

# Výjimky a zpracování chyb v .NET/C#

Zpracování chyb

Výjimky

Logování

Code Contracts

# Chybové kódy

- nic nenutí k ošetření
- musí se ošetřit na místě
- můžou kolidovat s návratovou hodnotou
  - např. `atoi`

# Výjimky

- jsou objektově orientované
- oddělují zpracování chyb od výkonného kódu
- nesou více informací
- zlepšují čitelnost programu
- lze je ošetřit daleko od místa vzniku
- vyvolání výjimky je náročná operace
  - = drahá
  - používat jen pro výjimečné, nikoliv běžné, události

# Výjimky v .NET/C#

- každá výjimka je objekt
  - všechny výjimky jsou odvozeny z třídy `System.Exception` nebo z některé z odvozených tříd
  - tvoří hierarchii
- každá výjimka obsahuje informace o svém původu a důvodu vzniku
- nejsou hlídané (checked)
  - hlídané výjimky často vedou ke špatně napsaným nebo prázdným blokům try-catch
  - na druhou stranu ale mnohy chybí dokumentace jejich existence

# Vlastnosti třídy System.Exception

Message	<code>string</code>	text popisující proč byla výjimka vyhozena
Source	<code>string</code>	jméno assembly, kde výjimka vznikla
StackTrace	<code>string</code>	posloupnost metod, které vedly k vyvolání výjimky
InnerException	<code>Exception</code>	pokud byla výjimka vyhozena v důsledku zpracování jiné výjimky, je původní výjimka obsažena zde
HelpLink	<code>string</code>	URL na soubor popisující chybu
Data	<code>IDictionary</code>	kolekce dalších uživatelských dat o výjimce

# Původní rozdělení výjimek

- .NET původně rozdělovat výjimky na dvě skupiny odvozené od `System.Exception`
  - `System.SystemException` pro výjimky CLR
  - `System.ApplicationException` pro výjimky uživatelských aplikací
- často se nedodrží ani v BCL
- toto rozdělení se dnes už považuje za **překonané**
  - Microsoft by se těchto tříd rád zbavil, ale to nejde kvůli zpětné kompatibilitě

# Výjimky pro parametry

- `System.ArgumentException`
  - byl předán parameter s neplatnou hodnotou
  - vyplňte vlastnost `ParamName` jménem parametru, který způsobil výjimku nebo `"value"` pokud se vyskytne v setteru vlastnosti (parameter se nastavuje konstruktorem)
- `System.ArgumentNullException`
  - hodnota parametru je `null`
- `System.ArgumentOutOfRangeException`
  - hodnota parametru je mimo povolený rozsah

# Aritmetické výjimky

- `System.ArithmeticException`
  - pokud nastala chyba při aritmetické operaci nebo převodu
- `System.DivideByZeroException`
  - nastane při dělení celých čísel nulou
- `System.OverflowException`
  - aritmetická operace nebo převod skončil přetečením výsledku
  - standardně se ale vznik přetečení nehlídá
    - lze ovlivnit nastavením projektu nebo explicitně klíčovými slovy `checked` a `unchecked`



# Další výjimky

- `System.InvalidOperationException`
  - metodu není možné vykonat protože objekt není ve platném/správném stavu
  - například zápis do proudu, který ještě nebyl otevřen
- `System.FormatException`
  - formát vstupu neodpovídá požadavkům: `int.Parse("veverka")`
  - nemusí to být nutně formát parametru, ale může být i vstup čtený ze souboru nebo zařízení

# Další výjimky

- `System.NotSupportedException`
  - obvykle metoda není podporována bázovou třídou a je na potomkovi, aby ji překryl
  - např. `NetworkStream.Seek`
- `System.NotImplementedException`
  - výjimka signalizující, že požadovaná metoda ještě nebyla implementována
  - typicky v automaticky vygenerovaném kódu

# Výjimky pro vstup/výstup

- `System.IO.IOException`
  - **bázová výjimka pro chyby při zpracování vstupu a výstupu**
- `System.IO.FileNotFoundException`
- `System.IO.DriveNotFoundException`
- `System.IO.PathTooLongException`

# Výjimky CLR

- tyto výjimky byste neměli vyhazovat
- `System.IndexOutOfRangeException`
  - pokoušíte se manipulovat v poli s prvkem jehož index je mimo rozsah pole
- `System.NullReferenceException`
  - pokoušíte se o dereferencování objektu, který je `null`
  - například se na něm pokoušíte volat metodu
- `System.InvalidCastException`
  - při pokusu o neplatné přetypování
  - `(int)"Veverka"`

