StringBuilder.

Vstup a výstup – Třída System.String (2)

- Interování řetězců
  - Neměnitelnost řetězců umožňuje jejich internování.
    - Internování je proces, kdy se všechny stejné konstantní řetězce uloží na stejné místo.
    - Šetří prostor za běhu programu.
  - Může být zdrojem neočekávaných výsledků při porovnávání:

```
string a = "ahoj";
string b = "ahoj";

//True for string only
Console.WriteLine(a == b);
//True for all objects
Console.WriteLine(a.Equals(b));
//True?!
Console.WriteLine((object)a == (object)b);
```


Vstup a výstup – Třída System.String (3)

- Kompilátor převádí operace přičtení (+), kde je na levé straně řetězec, na metody Concat().
- System.String je hodnotový typ, přesto jej lze porovnat operátorem ==.
  - To proto, že je přetížený, lze porovnávat řetězce hodnotou:

```
string a = "abracadabra";
string b = "abracadabra";

//write true
Console.WriteLine(a==b);
```

Vstup a výstup – Třída System.Text.StringBuilder (1)

- Představuje měnitelný řetězec.
  - Obsahuje pole znaků s předdefinovanou velikostí (standardně 16).
  - Pole dynamicky roste při přidávání znaků buď bez omezení nebo do definovaného maxima.
- Má vlastnosti:
  - Length - aktuální délka řetězce;
  - Capacity - objem rezervované paměti.
- Některé metody třídy System.Text.StringBuilder:
  - Append() - připojí řetezec;
  - Insert() - vloží řetězec;
  - Remove() - odstraní znaky z řetězce;
  - Replace() - nahradí výskyty znaku nebo podřetězce;
  - ToString() - Vrací obsah řetězce přetypovaný na System.String.

Vstup a výstup – Třída System.Text.StringBuilder (2)

```
using System;
using System.Text

class TestStringBuilder
{
  static void Main()
  {
    StringBuilder sb = new StringBuilder(
          "Hello ");
    sb.Append("world");
    sb[10] = '!';
    //output: "Hello world"
    Console.WriteLine(sb);
  }
}
```

Vstup a výstup – Podpora regulárních výrazů (1)

- System.Text.RegularExpressions.Regex je srdcem podpory regulárních výrazů.
  - Používá se jako instance objektů i jako statický typ.
  - Představuje neměnitelnou, kompilovanou instanci nějakého regulárního výrazu.
  - Tu lze pak aplikovat na řetězec.

Vstup a výstup – Podpora regulárních výrazů (1)

- Některé statické metody třídy Regex:
  - Escape() - ignoruje metaznaky regulárního výrazu v řetězci;
  - IsMatch() - metody vracející logickou hodnotu, zda byla v řetězci nalezena shoda s regulárním výrazem;
  - Match() - metody vracející instance shody;
  - Matches() - metody vracející seznam instancí shod jako kolekci;
  - Replace() - metody nahrazující vyhledané shody náhradním řetězcem
  - Split() - Metody vracející pole řetězců určených výrazem;
  - Unescape - ruší převádění všech znaků v escape sekvencích v řetězci.

Vstup a výstup – Podpora regulárních výrazů (3)

- Několik metaznaků, které může regulární výraz obsahovat:
  - ^ - počátek vstupního textu;
  - $ - konec vstupního textu;
  - . – jakýkoliv znak kromě konce řádku;
  - * - libovolný počet výskytů znaku uvedeného před hvězdičkou;
  - + - jeden nebo více výskytů znaku uvedeného před plus;
  - ? – 0 nebo 1 výskyt znaku uvedeného před hvězdičkou;
  - \s – bílý znak;
  - \S – vše kromě bílého znaku;
  - \b – hranice slova;
  - \B – jakákoli pozice, která není hranicí slova;
  - kulaté závorky – ohraničují skupiny znaků (metazaků).

