ROZHRANÍ – POKRAČOVÁNÍ

Předdefinovaná rozhraní – pokračování

IDisposable – pokračování

Automatická správa paměti

Instance referenčních typů se vytvářejí na haldě. Platforma .NET používá tzv. *řízenou haldu*.

Řízená halda je oblast paměti v rámci virtuálního paměťového prostoru vyhrazeného danému procesu, která funguje podobně jako zásobník. Je rozdělena na dva bloky: blok obsazené a volné paměti. Paměť se přiděluje z vrcholu haldy, to znamená, že přidělení paměti pro objekt je velmi rychlé.

Při spuštění úklidu paměti se z haldy nejprve odstraní všechny objekty, na které se neodkazuje žádná reference. Po této operaci budou v haldě roztroušeny objekty, mezi kterými budou bloky volné paměti. Potom automatická správa paměti zkomprimuje haldu tak, že přesune všechny zbývající objekty tak, aby tvořily jediný souvislý blok obsazené paměti. Přitom aktualizuje všechny reference na tyto objekty tak, aby obsahovaly jejich nové adresy.

Spuštění úklidu paměti lze provést i explicitně zavoláním metody `Collect` třídy `GC`, např. po zrušení odkazů na mnoho objektů.

IFormattable

```
public interface IFormattable {
   string ToString (string format, IFormatProvider formatProvider);
}
```

Toto rozhraní implementují typy, které umožňují formátování při konverzi na řetězec znaků. Jedná se např. o všechny základní datové typy v C#. Bližší informace viz samostatná kapitola.

IEnumerable

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

Toto rozhraní je využíváno příkazem `foreach`. Přesněji kolekce, která se má procházet pomocí příkazu `foreach`, musí implementovat toto rozhraní.

Metoda `GetEnumerator` vrací enumerátor, tj. objekt, který umožňuje procházet jednotlivé prvky kolekce od prvního k poslednímu.

IEnumerator

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

Popis složek tohoto rozhraní – viz dříve kapitola „Příkaz foreach“.

Další rozhraní

- `ICollection` – implementují jej všechny kolekce včetně polí. Poskytuje metodu `CopyTo` pro kopírování části kolekce, vlastnost `Count` udávající počet prvků kolekce aj.

- `IList` – implementují jej kolekce, s nimiž lze zacházet pomocí indexování, podobně jako s poli, jako např. třída `ArrayList`.
- `IDictionary` – obsahuje vlastnosti typické pro datovou strukturu zvanou slovník (v C++ párový asociativní kontejner).
- Generická rozhraní – viz samostatná kapitola.

Iterátory

Rozhraní `IEnumerator` (a případně `IEnumerable`), které je využíváno v příkazu `foreach`, lze od verze .NET 2.0 implementovat také jednodušším způsobem než deklarováním nové třídy implementující toto rozhraní. Lze k tomuto účelu použít iterátory.

Iterátor (angl. *iterator*) je blok příkazů, který poskytuje posloupnost hodnot. Iterátor se liší od normálního složeného příkazu přítomností jednoho nebo více příkazů `yield`, který má následující syntaxi:

příkaz_yield:

yield return *výraz* **;**

yield break ;

První verze poskytuje následující hodnotu prováděné iterace.

Druhá verze indikuje, že iterace je kompletní.

Iterátor může být použit jako tělo metody, tělo přístupové metody vlastnosti nebo tělo přetíženého operátoru, přičemž návratový typem této metody, vlastnosti nebo operátoru musí být `IEnumerator` nebo `IEnumerable` nebo jejich generické obdoby.

Metoda nebo vlastnost, která je implementována pomocí iterátoru, může být předefinována, nebo v případě metody také přetížena, metodou nebo vlastností, která může ale nemusí být implementována pomocí iterátoru.

