Výjimky a aserce, debugování a zpracování chyb

Error

V programování existují tři typy chyb, pokud pomineme compilation/runtime klafikaci:

- Usage Error
- Programme Error
- System Failures

Usage Error

Tento typ chyb se vyskytuje především u vstupu dat a nastává, když je například argument metody *null*, nebo pokud je hodnota dělitele u matematické operace 0. Pro tento typ chyb, používání exceptions je značně nevhodné kvůli jejich zbytečnosti.

Problem

```
int variable0 = Whatever.GetProperty();
int variable1 = Whatever.GetOtherProperty();

//variable1 may be zero, that would result in DivideByZeroException()
int quotient = variable0 / variable1;
```

Bad Code

```
int variable0 = Whatever.GetProperty();
int variable1 = Whatever.GetOtherProperty();

try
{
int quotient = variable0 / variable1;
}
catch(Exception)
{
Console.WriteLine("Cannot divide by zero, lol");
}
```

Good Code

```
int variable0 = Whatever.GetProperty();
int variable1 = Whatever.GetOtherProperty();

if (variable1 != 0) int quotient = variable0 / variable1;
```

Programme Error

'A program error is a runtime error that cannot necessarily be avoided by writing bug-free code.'

Nejběžnějším příkladem je uživatelský vstup. Bez ohledu na to, jak dobrý je náš program, pokud uživatel zareaguje na "Input any positive number below:", tím že do konzole zadá "Ale ovšem, lásko", máme problém.

Stejně jako u předchozího typu chyb, používání Exceptions je špatný způsob, jak se s tímto vypořádat a obecně existují dva způsoby, jak toto řešit: využitím objektů typu "Type" and keywordu "typeof", nebo přes takzvané "bezpečné metody".

Problem

```
string input = Console.ReadLine();
int variable = int.Parse(input);
```

Bad Code

```
string input = Console.ReadLine();
            int variable;
            try
            {
                variable = int.Parse(input);
            }
            catch (Exception)
            {
                Console.WriteLine("I wanted a number...");
            }
```

Good Code

```
 string input = Console.ReadLine();
            int variable;

            if (!int.TryParse(input, out variable))
                Console.WriteLine("Still not a number...");
```

Dalším problémem podobného typu se vyskytuje u používání objektů typu File. Stejně jako u příkladu výše je doporučenou metodou nejdříve zkontrolovat manuálně, zdali soubor existuje přes File.Exists, než se spoléhat na try-catch bloky.

Try-catch přesto ale své uplatnění u souborů najde. Je doporučenou strategií od Microsoftu zkontrolovat existenci souboru manuálně, ale stejně kód zabalit do struktury chytající výjimky pro případ, že se soubor zkoruptuje během běhu programu nebo pro případ, že soubor přestane existovat. Pointou je eliminovat logickou závislost na try-catch blocích, ne je však úplně odstranit! Důvodem je zpomalení běhu programu, které nastává při chytání výjimky.

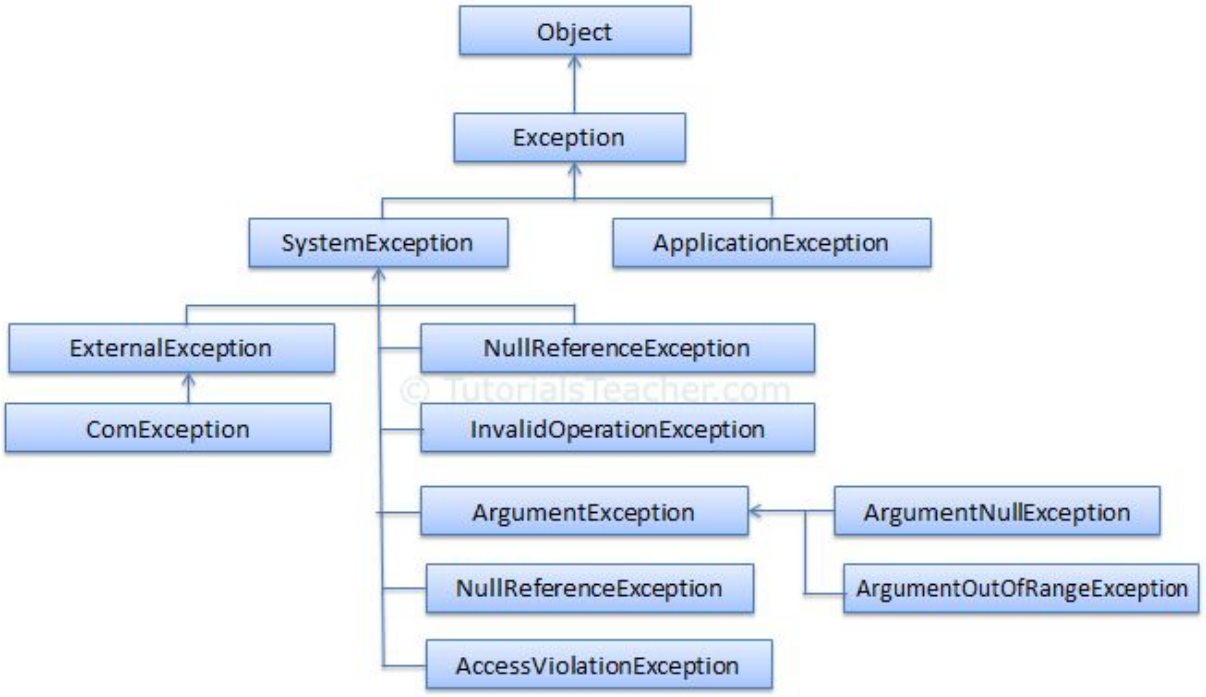
System Failure

Tento typ chyb nelze řešit programově. Jedná se především o chyby způsobené například nedostatkem fyzické paměti nebo poškození hardwaru během běhu programu.