Vstup a výstup – Podpora regulárních výrazů (4)

```
//pattern corresponds to string
//built from characters a or b or r
Match m = Regex.Match("abracadabra",    "(a|b|r)+");
//test if anything was found
if (m.Success)
{
    Console.WriteLine("Match = " + m.ToString());
}

string s = Regex.Replace("abracadabra", "abra",
    "zzzz");
//prints "zzzzcadzzzz"
Console.WriteLine(s);

s = Regex.Replace(" abra ", @"^\s*((.)*?)\s*$", "$2-$1");
//prints "a-abra"
Console.WriteLine(s);
```

Vstup a výstup – Příklady regulárních výrazů (1)

- **Vyhledávání římských číslic**

```
string p1 = "^m*(d?c{0,3}|c[dm])" +
"(1?x{0,3}|x[lc])(v?i{0,3}|i[vx])$";
string t1 = "vii";
Match m1 = Regex.Match(t1, p1);
```

- **Prohození dvou prvních slov**

```
string t2 = "very fast brown fox";
string p2 = @"(\S+)(\s+)(\S+)";
Regex x2 = new Regex(p2);
string s2 = x2.Replace(t2, "$3$2$1", 1);
```

- **Vyhledání vzorů dvojic "klíčové-slovo = hodnota"**

```
string t3 = "myValue = 3";
string p3 = @"(\w+)\s*=\s*(.*)\s*$";
Match m1 = Regex.Match(t3, p3);
```

- **Vyhledávání řádku s přinejmenším 80 znaky**

```
string t4 = "******************** … ******************";
string p4 = ".{80,}";
Match m1 = Regex.Match(t4, p4);
```


Vstup a výstup – Příklady regulárních výrazů (2)

- **Odstranění úvodních a závěrečných prázdných znaků**

```
string t5 = "  text  ";
string p5 = "^\s+(\S+)\s+$";
Regex x5 = new Regex(p5);
string s5 = x5.Replace(t5, "$1", 1);
```

- **Změna '\' s následným 'n' na skutečný řetězec nového řádku**

```
string t6 = @"\ntest\n";
string r6 = Regex.Replace(t10, @"\n", "\n");
```

- **Detekování IP adresy**

```
string t7 = "158.196.154.160";
string p7 = "^" + @"([01]?\d\d|2[0-4]\d|25[0-5]\." +
                  @"([01]?\d\d|2[0-4]\d|25[0-5]\." +
                  @"([01]?\d\d|2[0-4]\d|25[0-5]\." +
                  @"([01]?\d\d|2[0-4]\d|25[0-5]" + "$";
Match m7 = Regex.Match(t7, p7);
```

- **Získání všech čísel z řetězce**

```
string t8 = "test 1     test 2     test 2.3     test 47";
string p8 = @"(\d+\.?\d*|\.\d+)";
MatchCollection mc8 = Regex.Matches(t14, p14);
```

Vstup a výstup – Příklady regulárních výrazů (3)

- **Vyhledání všech slov s velkým počátečním písmenem**

```
string t9 = "Toto JE Test velkých Počátečních písmen";
string p9 = @"(\b[^Wa-z0-9_][^WA-Z0-9_]*\b)";
MatchCollection mc9 = Regex.Matches(t9, p9);
```

- **Vyhledání odkazů v jednoduhém HTML kódu**

```
string t10 = @"
<html>
   <a href=\"http://windows.oreilly.com/news/first.htm\">first
text</a>
   <a href=\"http://windows.oreilly.com/news/next.htm\">second
text</a>
</html>
";

string p10 = @"<A[^>]*?HREF\s*=\s*[""']?" + @"([^'"" >]+?)[ '""]?>";
MatchCollection mc10 = Regex.Matches(t18, p18, "si");
```


Kolekce

- Kolekce jsou standardní datové struktury doplňující pole (jediná vestavěná datová struktura v jazyce C#).
- Jazyk obsahuje sadu typů poskytujících datové struktury a podporu vytváření vlastních typů.
- Dělí se do dvou kategorií:
  - rozhraní definující standardizovanou sadu vzorů návrhu pro kolekce obecně;
  - konkrétní třídy implementující tato rozhraní.
- Existuje mnoho různých druhů kolekcí.
- Interní implementace se značně liší, procházení kolekcemi je téměř univerzální.
- Tato funkčnost je zajištěna dvěma rozhraními:
  - IEnumerable
  - IEnumerator

Kolekce - Rozhraní IEnumerable a IEnumerator

- ```
  public interface IEnumerable
  {
    IEnumerator GetEnumerator();
  }
  ```

- ```
  public interface IEnumerator
  {
    bool MoveNext();
    object Current {get;}
    void Reset();
  }
  ```