Příklad

```
class Zasobnik : IEnumerable
{
   object[] pole = new object[100];
   int pocet;

   public void Vloz(object i)
   {
      if (pocet < pole.Length) pole[pocet++] = i;
   }
   public object Odeber() { ... }
   public IEnumerator GetEnumerator()
   {
      for (int i = pocet - 1; i >= 0; --i)
         yield return pole[i];;
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      Zasobnik zasobnik = new Zasobnik();
      for (int i = 0; i < 10; i++) zasobnik.Vloz(i);
      foreach (int i in zasobnik) Console.Write("{0} ", i);
      Console.WriteLine();
   }
}
```

Třída `Zásobník` představuje zásobník libovolných prvků. Implementuje rozhraní `IEnumerable` deklarováním metody `GetEnumerator`. Tato metoda obsahuje iterátor, který prochází prvky zásobníku od posledního k prvnímu (od vrcholu ke spodku). Výstup programu bude následující:

```
9 8 7 6 5 4 3 2 1 0
```

Iterátory deklarované jako vlastnosti lze využít k procházení kolekce dalšími způsoby než nabízí implementovaná metoda `GetEnumerator`. Vlastnosti musí v takovém případě vracet rozhraní `IEnumerable`. Třída `Zasobnik` může být rozšířena o dvě vlastnosti.

```
class Zasobnik : IEnumerable
{
   // dříve uvedené složky

   public IEnumerable OdVrcholu
   {
      get { return this; }
   }
   public IEnumerable OdSpodku
   {
      get {
         for (int i = 0; i < pocet; i++)
            yield return pole[i];
      }
   }
}
class Program
{
   static void Main(string[] args)
   {
      Zasobnik zasobnik = new Zasobnik();
      for (int i = 0; i < 10; i++) zasobnik.Vloz(i);
      foreach (int i in zasobnik.OdSpodku) Console.Write("{0} ", i);
      Console.WriteLine();
      foreach (int i in zasobnik.OdVrcholu) Console.Write("{0} ", i);
      Console.WriteLine();
   }
}
```

Vlastnost `OdVrcholu` poskytuje rozhraní pro procházení prvků zásobníku od vrcholu ke spodku, tj. stejně jako rozhraní, které vrací metoda `GetEnumerator`. Tato vlastnost může tedy pouze vrátit `this`, jelikož třída `Zasobnik` implementuje rozhraní `IEnumerable` s požadovaným způsobem iterace.

Vlastnost `OdSpodku` poskytuje rozhraní pro procházení prvků zásobníku od spodku k vrcholu. Přístupová metoda `get` je implementována pomocí iterátoru. Výstup programu bude následující:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

Uvedené vlastnosti mohou být také použity mimo příkaz `foreach`

```
class Program
{
   static void Vypis(IEnumerable kolekce)
   {
      foreach (int i in kolekce) Console.Write("{0} ", i);
      Console.WriteLine();
   }
   static void Main(string[] args)
   {
      Zasobnik zasobnik = new Zasobnik();
      for (int i = 0; i < 10; i++) zasobnik.Vloz(i);
      Vypis(zasobnik.OdSpodku);
      Console.ReadKey();
   }
}
```

Iterátor může obsahovat i metoda, která má parametry. Nesmí se však jednat o parametry s modifikátorem `ref` nebo `out`.

Příklad

```
class Program
{
   static IEnumerable Posloupnost(int od, int @do, int krok)
   {
      for (int i = od; i <= @do; i += krok) {
         yield return i;
      }
   }
   static void Vypis(IEnumerable kolekce)
   {
      foreach (int i in kolekce) Console.Write("{0} ", i);
      Console.WriteLine();
   }
   static void Main(string[] args)
   {
      Vypis(Posloupnost(10, 20, 2));
      Console.ReadKey();
   }
}
```

Výpis programu bude následující

```
10 12 14 16 18 20
```

Při každém volání metody `GetEnumerator` poskytuje iterátor samostatný objekt enumerátoru. Tento objekt si pamatuje případné aktuální hodnoty formálních parametrů metody, která obsahuje daný iterátor.

Příklad

```
class Program
{
   static IEnumerable Posloupnost(int od, int @do)
   {
      while (od <= @do) yield return od++;
   }
   static void Main(string[] args)
   {
      IEnumerable ie = Posloupnost(1, 5);
      foreach (int x in ie) {
         foreach (int y in ie) {
            Console.Write("{0,3} ", x*y);
         }
         Console.WriteLine();
      }
      Console.ReadKey();
   }
}
```