Exception

Objekt[1] typu "Exception" se v C# nachází v namespacu System, společně s objekty například typu enum, Object, Random, String, a tak dále.

Základní hierarchie:



Properties

Každý objekt typu Exception obsahuje následující vlastnosti:

- StackTrace - Property typu string, která ukládá místo na stacku, kde byla vyjímka vyvolána, respektive, kde se nachází keyword "throw", který ji vyvolal
- InnerException - Readonly property typu Exception, která má představovat upřesnění jakého typu "vnější" Exception je. Toto se používá pro kombinované objekty typu Exception. Jedná se především o nástroj pro vytváření vlastních výjimek.
- Message - String property, která obsahuje user-friendly interpretaci výjimky.
- HelpLink - String property pro uložení URL linku pro uživatele v případě vyvolání výjimky. Může například obsahovat link na Microsoft support nebo indická YouTube fix videa.
- HResult - Property typu int, představuje numerickou reprezentaci výjimky.
- Source - String property, která nese jméno objektu nebo aplikace, která výjimku vyvolala.
- TargetSite - ???
- Data - Property typu Dictionary, která obsahuje extra informace pro programátora, obzvláště užitečné pro vlastní typy výjimek.

[1] Exception je skutečně oddědění od prapředka Object

Dopady na běh programu

Vyhazování výjimek (ne vsoukání se do try-bloků) vyžaduje poměrně dost času a prostředků a programátor by se vždy měl snažit minimalizovat počet vyhozených výjimek na minimum.
Doporučená strategie je používat Exceptiony pouze na výjimečné situace (hádám, že od toho to je "výjimka" (Anglicky "Exceptional" znamená výjimečný), nikdy ne pro předvídatelné stavy nebo pro flow-control! Dále se doporučuje nahrazovat vyhazování výjimek například z metod něčím, čemu se říká "return-codes", t.j. raději returnovat -1 v metodě pro počítání absolutních hodnot nežli vyhazovat výjimku. Nezapomeňte, že v Javě se musí udávat do hlavičky metody, jestli vyhazuje výjimku či nikoliv.

Vytváření vlastních Exceptionů

1) Definuj třídu, která má prapředka třídu Exception a definuj veškeré potřebné properties pro uchování extra informací. Třída ArgumentException například obsahuje property uchovávající jméno parametru, který výjimku způsobil.
2) Přepiš existující Metody a properties dle potřeby. S tímto by neměl být problém, jelikož převážná většina metod a properties Exceptionů jsou virtual.
3) Rozhodni, zdali tvoje třída implementuje rozhraní ISerializable.
4) Rozdni, které typy konstruktorů mohou definout instance tvé Exceptiony. Běžnými příklady jsou:
    a) Exception()
    b) Exception(string) - Modifikace property Message
    c) Exceptionh(string, Exception) - Modifikace property Message a InnerException

Assert Class

Assert class je definována jako "public sealed[2] class" a má jednu jedinou property a to je singleton static instance této třídy zvaná "that". Jméno bylo zvoleno tak, aby nested metody třídy Assert dávaly jazykově smysl[3] (podobně jako v Linq nebo SQL).

Třída Assert má využití hlavně v debugu a v testování obecně, především v Unit Testech.

Unit Testing s Asserty

Unit testing se používá jako technika pokročilého odchytávání chyb. Tento typ testování se nazývá "unit" protože se funkcionalita programu rozbije na jednotlivé testovatelné oddíly (units), které se ladí odděleně. Tímto se (narozdíl od testování v celém kontextu programu) usnadňuje testování práce s různými a nekompatibilními typy dat.[4]

[2] "Sealed" znamená, že se od ní nedá dědit.
[3] Například "Assert.that.IsEqual(variable0, variable1)".
[4] Visual Studio Enterprise automaticky Unit Testuje Váš program v real-timu. Nebo alespoň to tak tvrdí Microsoft.

Přidání Unit Testů do existujícího solutionu

1) V Solution Exploreru vyber "Add" a následně "New Project"
2) Vyber framework programu, který testuješ
3) Přidej referenci (Add Reference) do programu, který testuješ.
    a) Vyber projekt v Solution Exploreru
    b) Vyber možnost "Add Reference"
    c) V Reference Manageru, vyber jméno projektu a potvrď

Samotné Testování

Testování je rozděleno do tří fází:

1) Arrange[5]
2) Act[6]
3) Assert[7]

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);

    // act
    account.Withdraw(withdrawal);

    // assert
    Assert.AreEqual(expected, account.Balance);
}
```

Mezi nejužitečnější Assert metody patří:

1) AreEqual(Object, Object) - Různé datové typy automaticky vyhazují false
2) AreNotEqual(Object, Object)
3) IsInstanceOfType(Object, Type) - Užitečné pro například factory metody
4) IsNotInstanceOfType(Object, Type)
5) IsNull(Object)
6) IsNotNull(Object)
7) IsFalse(bool)
8) IsTrue(bool)

---

[5] Aka definuj data se kterými budeš testovat
[6] Aka použij testovanou metodu
[7] Aka použij metodu třídy Assert pro testování, například "IsEqual"
Mimochodem, všiml si někdo, že jsou tyto tři fáze pojmenovány tak, aby začínaly na stejné písmeno? V literatuře se to označuje aliterace.