Kolekce - Příklad procházení kolekce

- ```
  MyCollection myCollection = new MyCollection()
  ...
  //IEnumerator usage, insert your type instead of XXX
  IEnumerator ie = myCollection.GetEnumerator();
  while(ie.MoveNext())
  {
    XXX  item = (XXX) ie.Current;
    Console.WriteLine(item);
    ...
  }
  ```
- ```
  MyCollection myCollection = new MyCollection()
  ...
  //usage of foreach, insert your type instead of XXX
  foreach(XXX item in myCollection)
  {
    Console.WriteLine(item);
    ...
  }
  ```

Kolekce - Rozhraní IDictionaryEnumerator

- Používá se u slovníkových datových struktur.
- Standardizované rozhraní používané k postupnému procházení obsahu slovníku.
- Každý element má klíč a hodnotu.

```
public interface IDictionary :
IEnumerator
{
  DictionaryEntry Entry {get;}
  object Key {get;}
  object Value {get;}
}
```


Kolekce - Standardní rozhraní kolekcí (1)

- Rozhraní ICollection.
  - Standardní rozhraní pro počitatelné kolekce.
  - Umožňuje určit velikost, možnost změny, synchronizaci kolekce a podobně.
  - ICollection rozšiřuje IEnumerable.

```
public interface ICollection : IEnumerable
{
  void copyTo(Array array, int index);
  int Count {get;}
  bool IsReadOnly {get;}
  bool IsSynchronized {get;}
  object SyncRoot {get;}
}
```

Kolekce - Standardní rozhraní kolekcí (2)

- Rozhraní IList
  - Standardní rozhraní pro indexované kolekce.
  - Umožňuje indexovat prvky pomocí pozice.
  - Lze odstraňovat, přidávat a měnit prvky kolekce.

```
public interface IList : ICollection, IEnumerable
{
  object this[int index] {get; set;}
  int Add(object o);
  void Clear();
  bool Contains(object value);
  int IndexOf(object value);
  void Insert(int index, object value);
  void Remove(object value);
  void RemoveAt(int index);
}
```


Kolekce - Standardní rozhraní kolekcí (3)

- Rozhraní IDictionary
  - Standardní rozhraní pro kolekce používající dvojice klíč/hodnota.
  - Podobá se IList, ale umožňuje přistupovat k prvkům na základě klíče.

```
public interface IDictionary : ICollection,
IEnumerable
{
  object this[object key] {get; set;}
  ICollection Keys {get;}
  ICollection Values {get;}
  void Clear();
  bool Contains(object value);
  IDictionaryEnumerator GetEnumerator();
  void Remove(object key);
}
```

Kolekce - Předdefinované třídy kolekcí (1)

- Třída Array
  - Datová struktura představující pole pevné velikosti odkazů na objekty jednotného typu.
  - Implementuje rozhraní ICollection, IEnumerable a IList.
  - Nabízí možnost řazení a prohledávání pole.

```
string[] strs1 = { "now", "time", "right", "is" };
Array.Reverse(strs1);
Array strs2 = Array.CreateInstance(typeof(string), 3);
strs2.SetValue("for", 0);
strs2.SetValue("all", 1);
strs2.SetValue("people", 3);
Array strings = Array.CreateInstance(typeof(string), 8);
Array.Copy(strs1, strings, 4);
strs2.CopyTo(strings, 4);
foreach(string s in strings)
  Console.WriteLine(s);
```


Kolekce - Předdefinované třídy kolekcí (2)

- Třída ArrayList
  - Dynamické pole objektů implementující rozhraní IList.
  - Udržuje interní pole objektů - nahrazení větším polem při naplnění.
  - Třída je efektivní při vkládání objektů.

```
ArrayList a = new ArrayList();
a.Add("Vernon");
a.Add("Corey");
a.Add("William");
a.Add("Muzz");
a.Sort();
for(int i = 0; i < a.Count; i++)
   Console.WriteLine(a[i]);
```

Kolekce - Předdefinované třídy kolekcí (3)

- Třída Hashtable
  - Standardní slovníková datová struktura.
  - Používá hašovací algoritmus k ukládání a indexování hodnot.

```
Hashtable ht = new Hashtable();
ht["one"] = 1;
ht["two"] = 2;
ht["three"] = 3;
Console.WriteLine(ht["two"]);
```


Kolekce - Předdefinované třídy kolekcí (4)

- Třída Queue
  - Datová struktura reprezentující frontu (FIFO - First In First Out).
  - Operace Enqueue a Dequeue.
- Třída Stack
  - Datová struktura reprezentující zásobník (LIFO - Last In First Out).
  - Operace Push a Pop.
- Třída BitArray
  - Dynamické pole hodnot bool.

