
04 – Ausnahmebehandlung

Agenda

- Ausnahmebehandlung
- Ausnahmebehandlungsstrategien

Ausnahmebehandlung: Geschichte (1/2)

- Konventionell werden Fehlercodes als Rückgabewert zurückgegeben (siehe auch TryParse)

```
bool CallFunction() {  
    File file = new File("test.txt");  
    if (file == null) return false;  
    // Ausgabe in File hier  
    return true;  
}  
  
bool success = CallFunction();  
if (!success) {  
    // Fehlerbehandlung  
}
```

- Nachteil: es kann keine Funktion mit Rückgabewert geschrieben werden

Ausnahmebehandlung: Geschichte (2/2)

- Fehlercodes, welche als Rückgabewert zurückgegeben werden, müssen nicht konsumiert werden!

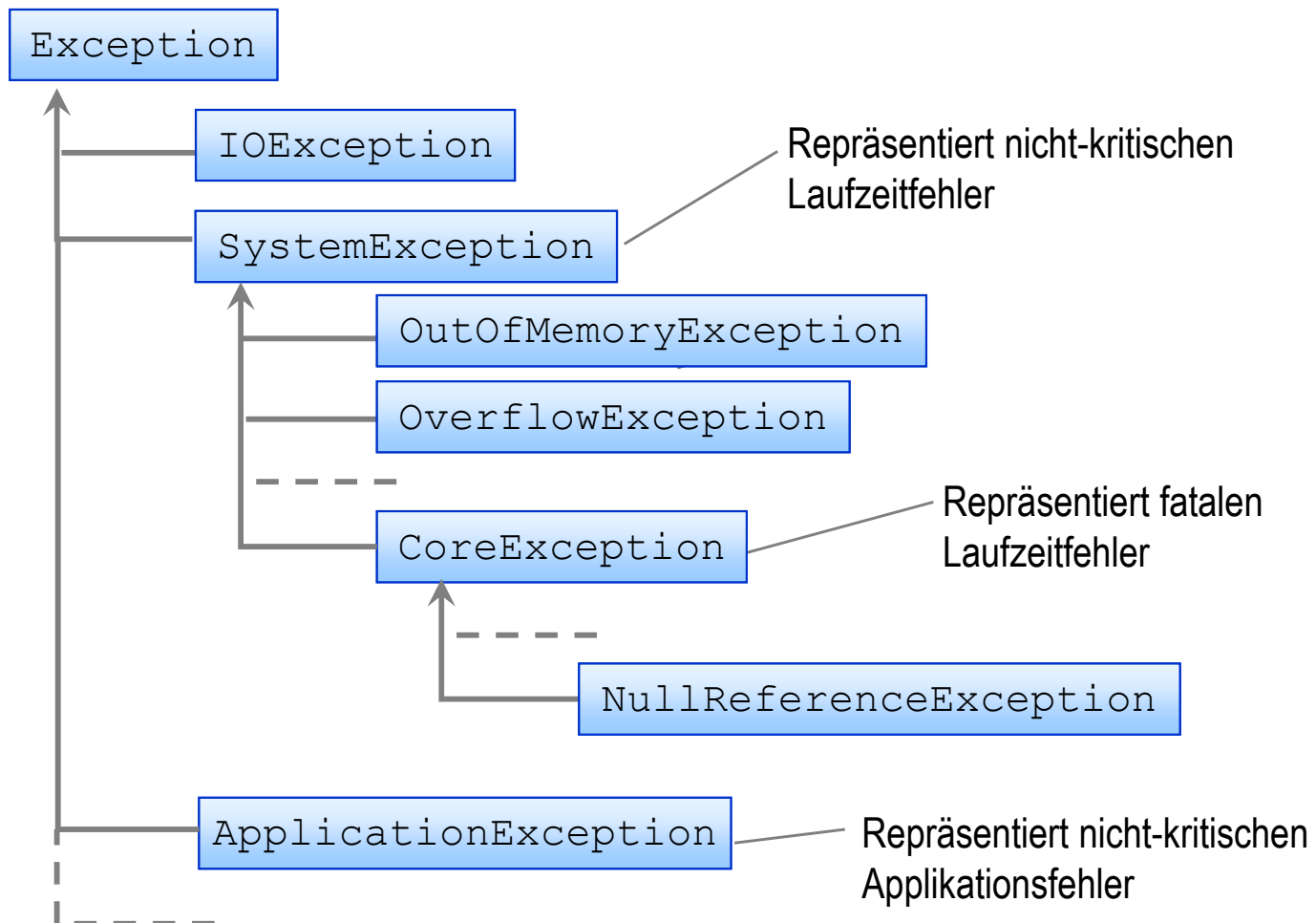
```
bool success = CallFunction();  
if (!success)  
{  
    // Fehlerbehandlung  
}
```

```
CallFunction();
```