Příklad vypíše tabulku násobků čísel 1 až 5. Metoda `Posloupnost` je vyvolána jen jednou k vytvoření objektu rozhraní `ie`. Avšak metoda `ie.GetEnumerator()` je volána dvakrát dvěma vnořenými příkazy `foreach`. Vytvoří se tudíž dva samostatné objekty enumerátorů. Tyto objekty zapouzdřují kód iterátoru metody `Posloupnost`, jenž modifikuje parametr `od`. Objekty enumerátorů si však uchovávají svojí vlastní kopii parametrů `od` a `@do`. Výpis programu bude následující:

```
  1   2   3   4   5
  2   4   6   8  10
  3   6   9  12  15
  4   8  12  16  20
  5  10  15  20  25
```

VÝJIMKY

Základní odlišnosti při práci s výjimkami v C# oproti C++:

- K bloku `try` může být kromě handlerů připojena ještě koncovka `finally`, která se používá ve strukturovaných výjimkách jazyka C v prostředí Win32 API.
- K přenášení informací o výjimce lze použít pouze instanci třídy odvozené od třídy `System.Exception`.
- Neprovádí se automatické volání destruktorů lokálních instancí objektových typů.
- V deklaraci metody nelze specifikovat typy výjimek, které se z ní mohou rozšířit.
- Blok `try` nemůže tvořit tělo metody nebo konstruktoru.

Syntaxe

Výjimka se vyvolá příkazem `throw`. Jeho syntaxe je podobná jako v C++.

Syntaxe:

příkaz throw:

throw *výraz*$_{\text{nep}}$ **;**

Výraz – výraz, jehož výsledkem je instance třídy `System.Exception` nebo jejího potomka.

Typicky jde o vytvoření nové instance pomocí operátoru `new`, např.

```
throw new Exception("Nastala výjimka");
```

Výraz lze vynechat, pokud je příkaz `throw` v těle handleru. Význam příkazu `throw` s vynechaným výrazem je stejný jako v C++, tj. znovu vyvolá výjimku, kterou handler zachytil.

Typ *výrazu* se nazývá *typ výjimky*.

Příkazy, které mohou vyvolat výjimku, jsou uvedeny v bloku `try`. Za blokem `try` může následovat jeden nebo několik handlerů a nejvýše jedna koncovka.

Celá tato konstrukce se v C# nazývá *příkaz try* (v jazyce C++ se nazývá *pokusný blok*).

Syntaxe:

příkaz_try:

try { *příkazy*$_{\text{nep}}$ **}** *seznam_handlerů*$_{\text{nep}}$ *koncovka*

try { *příkazy*$_{\text{nep}}$ **}** *seznam_handlerů*

seznam_handlerů:

handler seznam handlerů$_{\text{nep}}$

handler:

catch (*typ identifikátor*$_{\text{nep}}$ **)** *blok*

catch *blok*

Typ – typ výjimek, které daný handler může zachytit a ošetřit – *typ handleru*.

Identifikátor – identifikátor reference na instanci typu *typ*. Lze jej vynechat.

Blok – příkazy, které zachycenou výjimku ošetří, uzavřené do složených závorek.

Druhá varianta handleru, v níž chybí specifikace typu výjimky, představuje syntaktickou zkratku pro handler ve tvaru

```
catch (System.Exception) { ... }
```

Tento handler zachytí všechny výjimky – *univerzální handler*. V jeho těle nelze zjistit informace o výjimce. Univerzální handler musí být posledním v seznamu handlerů.

Koncovka je blok začínající klíčovým slovem `finally`.

Syntaxe:

koncovka:

finally *blok*

Blok – příkazy, které se provedou vždy, ať skončí blok `try` jakýmkoli způsobem.

Vznik a šíření výjimky

Vznik výjimky

Výjimku lze vyvolat příkazem `throw`. Výjimka může také vzniknout v důsledku kontroly celočíselného přetečení pomocí operátoru nebo příkazu `checked`, při nesprávném přetypování, při překročení mezí polí a v některých dalších situacích.

Také některé metody tříd knihovny BCL mohou vyvolat výjimku při jejich volání s nesprávnými parametry nebo v nesprávné situaci.

Výjimku lze vyvolat i v handleru nebo v koncovce.

Šíření výjimky

Vznikne-li výjimka, začne program hledat odpovídající handler. Přitom postupuje podobně jako v C++. Pokud program neobsahuje koncovky, činnost je následující.