Kolekce - Předdefinované třídy kolekcí (5)

- Třída SortedList
  - Slovníková datová struktura implementující rozhraní IDictionary.
- Třída StringCollection
  - Datová struktura kolekce pro ukládání řetězců implementující rozhraní ICollection.
- Třída StringDictionary
  - Slovníková datová struktura.
  - Nabízí podobné metody jako třída Hashtable.
  - Implementuje standardní rozhraní IEnumerable.

Kolekce – Vytvoření nové kolekce (1)

```
public class MyCollection : IEnumerable
{
    int[] data;
    public virtual IEnumerator GetEnumerator()
    {
            return new MyCollection.Enumerator(this);
    }

    private class Enumerator:IEnumerator
    {
            MyCollection outside;
            int actualIndex = -1;
            internal Enumerator(MyCollection outside)
            {
            this.outside = outside;
            }
```

Kolekce – Vytvoření nové kolekce (2)

```
        public object Current
        {
                get
                {
                        if (actualIndex == outside.data.Length) throw new
                                InvalidOperationException();
                        return outside.data[actualIndex];
                }
        }
        public bool MoveNext()
        {
                if (actualIndex > outside.data.Length) throw new
                        InvalidOperationException();
                return ++actualIndex < outside.data.Length;
        }
        public void Reset() { actualIndex = -1; }
    }
}
```


Kolekce - Vytvoření nové kolekce (3)

- Ve verzi 2.0 přibyla možnost použití *iterátorů*.
  - Procházení kolekcí je řešeno na bázi přepínání kontextu.

```
public IEnumerator GetEnumerator()
{
        for(int i=0; i<pos; i++)
        {
                yield return data[i];
        }
}
```

  - Použití:

```
foreach(string v in s)
{
  Console.WriteLine(v);
}
```

Kolekce - Vytvoření nové kolekce (4)

- Iterátory lze využít i jinak, můžeme například implementovat takovouto metodu.

```
IEnumerable Power(int N)
{
    int counter = 0; int result = 1;
    while(counter++ < N)
    {
          result *= 2;
          yield return result;
    }
}
```

- Iterátor použijeme takto:

```
foreach(int i in Power(10))
    Console.WriteLine(i);
```


Kolekce – Třídění instancí (1)

- Schopnosti řazení a prohledávání kolekcí závisí na prvcích obsažených v kolekci.
- K porovnávání se využívá vygenerované číslo, tzv. hešovací kód (hashcode).
- Rozhraní IComparable
  - Umožňuje jednomu objektu indikovat své pořadí vzhledem k jiné instanci téhož typu.

```
public interface IComparable
{
  int CompareTo(object rhs);
}
```

Kolekce – Třídění instancí (2)

- Sémantická pravidla:
  - a patří před b, pak a.CompareTo(b) < 0;
  - a patří za b, pak a.CompareTo(b) > 0;
  - a je rovno b, pak a.CompareTo(b) = 0;
  - null je první: a.CompareTo(null) > 0;
  - když a.CompareTo(b), pak a.GetType() == b.GetType() .
- Rozhraní IComparer
  - Implementace tohoto rozhraní provádí porovnávání (či řazení).
  - Obsahuje jedinou metodu int Compare(object x, object y).