Einführung

- Exceptions bieten elegante Fehlerbehandlung in OO-Sprachen
- Fehler müssen verarbeitet werden
- Exception-Klassen beinhalten Informationen zum aufgetretenen Fehler
- Hierarchische Exception-Klassen bieten zusätzlichen Informationsgehalt
- Fehler können an der Stelle bearbeitet werden, wo es sinnvoll ist
- Alle Exceptions leiten von der gemeinsamen Basisklasse `Exception` ab

.NET-Exception-Klassen-Struktur



Try- und Catch-Blöcke

```
try {  
    FileInfo source = new FileInfo("code.cs");  
    int length = (int)source.Length;  
    char[ ] contents = new char[length];  
    ...  
}  
catch (SecurityException e) { ... }  
catch (IOException e) { ... }  
catch (OutOfMemoryException e) { ... }  
catch { ... }
```

Exception werfen mit throw

```
if (minute < 1 || minute > 59) {  
    throw new InvalidTimeException(minute +  
                                    "is not a valid minute");  
    // Not reached!  
}
```


Der finally-Block

- Die Ausdrücke in einem finally-Block werden auf jeden Fall ausgeführt

```
CriticalSection.Enter(x);  
try {  
    ...  
}  
finally {  
    CriticalSection.Exit(x);  
}
```

Ausnahmebehandlung

catch und finally

- Können kombiniert werden
- Im Ausnahmefall werden beide Blöcke durchlaufen

```
CriticalSection.Enter(x);  
try  
{  
    ...  
}  
catch (OutOfMemoryException e)  
{  
    ...  
}  
finally  
{  
    CriticalSection.Exit(x);  
}
```

Exception im Eigenbau

- Exception-Klassen können selber geschrieben werden
- Sie sollten von der Klasse `Exception` ableiten; ursprünglich war `ApplicationException` angedacht. In der Praxis hat sie keine Vorteile gezeigt.

```
class MeineException : Exception
{
    public MeineException() {}
    public MeineException(string msg) : base(msg) {}
    public MeineException(string msg,
                           Exception inner) : base(msg, inner) {}
}
```

Ausnahmebehandlung

Vorteile

- Fehlercodes müssen nicht als Rückgabewert zurückgegeben werden
- Fehler können nicht einfach “vergessen” werden
- Strukturierte Fehlerbehandlung

Nachteile

- Overhead wird generiert
 - Schlechtere Performanz
 - Mehr Code, grösserer Speicherverbrauch

Arithmetischer Überlauf

- Standardmässig wird nicht auf arithmetische Überläufe geprüft
- Mit einem `checked`-Ausdruck kann die Überprüfung eingeschaltet werden

```
checked {  
    int number = int.MaxValue;  
    Console.WriteLine(++number);  
}
```

OverflowException

Ein Exception-Objekt wird
geworfen. WriteLine wird
nicht ausgeführt.

```
unchecked {  
    int number = int.MaxValue;  
    Console.WriteLine(++number);  
}
```

MaxValue + 1 wird
negative?