# Výjimky CLR

- tyto výjimky nevyhazujte, chytat je nemá smysl
- `System.OutOfMemoryException`
  - není dostatek paměti pro pokračování programu
- `System.StackOverflowException`
  - pokud přeteče prováděcí zásobník vlákna
  - obvykle příliš mnoho volaných metod / rekurzivní volání
  - od verze .NET 2.0 nejde zachytit vůbec

```
public string BadProperty {  
    get { return badProperty; }  
    set { BadProperty = value; } // <-- StackOverflowException  
}
```

Klasický případ StackOverflowException

# Dokumentace výjimek

- v XML dokumentaci pomocí tagu `<exception cref="typ">důvod</exception>`
  - atribut `cref` obsahuje typ popisované výjimky
  - obsah elementu `exception` popisuje důvod pro vyhození výjimky

```
/// <summary>Nakrmí kočku.</summary>  
/// <exception cref="System.ArgumentException">  
///     kočka je mrtvá  
/// </exception>  
public void FeedSchrodingersCat(Cat c) { ... }
```

# Blok Try-Catch

- klíčové slovo **try**
  - uvozuje chráněný blok pro zachytání výjimek
- za blokem try může následovat blok klíčovým slovem **catch**
  - může být i více bloků catch k jednomu bloku try
  - obsahuje kód pro zvládnutí výjimky
  - bezprostředně za catch je typ chytané výjimky
    - jestliže typ není uveden, zachytává se `System.Exception` = všechny výjimky
  - je-li catch bloků více, výjimku **zpracuje první blok**, na jehož typ je výjimka přetypovatelná

# Blok Try-Catch

- jestliže při volání metody nastane výjimka
  - provádění metody je přerušeno
- pokud výjimka nastala v bloku try
  - je vyvolán kód v odpovídajícím bloku catch
- není-li nalezen odpovídající blok catch nebo není-li v bloku try
  - výjimka je propuštěna do vnějšího bloku try-catch nebo do volající metody
- jestliže výjimka není zachycena nikde
  - je detekována CLR a proces je násilně ukončen



```
try
{
    c = a / b;
}
catch(DivideByZeroException)
{
    Console.WriteLine("Nastalo dělení nulou.");
}
catch(ArithmeticException e)
{
    Console.WriteLine("Jiná aritmetická výjimka: {0}", e);
}
catch // nikdy nechytejte přímo Exception
{
    Console.WriteLine("Oops. Nastala úplně jiná výjimka");
}
```

# Blok Try-Finally

- výjimka může zanechat program v nekorektním stavu
- blok **finally**
  - může následovat po posloupnosti bloků try a catch
  - vykoná bez ohledu na to, jestli výjimka v blok try nastala
  - ale až po vykonání bloku catch
- v C# není dovoleno opustit blok finally jinak než celým jeho vykonáním, nebo vyvoláním výjimky
  - nelze použít return nebo goto

# Vyvolání výjimek

- výjimky se vyvolávají pomocí klíčového slova **throw** za nímž obvykle následuje instance výjimky

```
throw new ArgumentNullException("cat");
```

- Zachycenou výjimku v bloku catch můžete znovu vyhodit použitím **throw** bez udání instance výjimky
  - takto znovu vyhozená výjimka si zachová svůj call stack (StackTrace), kdežto u nově vyhozené výjimky její StackTrace začne od místa vyhození – tedy od catch bloku

```
... catch(ArgumentNullException ex) { Log(ex); throw; }
```

# Řetězení výjimek

- vytvořením nové výjimky v catch bloku s předáním původní výjimky
- tato původní výjimka je pak dostupná pomocí vlastnosti `InnerException` nové výjimky
- předávání informace o původní chybě
- zabalování výjimek pro dodržení kontraktu

```
... catch(FileNotFoundException ex)
{
    throw new DataSourceNotFoundException(
        "Datovy soubor nenalezen", ex);
}
```

# Vlastní výjimky

- podědit z třídy `System.Exception` nebo z některé jiné obecné třídy výjimek
- konvence pojmenování: název končí `Exception`
- přidat vlastnosti pro další informace pro další programové zpracování
  - **nedávejte důležité informace jen do vlastnosti `Message`**
- implementovat alespoň 4 základní konstruktory jako třída `Exception`
- výjimka by měla být serializovatelná
  - **např. označena atributem `SerializableAttribute`**
  - **aby se mohla přenášet mezi vzdálenými systémy**