Kolekce – Třídění instancí (3)

```
public sealed class Person : IComparable
{
    public string Name;
    public int Age;
    public int CompareTo(object o)
    {
        if (o == null) return 1;
        if (o.GetType() != this.GetType())
            throw new ArgumentException();
        Person rhs = o as Person;
        if (Age < rhs.Age) return 1;
        if (Age < rhs.Age) return -1;
        return 0;
    }
}
```

Kolekce – Generické kolekce

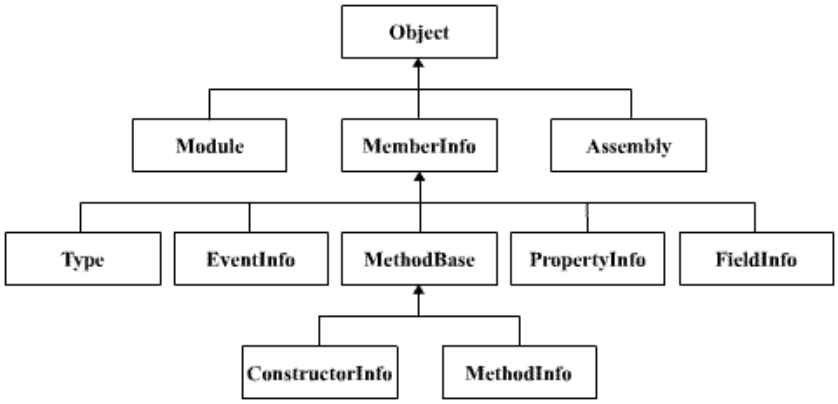
Generické kolekce	Negenerické
List<T>	ArrayList
SortedList<TKey, TValue>	SortedList
Dictionary<TKey, TValue>	Hashtable
SortedDictionary<TKey, TValue>	SortedList
Stack<T>	Stack
Queue<T>	Queue
LinkedList<T>	(není)

Reflexe (1)

- Pozdní vazba, serializace, vzdálené řízení, atributy a podobně, závisejí na přítomnosti metadat.
- Reflexe je prozkoumávání existujících typů prostřednictvím metadat.
- Uskutečňuje se sadou typů v oboru názvů `System.Reflection`.
- Je možné dynamicky vytvářet nové typy pomocí tříd v oboru názvů `System.Reflection.Emit`.
- Reflexe představuje procházení a manipulování s objektovým modelem aplikace.

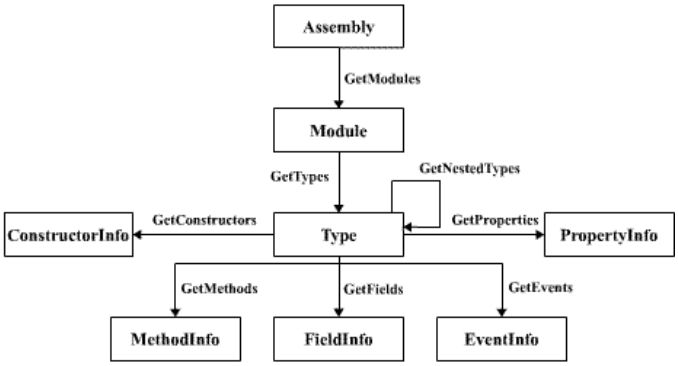
Reflexe (2)

- Vztahy dědičnosti mezi reflexními typy .NET



Reflexe (2)

- Pohyb hierarchií reflexe .NET



Reflexe – Type (1)

- Třída `Type` je nejzákladnějším typem reflexe a reprezentuje metadata pro jednotlivé deklarace typů v aplikaci.
- `Typy` obsahují členy, které zahrnují:
  - konstruktory;
  - proměnné;
  - vlastnosti;
  - události;
  - metody;
  - vnořené typy.
- Typy jsou seskupené do modulů.
- Moduly jsou obsažené v sestavách (assembly).

Reflexe – Type (2)

- V jádru systému reflexe je `System.Type`.
- K instanci třídy `Type` lze přistoupit pomocí metody `GetType()`.
  - Metoda vrátí konkrétní typ `System.Type`, který dokáže typ reflektovat a manipulovat s ním.

```
Type t = (new MyClass()).GetType()
```

- Instanci třídy Type lze převzít pomocí názvu prostřednictvím statické metody GetType().

```
Type t = Type.GetType("System.Int32");
Type t2 = Type.GetType("MyNamespace.MyType",
MyAssembly);
Type t = typeof(System.Int32);
```

Reflexe – Příklad použití

```
using System;
using System.Reflection;

class Test
{
  static void Main()
  {
    object o = new Object();
    Information(o.GetType());
    Information(typeof(int));
    Information(Type.GetType("System.String"));
  }
  static void Information(Type t)
  {
    Console.WriteLine("Type: {0}", t);
    MemberInfo[] miarr = t.GetMembers();
    foreach(MemberInfo mi in miarr)
      Console.WriteLine(" {0}={1}", mi.MemberType, mi);
  }
}
```