02147483648

Der `using`-Block

- Für lokal verwendete Ressourcen
- `using`-Block um die verwendete Ressource herum
- Ressource wird beim Verlassen des Blocks automatisch freigeben
- `using` wird in ein `try-finally` umgewandelt

```
using (File myFile = new File("C:\\test.txt"))  
{  
    myFile.Write("Test");  
}
```

Ausnahmebehandlung

Beispiel für `using`-Block:

- Limitiertes Benutzen einer Datenbankverbindung

```
using (SqlConnection con = new SqlConnection(ConnectionString))
{
    command.Connection = con;
    con.Open();
    SubCategoryName = command.ExecuteScalar().ToString();
    // con.Close(); Diese Zeile wird durch "using" überflüssig
}
```

rufsbildung

Strategien

■ Caller-Beware:

- Exception nicht fangen (d.h. leeren catch-Block programmieren)
- Keine Information für den Aufrufer
- Objekt bleibt in einem „ungültigen“ Zustand

■ Caller-Confuse:

- Exception fangen
- Aufräumen → Objekt in einem „aufgeräumten“ Zustand
- Gleiche Exception weiter werfen

■ Caller-Inform:

- Exception fangen
- Aufräumen → Objekt in einem „aufgeräumten“ Zustand
- Neue Exception werfen, welche die Original-Exception plus Zusatzinfos zum Kontext enthält

Ausnahmebehandlungsstrategien

Beispiel Strategie Caller-Confuse:

```
public static void CallerConfuse()
{
    ExceptHandlingCallerConfuse callerConfuse = new ExceptHandlingCallerConfuse();
    try
    {
        double resultat = callerConfuse.DoAverage();
        Console.WriteLine("Mittelwert = {0}", resultat);
    }
    catch (Exception e)
    {
        Console.WriteLine();
        Console.WriteLine("Test Caller Confuse:");
        Console.WriteLine("Exception {0}", e);
    }
}
```

```
public class ExceptHandlingCallerConfuse
{
    public double DoAverage()
    {
        int summe = 0;
        int count = 0;
        double mittelWert;
        try
        {
            mittelWert = summe / count;
            return mittelWert;
        }
        catch (DivideByZeroException e)
        {
            // Hier wird bereinigt
            // und anschl. die gleiche Exception
            // weiter geworfen
            throw e;
        }
    }
}
```


«throw e» setzt den Stacktrace zurück, was schlecht ist;
es sieht dann so aus, als käme der Fehler von hier.

«throw» ohne «e» ist besser, weil der ganze Stacktrace mitgeschickt wird.

Ausnahmebehandlungsstrategien

Beispiel Strategie Caller-Inform:

```
public static void CallerInform()
{
    ExceptHandlingCallerInform callerInform = new ExceptHandlingCallerInform();
    try
    {
        double resultat = callerInform.DoAverage();
        Console.WriteLine("Mittelwert = {0}", resultat);
    }
    catch (Exception e)
    {
        Console.WriteLine();
        Console.WriteLine("Test Caller Inform:");
        Console.WriteLine("Exception {0}", e);
    }
}
```



```
public class ExceptHandlingCallerInform
{
    public double DoAverage()
    {
        int summe = 0;
        int count = 0;
        double mittelWert;
        try
        {
            mittelWert = summe / count;
            return mittelWert;
        }
        catch (DivideByZeroException e)
        {
            // Hier wird bereinigt
            // und anschl. eine neue Exception geworfen
            // mit Informationen zum Kontext und der ursprünglichen Exception
            throw new DivideByZeroException(
                "Division durch 0 in DoAgerage(): (count = 0) " , e);
        }
    }
}
```

Übung 4.1



Exceptions

- 1) Implementiere eine Klasse `Person` mit folgenden Attributen:
 - Name
 - Vorname
 - Stadt
- 2) Ergänze die Klasse `Person` um die Methode `CompareTo(Object p)`, in welcher eine Exception geworfen wird, falls das Argument `p` nicht vom Typ `Person` ist.
- 3) Schreibe ein Programm, um die Methode `CompareTo()` zu testen.