- Po vzniku výjimky program přeskočí všechny zbývající příkazy v aktuálním bloku. Jedná-li se o blok `try`, začne po řadě hledat vhodný handler, který může výjimku ošetřit. O tom, zda handler výjimku zachytí, rozhoduje typ handleru. Přitom se uplatňuje pravidlo, že potomek může zastoupit předka. To znamená, že handler typu `T` zachytí výjimky typu `T` a typů, které jsou od typu `T` odvozeny.
- Vznikne-li výjimka v jiném bloku než bloku `try` nebo k němu není připojen vhodný handler, přejde program do dynamicky nadřízeného bloku. To znamená, že vznikne-li výjimka v těle metody, opustí program tuto metodu a přejde do metody, která ji zavolala. Opět přeskočí zbylé příkazy v aktuálním bloku a bude u něj hledat vhodný handler.
- Pokud program najde vhodný handler, přejde do něj a provede jeho příkazy. Po jejich vykonání přeskočí případné další handlery a bude pokračovat za příkazem `try`.
- Jestliže program nenajde vhodný handler ani v metodě `Main`, předá program výjimku prostředí .NET Framework, které vypíše zprávu o chybě a program ukončí.
- Po vstupu do handleru se výjimka považuje za ošetřenou. To znamená, že tělo handleru může být i prázdné.

Při šíření výjimky se opouštěné bloky korektně ukončí. To znamená, že se např. ze zásobníku odstraní všechny lokální proměnné. Tím mohou zaniknout i odkazy na instance tříd. Odstranění těchto instancí provede automatická správa paměti podle potřeby.

Program zkoumá jednotlivé handlery v pořadí, v jakém jsou uvedeny v příkazu `try`. Pokud příkaz `try` obsahuje handlery typů ve stejné dědické hierarchii, musí se jako první uvést handler nejvíce odvozeného typu, potom handler druhého nejvíce odvozeného typu atd. Jako poslední musí být uveden handler, jehož typ je předkem typů všech předchozích handlerů. Toto pravidlo platí i v C++. Na rozdíl od C++ se však v C# považuje jeho porušení za chybu.

Koncovka

Příkazy bloku `finally` se provedou vždy bez ohledu na to, zda blok `try` skončí normálně (přechodem přes ukončovací složenou závorku), příkazem skoku (`break`, `continue`, `goto` nebo `return`), jehož cíl leží mimo tento blok, nebo vznikem výjimky.

Příkazy koncovky se provedou okamžitě poté, co:

- blok `try` skončí normálně,
- blok `try` skončí příkazem skoku nebo
- se provedou příkazy vhodného handleru.

Příkazy `break`, `continue` a `goto` nesmějí přenést řízení ven z koncovky.

Příkaz `return` se nesmí v koncovce objevit.

Příklad

```
class Program
{
   const int n = 10;

   static void f(int i)
   {
      Console.WriteLine("Začátek f({0})", i);
      if (i == n) throw new Exception();
      Console.WriteLine("Konec f({0})", i);
   }
   static void Main(string[] args)
   {
      try {
         for (int i = 0; i < 2; i++) {
            Console.WriteLine("Před voláním f");
            f(i);
            Console.WriteLine("Za voláním f");
         }
      }
      catch (ArithmeticException) {
         Console.WriteLine("Aritmetická výjimka");
      }
      catch (Exception) {
         Console.WriteLine("Obecná výjimka");
      }
      Console.WriteLine("Konec programu");
      Console.ReadKey();
   }
}
```

Pokud konstanta `n` bude mít hodnotu 10, výjimka v programu nevznikne a výstup programu bude následující:

```
Před voláním f
Začátek f(0)
Konec f(0)
Za voláním f
Před voláním f
Začátek f(1)
Konec f(1)
Za voláním f
Konec programu
```

Pokud konstanta `n` bude mít hodnotu 1, při druhém volání funkce `f` vznikne výjimka typu `Exception` a výstup programu bude následující:

```
Před voláním f
Začátek f(0)
Konec f(0)
Za voláním f
Před voláním f
Začátek f(1)
Obecná výjimka
Konec programu
```