Reflexe – Type (2)

- V jádru systému reflexe je `System.Type`.
- K instanci třídy `Type` lze přistoupit pomocí metody `GetType()`.
  - Metoda vrátí konkrétní typ `System.Type`, který dokáže typ reflektovat a manipulovat s ním.

```
Type t = (new MyClass()).GetType()
```

- Instanci třídy Type lze převzít pomocí názvu prostřednictvím statické metody GetType().

```
Type t = Type.GetType("System.Int32");
Type t2 = Type.GetType("MyNamespace.MyType",
MyAssembly);
Type t = typeof(System.Int32);
```

Reflexe – Další vlastnosti

- Pozdní vazba
  - Dynamické vytváření instancí.
  - Použití typu za běhu.
  - Assembly.LoadFrom() - načítá dynamicky sestavu.
  - Activator.CreateInstance() - vytváření instancí.
- Vytváření nových typů za běhu
  - Obor názvů System.Reflection.Emit obsahuje třídy, které dokáží vytvořit za běhu úplně nové typy. Třídy umožňují:
    - definovat dynamickou sestavu v paměti;
    - definovat dynamický modul v této sestavě;
    - definovat nový typ v tomto modulu, včetně všech jeho členů;
    - vytvořit kódy MSIL potřebné k implementování aplikační logiky ve členech.

Reflexe – Komplexní příklad (1)

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ConsoleApplication1
{
    public class ReflectionEmitDemo
    {
        public Assembly CreateAssembly()
        {
            AssemblyName assemblyName = new AssemblyName();
            assemblyName.Name = "Math";

            AssemblyBuilder CreatedAssembly =
                 AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
                 AssemblyBuilderAccess.RunAndSave );

            ModuleBuilder AssemblyModule =
               CreatedAssembly.DefineDynamicModule("MathModule","Math.dll");
```

Reflexe – Komplexní příklad (2)

```
   TypeBuilder MathType = AssemblyModule.DefineType("DoMath",
        TypeAttributes.Public | TypeAttributes.Class);
   System.Type [] ParamTypes = new Type[] { typeof(int),typeof(int) };
   MethodBuilder SumMethod = MathType.DefineMethod("Sum",
        MethodAttributes.Public, typeof(int), ParamTypes);
   ParameterBuilder Param1 = SumMethod.DefineParameter(1,
        ParameterAttributes.In, "num1");
   ParameterBuilder Param2 = SumMethod.DefineParameter(2,
        ParameterAttributes.In, "num2");

    ILGenerator ilGenerator = SumMethod.GetILGenerator();
    ilGenerator.Emit(OpCodes.Ldarg_1);
    ilGenerator.Emit (OpCodes.Ldarg_2);
    ilGenerator.Emit (OpCodes.Add );
    ilGenerator.Emit(OpCodes.Ret);
    MathType.CreateType();

    return CreatedAssembly;
}}}
```

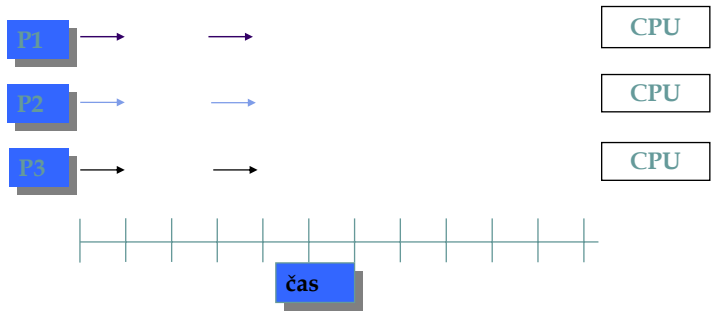

Reflexe – Komplexní příklad (3)