# Kontrakt a skrývání detailů

- při navrhování rozhraní a bázových tříd se zamyslet i nad budoucím použitím
- kontrakt by neměl omezovat implementaci a seznam vyhazovaných výjimek je (by měl být) součástí kontraktu rozhraní
  - např. datový zdroj implementovaný nad soubory vyhazuje `FileNotFoundException` a ostatní třídy s tím počítají
  - nová implementace vytvořená později a nad relační databází vyhazuje `SQLException`
  - jenže s ní už třída používající datový zdroj nepočítá...
- řešením je zabalení výjimek závislých na implementaci do vlastních výjimek

# Best practices

- používejte výjimky jen pro výjimečné stavy
  - zpracování výjimky je náročnější než obyčejný `if`
  - pokud lze chybu očekávat často, je lepší ji řešit programově (pomocí `if` nebo i návratové hodnoty)
- někdy je dobré mít i jiný způsob kontroly
  - `int.TryParse`, `Stream.CanSeek`
- nikdy nevyhazujte přímo třídu `Exception`
- nikdy nepoužívejte prázdný blok `catch`
- výjimky detailně popište
- používejte odpovídající výjimky
  - pokud existuje vhodná výjimka v BCL, použijte ji

# Best practices

- normální užívání třídy by nikdy nemělo způsobit výjimku
  - např. metoda `File.Open` vrací `null` když soubor neexistuje, ale vyhazuje pokud je zamčený
- po vyhození výjimky po sobě uklidíte a vraťte změny tak, aby třída byla v konzistentního stavu
- logujte informace o výjimkách, které nastaly
  - používejte metodu `ToString()` místo vlastnosti `Message`
  - obsahuje více informací (`StackTrace`)



# Vývojevový vzor Disposable

# Vývojový vzor Disposable

- běhové prostředí .NET uvolňuje zdroje **nedeterministicky**
  - není určeno kdy přesně je zdroj uvolněn
  - obecně někdy po ztrátě poslední reference na něj
  - např. pokud proměnná vyjde z oboru platnosti
- neřízené zdroje je vhodné co nejdříve **uvolnit explicitně**
  - zavřít soubory
  - zavřít databázová spojení
  - vrátit systémové zdroje
- při práci s cennými zdroji je třeba vždy používat blok finally a tím zajistit jejich uvolnění i pokud nastane výjimka

# Vývojový vzor Disposable

- pro usnadnění práce s neřízenými zdroji byl zaveden vývojový vzor Disposable
- rozhraní `System.IDisposable` předepisuje jen metodu `Dispose`, která zajišťuje uvolnění všech zdrojů alokovaných třídou
- blok `using` je zjednodušením bloku `try-finally` s `IDisposable`
  - nepleťte si jej s direktivou `using <namespace>`
- sama existence metody `Dispose` ale stále nezaručuje její zavolání
  - je vhodné ji explicitně volat i v destruktoru
  - při implementaci je potřeba zajistit, aby se provedlo jen první volání `Dispose` a každé další se ignorovalo

```
Stream s = new FileStream(...)
try {
    DoWork(s); // dělej něco s proudem
}
finally {
    if (s != null)
        ((IDisposable)s).Dispose(); // na konci musíme uklidit
}
```

```
// následující kód je ekvivalentní kódu nahoře
using (Stream s = new FileStream(...))
{
    DoWork(s); // dělej něco s proudem
}
```

# Trasování programu

System.Diagnostics  
třídy Trace a Debug  
Trace Listeners  
Trace Switchers  
Trace Source

# Trasování

- jedná se o sledování stavu aplikace a vedení záznamu o její činnosti
- nikdy nejste schopni otestovat všechny možné kombinace
- chyby v reálném nasazení ale nelze obvykle ladit
- abyste mohli chybu reprodukovat u sebe je potřeba vědět co se stalo
- umístění kódu pro sledování chodu aplikace na strategická místa má umožnit rychleji zjistit, co se stalo, případně upozornit správce systému na možné problémy

# Konfigurace

- požadavky na trasování se často mění
  - podle nasazení systému: např. umístění záznamů
  - podle aktuální potřeby: normálně se zaznamenávají jenom chyby, při problémech se začne zaznamenávat všechno
- všechna nastavení lze provést buď
  - programově v kódu
  - deklarativně v konfiguračním souboru aplikace (Application Configuration File, `app.config`)
- při použití konfiguračního souboru lze nastavení měnit operativně bez nutnosti nové kompilace programu