Příklad

```
class Program
{
   const int n = 1;

   static void f(int i)
   {
      Console.WriteLine("Začátek f({0})", i);
      if (i == n) throw new Exception();
      Console.WriteLine("Konec f({0})", i);
   }
   static void Main(string[] args)
   {
      try {
         try {
            for (int i = 0; i < 2; i++) {
               Console.WriteLine("Před voláním f");
               f(i);
               Console.WriteLine("Za voláním f");
            }
         }
         catch (ArithmeticException) {
            Console.WriteLine("Aritmetická výjimka");
         }
         finally {
            Console.WriteLine("Koncovka vnitřního bloku");
         }
      }
      catch {
         Console.WriteLine("Obecná výjimka");
      }
      finally {
         Console.WriteLine("Koncovka vnějšího bloku");
      }
      Console.WriteLine("Konec programu");
      Console.ReadKey();
   }
} // class Program
```

Výstup programu bude následující:

```
Před voláním f
Začátek f(0)
Konec f(0)
Za voláním f
Před voláním f
```

```
Začátek f(1)
Koncovka vnitřního bloku
Obecná výjimka
Koncovka vnějšího bloku
Konec programu
```

Třídy pro výjimky

Knihovna BCL obsahuje řadu předdefinovaných tříd, které lze použít jako instanci výjimky. Lze také deklarovat svojí třídu výjimky, která bude odvozena od třídy `System.Exception` nebo jeho potomka.

Třída Exception

Konstruktory

Třída má 3 veřejné konstruktory:

`public Exception()`

Vytvoří instanci této třídy s implicitním textem chyby závislého na aktuální kultuře.

`public Exception(string message)`

Vytvoří instanci s textem chyby `message`.

`public Exception(string message, Exception innerException)`

Vytvoří instanci s textem chyby `message` a nastaví tzv. vnitřní výjimku (viz dále).

Vlastnosti

`public virtual string Message { get; }`

Poskytuje text chyby, který do instance uložil její konstruktor.

`public virtual string StackTrace { get; }`

Poskytuje řetězec popisující obsah zásobníku v podobě seznamu volaných metod od vzniku výjimky po handler, který výjimku zachytil, přičemž nebere v úvahu rekurzivní volání téže metody.

`public MethodBase TargetSite { get; }`

Poskytuje informace o metodě (pomocí reflexe), která výjimku vyvolala.

`public virtual string Source { get; set; }`

Poskytuje nebo nastavuje název aplikace nebo objektu, který výjimku způsobil.

`public virtual string HelpLink { get; set; }`

Poskytuje nebo nastavuje název souboru nápovědy, v němž jsou další informace o dané výjimce.

`public virtual IDictionary Data { get; }`

Nová vlastnost od verze .NET 2.0, která poskytuje uživatelská data o výjimce uložená ve slovníku.

`public Exception InnerException { get; }`

Poskytuje referenci na instanci tzv. vnitřní výjimky.

Občas se stane, že v určitém místě programu se zachytí výjimka, kterou lze ošetřit jen částečně. K úplnému vyřešení problému je třeba znovu vyvolat tutéž nebo jinou výjimku.

Tutéž výjimku lze vyvolat příkazem

```
throw;
```

Je-li potřebné vyvolat výjimku jiného typu a přitom zachovat informace o původní výjimce, použije se konstruktor s parametrem vnitřní výjimky, kterému se předá právě ošetřovaná výjimka. Např.:

```
catch (ArithmeticException e) {
   throw new Exception("Něco se stalo", e);
}
```

Handler, který zachytí takto vyvolanou výjimku, může prostřednictvím vlastnosti `InnerException` získat informace o původní příčině chyby. Takto lze zřetězit i několik výjimek.

Příklad

```
class Matematika
{
   public static int Faktorial(int n)
   {
      int s = 1;
      while (n > 0) s = checked(s * n--);
      return s;
   }
}
class Program
{
   static void f(int cislo)
   {
      Console.WriteLine("{0}! = {1}", cislo,
         Matematika.Faktorial(cislo));
   }
   static void Main(string[] args)
   {
      try {
         f(100);
      }
      catch (Exception e) {
         Console.WriteLine(e.Message);
         Console.WriteLine(e.StackTrace);
         if (e.InnerException != null) {
            Console.WriteLine("Vnitřní výjimka:");
            Console.WriteLine(e.InnerException.Message);
            Console.WriteLine(e.InnerException.StackTrace);
         }
      }
      Console.ReadKey();
   }
}
```

Pokud se program spustí ve složce, v němž se nacházejí zdrojové soubory, výstup programu bude následující:

```
Arithmetic operation resulted in an overflow.
   at Vyjimky03.Matematika.Faktorial(Int32 n)
   at Vyjimky03.Program.f(Int32 cislo)
   at Vyjimky03.Program.Main(String[] args)
```

Jinak výpis bude obsahovat i soubory včetně cesty, v nichž se uvedené metody nacházejí.