```
static void Main()
{
   ReflectionEmitDemo EmitDemo = new ReflectionEmitDemo();
    Assembly EmitAssembly = EmitDemo.CreateAssembly();
   System.Type MathType = EmitAssembly.GetType("DoMath");
    object[] Parameters = new object [2];
    Parameters[0] = (object) (5);
    Parameters[1] = (object) (9);
    object EmitObj = Activator.CreateInstance
   (MathType,false);
    object Result = MathType.InvokeMember("Sum",
    BindingFlags.InvokeMethod ,null,EmitObj,Parameters);
    Console.WriteLine("Sum of {0}+{1} is {2}",
        Parameters[0],Parameters[1],Result.ToString());
   //Sum of 5+9 is 14
    Console.ReadLine();
}
```

Vlákna - Základní pojmy

- **Proces (úloha – task)**
  - Úplně separátní program s vlastními proměnnými a vlastním adresovým prostorem.
  - O běh procesu se stará operační systém.
- **Multitasking –** schopnost operačního systému provádět současně více procesů.
- **Vlákno**
  - „odlehčený proces“
  - proces se může skládat z více vláken, všechny sdílejí stejný adresový prostor
  - vytvoření procesu je časově mnohem náročnější

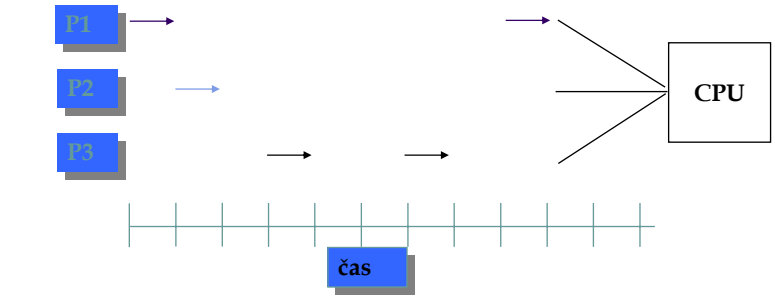
Vlákna - Provádění aplikace s více vlákny (1)



Počet prováděných procesů <= počtu procesorů

Vlákna - Provádění aplikace s více vlákny (2)

- Souběžné zpracování úloh



Počet prováděných procesů > počtu procesorů

Procesy

- Třída **System.Diagnostics.Process**
  - Monitorování
    - **int BasePriority{get;}**
    - **int ExitCode{get;}**
    - **int PagedMemorySize{get;}**
    - **ProcessThreadCollection Threads {get;}**
  - Spuštění
    - **ProcessStartInfo StartInfo {get; set;}**
      - **string FileName{get; set;}**
      - **string Arguments{get; set;}**
      - **bool UseSchellExecute {get; set;}**
    - **Start()**
  - Ukonční
    - **Kill()**
    - **bool CloseMainWindow()**

Vlákna

- Aplikace může "běžet" v jednom nebo více vláknech.
- Příklad vícevláknové aplikace:

```
using System;
using System.Threading;

class ThreadTest
{
  static void Main(){
    Thread t = new Thread(new ThreadStart(Run));
    t.Start();
    Run();
   }
  static void Run()
  {
    for(char c='a'; c<'z'; c++)
      Console.Write(c);
  }
}
```


Vlákna – Řízení vlákna

- Vlákno se po vytvoření spustí metodou **Start()**

```
Thread t = new Thread(new ThreadStart(metoda));
t.Start();
```

  - **Start()** lze volat jen jednou, jinak je vyvolána výjimka **ThreadStateException.**
  - Po zavolání **Start()** se vlákno dostává do stavu **ThreadState.Running**
- Běžící vlákno může být
  - Uspáno

```
static void Sleep(int)
```

    - Pouze vlákno samo se může uspat
  - Ukončeno

```
void Abort()
```

    - Výjimka **ThreadAbortException**
    - Voláním **static void ResetAbort()** lze zrušení přerušit.

Vlákna – Řízení vláken

- Třída Thread má celou řadu metod pro řízení a získávání informací o vláknech.
  - Některé vlastnosti třídy Thread.
    - IsAlive – zda vlákno ještě pracuje.
    - ThreadState - aktuální stav vlákna.
    - CurrentCulture – nastavení národního prostředí.
    - IsBackground
      - Vlastnost jak pro čtení tak pro zápis. Její hodnota indikuje, že dané vlákno běží na pozadí.
      - Aplikace skončí, pokud skončí všechny vlákna, která neběží na pozadí.

Vlákna – Řízení vlákna

- Metoda **void Join()** způsobí blokaci volajícího vlákna po dobu běhu vlákna, nad nímž byla byla **Join** volána.
- Metoda **void Interrupt()** probudí vlákno ve stavu Wait, Sleep, Join
- **ThreadPriority Priority {get; set;} -** Nastavení priority vlákna
  - **ThreadPriority.AboveNormal**
  - **ThreadPriority.BelowNormal**
  - **ThreadPriority.Highest**
  - **ThreadPriority.Lowest**
  - **ThreadPriority.Normal**

Vlákna - Synchronizace vláken (1)

- Technika zajišťující koordinovaný přístup ke sdíleným prostředkům.
- Příkaz lock
  - K bloku kódu může přistoupit pouze jedno vlákno.