# Konfigurace

- trasování se v konfiguračním souboru nastavuje v tagu `<system.diagnostics>`
- vnitřní tagy pak víceméně odpovídají trasovacím třídám

```
Trace.IndentSize = 2;  
Trace.AutoFlush = true;
```

Nastavení v kódu

```
<configuration>  
  <system.diagnostics>  
    <trace autoflush="true" indentsize="2" />  
  </system.diagnostics>  
</configuration>
```

Stejné nastavení v app.config



# Třídy Trace a Debug

- pro základní sledování stavu programu jsou k dispozici třídy Trace a Debug ze jmenného prostoru `System.Diagnostics`
- tyto třídy jsou téměř totožné a sdílí společné nastavení
  - v `app.config`: tag `<trace>`
- zapisují na trasovací výstup
- volání jejich metod je ovlivněno nastavením podmíněných symbolů TRACE nebo DEBUG v C# kompilery
  - ve Visual Studiu: menu Project/Properties, karta Build
  - příkazem preprocesoru: `#define DEBUG`
  - parameterem příkazové řádky: `/d:TRACE`

# Třídy Trace a Debug

- jestliže je symbol podmíněné kompilace definován, jsou metody dané třídy vykonány, jinak jsou jejich volání při kompilaci vynechány
- ve výchozím nastavení Visual Studia je symbol DEBUG definován v Debug sestavení a TRACE v Debug i Release
- ve skutečnosti jsou metody těchto tříd označeny pomocí atributu [Conditional("symbol")], který kompilátoru říká, že má danou metodu ignorovat pokud uvedený symbol podmíněné kompilace není použit

# Metody tříd Trace a Debug

- metody `Write` pro zápis informací bez přerušení běhu programu
  - nepodmíněné `Write/WriteLine` i podmíněné `WriteIf/WriteLineIf`
  - jen zapíše zprávu
- `Assert`
  - přeruší provádění programu a vypíše informace jestliže není splněna nějaká podmínka
  - ve výchozím nastavení zobrazí dialogové okno typicky se používá pro odhalení chyb během vývoje
  - musí být přítomen `DefaultTraceListener`
- `Fail`
  - nepodmíněný `Assert` – selže vždy

# Trace Listeners

- sledují trasovací výstup a zajišťují jeho formátování a zapsání jinam
  - soubor, email, protokol událostí
- je možné ještě dále omezovat které zprávy zapíše pomocí objektu ve vlastnosti `Filter`
- posluchačů pro trasovací výstup může být více
- třídy `Trace` a `Debug` sdílí jednu kolekci posluchačů dostupnou přes jejich vlastnosti `Listeners`
  - ta ve výchozím nastavení obsahuje jen třídu `DefaultTraceListener`

# Trace Listeners

DefaultTraceListener	posílá zprávy pomocí Windows API debuggeru (VS zobrazuje tyto zprávy v okně Output)
EventLogTraceListener	zapisuje do deníku událostí Windows
ConsoleTraceListener	zapisuje do konzole
TextWriterTraceListener	zapisuje do textového proudu
DelimitedListTraceListener	zapisuje do data oddělená nějakým znakem do souboru
XmlWriterTraceListener	zapisuje do XML souboru

# Trace Switches

- přepínače řídící trasování
- používají se s podmíněnými trasovacími metodami
  - `WriteIf`, `WriteLineIf`
- třída `BooleanSwitch`
  - `zapnuto` / `vypnuto`
- třída `TraceSwitch`
  - 5 úrovní
  - `Off`, `Error`, `Warning`, `Info`, `Verbose`
- lze je nastavit programově i v konfiguračním souboru

# Trace Source

- od verze .NET Frameworku 2.0
- kombinuje Trace Listeners a Trace Switches
- nahrazuje statické třídy Debug a Trace instancemi třídy TraceSource
- dovoluje mít více instancí s různým nastavením
- podrobnější a jemnější rozdělení sledovaných informací než Debug a Trace

# Další knihovny

- Apache Log4net
  - port Log4J pro .NET
  - <http://logging.apache.org/log4net/>
- NLog
  - asynchronní logování, rozložení zátěže
  - <http://nlog-project.org/>
- ELMAH (Error Logging Modules and Handlers)
  - logování chyb pro ASP.NET, řeší prohlížení výjimek, filtrování
  - snadné nasazení
  - podporuje RSS, Twitter, XML, export do Excelu,...
  - <http://code.google.com/p/elmah/>



9.4.2011

Error log for / on RAINBOW (Page #1) - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://www.example.com/elmah.axd

Most Visited Getting Started Latest Headlines

## Error Log for / on RAINBOW

RSS FEED RSS DIGEST DOWNLOAD LOG HELP ABOUT

Errors 1 to 5 of total 5 (page 1 of 1). Start with [10](#), [15](#), [20](#), [25](#), [30](#), [50](#) or [100](#) errors per page.