Pokud se funkce `f` změní takto

```
static void f(int cislo)
{
   try {
      Console.WriteLine("{0}! = {1}", cislo,
         Matematika.Faktorial(cislo));
   }
   catch (OverflowException e) {
      throw new Exception("Nepovedlo se spočítat faktoriál", e);
   }
}
```

výstup programu bude následující:

```
Nepovedlo se spočítat faktoriál
   at Vyjimky03.Program.f(Int32 cislo)
   at Vyjimky03.Program.Main(String[] args)
Vnitřní výjimka:
Arithmetic operation resulted in an overflow.
   at Vyjimky03.Matematika.Faktorial(Int32 n)
   at Vyjimky03.Program.f(Int32 cislo)
```

Další třídy

Instance třídy `System.Exception` se k přenosu informací o výjimce používá jen zřídka. Většinou se používají odvozené třídy, které umožňují přesnější klasifikaci vzniklých chyb. Třída `System.Exception` má dva přímé potomky:

- `System.SystemException` – předek tříd pro práci s výjimkami, definovaných v prostoru jmen `System`. Jedná se o výjimky, které může vyvolat prostředí .NET nebo i vámi zapsaný kód.
- `System.ApplicationException` – základní třída pro aplikační nefatální chyby.

Od třídy `System.SystemException` je odvozena celá řada tříd výjimek. Mezi nejběžnější patří následující třídy:

- `System.ArithmeticException` – používá se pro výpočtové chyby. Má následující potomky:
  - `System.DivideByZeroException`– vznikne při dělení celého nebo desítkového čísla nulou.
  - `System.OverflowException` – vznikne při celočíselném přetečení, je-li kontrolováno (např. operátorem `checked`)
- `System.IndexOutOfRangeException` – vznikne, pokud index prvku pole je mimo meze tohoto pole.
- `System.TypeInitializationException` – vznikne, pokud se ze statického konstruktoru rozšíří výjimka. Vlastnost `InnerException` obsahuje výjimku, která se ze statického konstruktoru rozšířila.
- `System.RankException` – vznikne, pokud se použije identifikátor pole s nesprávným počtem rozměrů.
- `System.InvalidCastException` – vznikne při pokusu o nesprávné přetypování za běhu programu.

- `System.InvalidOperationException` – vznikne, je-li volání metody v daném kontextu chybné. Výjimku např. vyvolá metoda `MoveNext` enumerátoru, jestliže prvky kontejneru byly modifikovány, potom co byla vytvořena instance enumerátoru. Má mj. následujícího potomka:
  - `System.ObjectDisposedException` – vznikne při pokusu provést operaci se zlikvidovaným objektem, tj. s objektem, pro nějž byla volána metoda `Dispose`.
- `System.NullReferenceException` – vznikne při přístupu ke složce objektového typu pomocí reference mající hodnotu `null`.
- `System.ArgumentException` – je předkem výjimek, které vzniknou při volání metody s nesprávným parametrem. Kromě konstruktorů třídy `Exception` má další konstruktory s parametrem `paramName` typu `string`, který udává jméno parametru, který výjimku způsobil. Tento parametr je přístupný pomocí vlastnosti `ParamName`.