```
class LockTest
{
  static void Main()
  {
    LockTest lt = new LockTest();
    Thread t = new Thread(new ThreadStart(lt.Run));
    t.Start();
    lt.Run();
  }
  public void Run()
  {
    lock(this) {
      for(char c='a'; c<'z'; c++)
        Console.Write(c);
    }
  }
}
```


Vlákna - Synchronizace vláken (2)

- Operace Pulse a Wait
  - Umožňuje vláknům navzájem komunikovat prostřednictvím monitoru.

```
class MonitorTest{
   static void Main(){
      MonitorTest mt = new MonitorTest();
      Thread t = new Thread(new ThreadStart(mt.Bez));
      t.Start();
      mt.Bez();
   }
   static void Bez(){
      for(char c='a'; c<'z'; c++)
         lock(this){
            Console.Write(c);
            Monitor.Pulse();
            monitor.Wait();
         }
   }
}
```

Vlákna – Automatická synchronizace vláken

- Atribut **MethodImplAttribute** (**System.Runtime.CompilerServices**)
- Metoda ozančená atributem **MethodImplAttribute** s parametrem **MethodImplOptions.Synchronized** se celá stáva kiritickou sekcí

```
[MethodImpl(MthodImplOptions.Synchronized)]
public void CriticalSection(int parameter)
{
    …
}
```


Vlákna – Prostředky synchronizace(3)

- Třída Monitor
  - Jde o nejzákladnější třídu vláken.
  - System.Threading.Monitor poskytuje implementaci Hoareho monitoru.
- Metody Enter a Exit
  - Získávají (resp. uvolňují) zámek nějakého objektu.
  - Enter() čeká dokud není uvolněn zámek.
  - Každému volání Enter() by mělo odpovídat volání Exit().
  - TryEnter(object) – test zda lze získat zámek na objekt.
- Metoda PulseAll
  - Pulse() probudí jen první vlákno ve frontě.
  - Metoda PulseAll() postupně probudí všechny vlákna ve frontě.
- Existují další třídy pro synchronizaci vláken: Interlocked, Mutex,…

Asynchronní delegáti (1)

- Runtime nabízí standardní způsob jak asynchronně volat metody.
  - `návratový-typ Invoke(seznam-parametrů);`
  - Volá metodu synchronně, volající musí čekat až delegát skončí vykonávání.
  - `IAsyncResult BeginInvoke(seznam-parametrů, IAsyncCallback ac, object stav);`
    - BeginInvoke volá delegáta se zadaným seznamem parametrů a pak se okamžitě vrací.
  - `návratový-typ EndInvoke(ref/out seznam-parametrů, IAsyncCallback ac);`
    - EndInvoke přebírá návratovou hodnotu volané metody se všemi odkazovanými parametry a výstupními parametry.

Asynchronní delegáti (2)

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

delegate int Compute(string s);

class Class1
{

  static int TimeConsumingFunction(string s)
  {
    return s.Length;
  }
  static void ViewResultFunction(IAsyncResult ar)
  {
    Compute c = (Compute)(((AsyncResult)ar).AsyncDelegate);
    int result = c.EndInvoke(ar);
    string s = (string)ar.AsyncState;
    Console.WriteLine("{0} contains {1} chars", s, result);
  }
```

Asynchronní delegáti (3)

```
 static void Main()
  {
    Compute c = new Compute(TimeConsumingFunction);

    AsyncCallback ac = new
AsyncCallback(ViewResultFunction);

    string s1 = "Christopher";
    string s2 = "Nolan";

    IAsyncResult ar1 = c.BeginInvoke(s1, ac, s1);
    IAsyncResult ar2 = c.BeginInvoke(s2, ac, s2);

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