Host	Code	Type	Error	User	Date	Time
RAINBOW	404	<b>Http</b>	File does not exist. <a href="#">Details...</a>	johndoe	5/13/2009	11:17 AM
RAINBOW	500	<b>Format</b>	Input string was not in a correct format. <a href="#">Details...</a>	johndoe	5/13/2009	11:16 AM
RAINBOW	404	<b>Http</b>	The file '/foobar.aspx' does not exist. <a href="#">Details...</a>	johndoe	5/13/2009	11:16 AM
RAINBOW	500	<b>Application</b>	Error in the application. <a href="#">Details...</a>	johndoe	5/13/2009	11:16 AM
RAINBOW	0	<b>Test</b>	This is a test exception that can be safely ignored. <a href="#">Details...</a>	johndoe	5/13/2009	11:16 AM

Powered by [ELMAH](#), version 1.0.11211.0. Copyright (c) 2004-9, Atif Aziz. All rights reserved. Licensed under [Apache License, Version 2.0](#). Server date is Wednesday, 13 May 2009. Server time is 11:30:15. All dates and times displayed are in the W. Europe Daylight Time zone. This log is provided by the SQLite Error Log.

Done

YSlow 0.029s

# Code Contracts

Design-by-Contract

# Design-by-Contract

- z programovacího jazyk Eiffel (1986)
- formální a ověřitelný popis rozhraní
- zaznamenání informací o
  - vstupních podmínkách,
  - výstupních podmínkách,
  - invariantech,
  - chybách a výjimkách

# K čemu je to dobré

- statická analýza (před a při kompilaci)
  - zjištění dodržení kontraktu a platnosti podmínek bez spuštění programu (kontrola na null, nebo rozsah pole)
- kontrola za běhu aplikace
- generování dokumentace
- generování testů

# Code Contracts

- integrováno do BCL v .NET Framework 4
- knihovna Microsoft.Contracts.dll pro starší verze
- jmenný prostor `System.Diagnostics.Contracts`
- je potřeba mít definován symbol podmíněného překladač `CONTRACTS_FULL`
- pro úplnou funkčnost je potřeba mít Code Contract Tools

# Code Contract Tools

- rozšíření Visual Studia
- dostupné z MSDN DevLabs
- přidává do vlastností projektu dialog pro nastavení
- analyzuje správnost kontraktů
- dokáže změnit zkompilovaný kód programu
  - pracuje na úrovni IL
  - vkládá další kontroly
  - dokáže využít informace o kontraktech v IntelliSense

# Jaké kontrakty lze vytvořit

- vstupní podmínky (requires)
  - co musí být splněno při vstupu do metody
- výstupní podmínky (ensures)
  - co musí být splněno při ukončení metody
- invarianty
  - co objekt musí splňovat po skončení (každé) metody
- tvrzení (asserts)
  - co musí být platné v daném okamžiku
- předpoklady (assumption)
  - co není ověřitelné statickou kontrolou

# Co lze použít v kontraktu

- pravdivostní výrazy (boolean)
  - `i s` voláním metod (bez vedlejších efektů)
- kvantifikátory
  - `Contract.ForAll`
  - `Contract.Exists`
- `Contract.Result`
  - odkazuje na návratovou hodnotu metody
- `Contract.OldValue`
  - odkazuje na hodnoty platné při vstupu do metody
- `Contract.ValueAtReturn`
  - odkazuje na hodnotu výstupního parametru při návratu z metody



# Odkazy

- Code Contracts Project (home page)
  - <http://research.microsoft.com/en-us/projects/contracts/>
- MSDN DevLabs (nástroje)
  - <http://msdn.microsoft.com/en-us/devlabs/dd491992>
- Code Contract na MSDN Library
  - <http://msdn.microsoft.com/en-us/library/dd264808.aspx>