  Má následující potomky:
  - `System.ArgumentNullException` – parametr má hodnotu `null`.
  - `System.ArgumentOutOfRangeException` – parametr je mimo rozsah.
  - `System.ComponentModel.InvalidEnumArgumentException` – parametr výčtového typu s nesprávnou hodnotou.
- `System.NotImplementedException` – vznikne při volání metody, která není implementována.
- `System.ArrayTypeMismatchException` – vznikne při pokusu uložit do prvku pole hodnotu nesprávného typu.
- `System.OutOfMemoryException` – vznikne při nedostatku paměti pro pokračování v běhu programu.
- `System.FormatException` – vznikne v metodě určené pro formátování hodnoty (řetězce), pokud skutečné parametry neodpovídají zadanému formátu. Výjimku vyvolá např. metoda `Console.WriteLine` nebo `string.Format`, pokud počet skutečných parametrů neodpovídá počtu parametrů specifikovaných ve formátovacím řetězci.
- `System.StackOverflowException` – vznikne při přetečení paměti zásobníku. K přetečení může nastat např. při neustálém rekurzivním volání metody.
- `System.IO.IOException` – je předkem výjimek, které vznikají při vstupních a výstupních operacích. Má následující potomky:
  - `System.IO.DirectoryNotFoundException` – vznikne při přístupu k neexistujícímu adresáři.
  - `System.IO.EndOfStreamException` – vznikne při pokusu čtení za koncem souboru.
  - `System.IO.FileNotFoundException` – vznikne při přístupu k neexistujícímu souboru.
  - `System.IO.FileLoadException` – vznikne, pokud nelze načíst existující sestavení.
  - `System.IO.PathTooLongException` – vznikne, pokud jméno souboru nebo cesty je delší než systémem definované maximum.

Možnosti ladění výjimek v prostředí Visual Studio

Menu **Debug | Exceptions** vyvolá dialogové okno, obsahující seznam předefinovaných výjimek. U každé z výjimek jsou k dispozici dvě zaškrtávací políčka:

- **Thrown** – program se pozastaví, pokud dojde k vyvolání výjimky, a to na místě vzniku výjimky. Pro výjimky CLR není implicitně zaškrtnuto.

- **User-unhandled** – program se pozastaví, pokud došlo k vyvolání výjimky, pro níž nebyl nalezen vhodný handler, a to na místě vzniku výjimky. Pro téměř všechny výjimky CLR je implicitně zaškrtnuto.

Do seznamu výjimek lze přidat i uživatelem definovanou výjimku pomocí tlačítka **Add** nebo ji odstranit pomocí tlačítka **Delete**.

Doporučení

- Metody by v případě vzniku chyby neměly vracet informaci o chybě (kód chyby), ale měly by vyvolat výjimku.
- Veřejné a chráněné metody, které vyvolávají výjimky, by měly být opatřeny dokumentačním komentářem obsahujícím seznam výjimek s popisem případů jejich vzniku.
- Veřejná metoda by neměla obsahovat parametr, který by udával, zda se má nějaká výjimka vyvolat.
- Z koncovky by se neměla vyvolat výjimka explicitně, tj. uvedením příkazu `throw`. Může být vyvolána implicitně, jako výsledek volání nějaké metody.
- Při opětovném vyvolání výjimky v handleru, který ji zachytil, se má použít prázdný příkaz `throw` a ne příkaz `throw` s instancí vyvolávané výjimky, aby se zabránilo vzniku výjimky související se zásobníkem. Např.

```
try {
   f(10);
}
catch (ArithmeticException e) {
   // ...
   // throw e; // Nespravně
   throw; // Správně
}
```

- Nedoporučuje se vyvolávat výjimky typu `Exception` nebo `SystemException`. Měly by se vyvolávat výjimky odvozených typů, které odpovídají charakteru chyby.
- Výjimka typu `ArgumentException` a její potomci by měly být vytvořeny pomocí konstruktoru s parametrem `paramName`, který udává jméno parametru, jenž výjimku způsobil. Např.

```
if (paramA == null) {
   throw new ArgumentNullException("paramA", "text chyby");
   // Nebo jen jméno parametru:
   // throw new ArgumentNullException("paramA");
}
```

  Pokud se tato výjimka vyvolává z přístupové metody `set` vlastnosti, jako jméno parametru se má použít text `"value"`.

Uživatelem definované výjimky

- Uživatelem definované výjimky by měly být odvozeny od třídy `Exception` nebo od jiné běžné základní třídy pro výjimky. Před verzí .NET 2.0 bylo doporučováno odvozovat uživatelem definované výjimky od třídy `ApplicationException`, ale ukázalo se, že toto doporučení nemělo žádný význam, proto bylo zrušeno.
- Jméno uživatelem definované výjimky má mít příponu `Exception`, tak jako je tomu u předdefinovaných tříd výjimek.
- Výjimky by měly být serializovatelné – viz kapitola o serializaci.

- Výjimky by měly obsahovat alespoň konstruktory třídy `Exception` (tři veřejné a jeden chráněný, určený pro serializaci).