**Exceptions und Exceptionhandling – Ausnahmen und Ausnahmebehandlung**

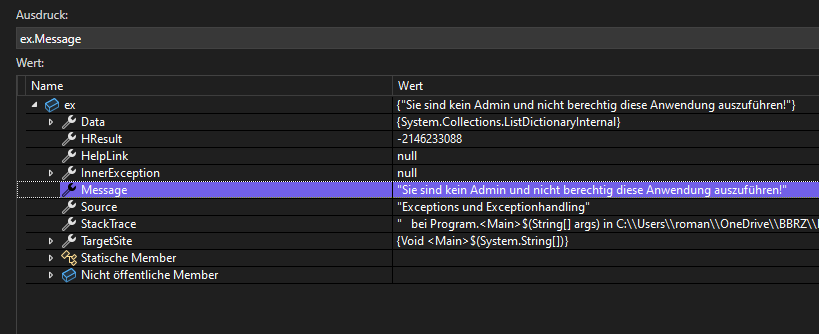
**Einführung**

**Definition**: Was sind **Exceptions (Ausnahmen)?**   
Eine Exception ist eine **Laufzeitabweichung**, die die normale Programmausführung **unterbricht**.

**Was kann eine Exception hervorrufen?** Ganz viel. Z.B.:

* + Division durch Null (**DivideByZeroException**)
  + Zugriff auf nicht vorhandene Array-Indizes (**IndexOutOfRangeException**)
  + Umgang mit null-Werten (**NullReferenceException**)
  + U.V.M.

Was beinhaltet ein Exception-Objekt?



**Behandlung von Exceptions**

**try**

{

// Code, der eine Ausnahme **auslösen könnte**

}

**catch** (Exception ex)

{

// Code zur **Fehlerbehandlung**

}

**finally**

{

// Code, der **immer** ausgeführt wird (auch bei Exceptions)

}

**Erklärung in einfachen Try-Catch-Finally-Block**

* + **try**: Hier befindet sich der "**riskante**" Code.
  + **catch**: Hier wird die Ausnahme abgefangen und ggf. behandelt. Objekt Exception ex enthält Informationen zur Exception. Diese könnten von da aus könnten **geloggt** werden. Es auch der Platz für die Aktivierung entsprechender Alarme an den System- oder Applikation-Administrator (Alerting).
  + **finally**: wird immer ausgeführt, unabhängig davon, ob eine Ausnahme auftritt oder nicht (z. B. um Ressourcen freizugeben). Es ist optional, d.h. man kann es im Gegensatz zu try und catch weg lassen.

**Mehrere catch-Blöcke mit Unterscheidung von Exception-Typen**

**try**

{

// Riskanter Code

}

**catch** (**DivideByZeroException** ex)

{

Console.WriteLine("Division durch Null!");

}

**catch** (**IndexOutOfRangeException** ex)

{

Console.WriteLine("Array-Index außerhalb des Bereichs!");

}

**catch** (**Exception** ex)

{

Console.WriteLine("Ein anderer Fehler ist aufgetreten: " + ex.Message);

}

**Mit throw Exception neu auslösen**

Eine Exception kann (z.B. nach deren Behandlung) auch **neu geworfen** und so **weitergereicht** werden können.

Diese Vorgangsweise ist besonders wichtig in **schichtenorientierten Applikationen**, wo jedes Layer „für sich lebt“, seine Exceptions selbst behandelt **aber auch** solche mittels **throw** neu wirft, damit **der Klient der jeweiligen Schicht** (die aufrufende Unterkomponente), die Chance hat, auf die Exception in **eigener Art und Weise** zu reagieren.

static void MethodeA()

{

MethodeB();

}

static void MethodeB()

{

try

{

MethodeC();

}

catch (Exception ex)

{

Console.WriteLine("Fehler in MethodeB: " + ex.Message);

**throw**; // Exception weitergeben

}

}

static void MethodeC()

{

throw new InvalidOperationException("Fehler in MethodeC");

}

**Design by Contract (DbC)**

Ein Programmieransatz, bei dem **Voraussetzungen, Nachbedingungen und Invarianten** explizit formuliert werden. Wenn eine dieser Bedingungen verletzt wird, wird **gezielt eine Exception geworfen**.

void ZieheGeld(decimal betrag)

{

if (betrag <= 0)

throw new ArgumentException("Betrag muss positiv sein.");

if (betrag > kontostand)

throw new InvalidOperationException("Nicht genug Geld am Konto.");

kontostand -= betrag;

}

Der Entwickler geht davon aus, dass die Methode nur mit korrekten Werten aufgerufen wird – andernfalls ist das ein **Vertragsbruch (contract violation).**

**Fail Fast Principle**

**Definition:**

Ein Prinzip, bei dem Fehler **so früh wie möglich** erkannt und **sofort durch eine Exception gemeldet** werden – anstatt später schwer auffindbare Folgefehler zu riskieren.

**„Wenn etwas nicht stimmen kann, brich sofort ab.“**

**Beispiel:**

public void SetAlter(int alter)

{

if (alter < 0 || alter > 130)

throw new ArgumentOutOfRangeException(nameof(alter));

this.alter = alter;

}

Statt stillschweigend z. B. alter = 0 zu setzen, wird **sofort ein Fehler geworfen**, um das Problem direkt sichtbar zu machen.

**Defensives Programmieren**

**Definition:**

Ein allgemeiner Programmierstil, bei dem der Code **nicht blind vertraut**, sondern alle Eingaben und Zustände überprüft – und bei Widerspruch ggf. **Exceptions** wirft.

Ziel: Robuster und weniger anfällig für unvorhergesehene Fehler.

Wenn du bei **Logikfehlern bewusst Exceptions wirfst**, folgst du in der Regel einem oder mehreren dieser Prinzipien:

|  |  |  |
| --- | --- | --- |
| Name | Ziel | Exception bei |
| Design by Contract | Vertragsverletzungen sichtbar machen | Verstoß gegen Vor-/Nachbedingungen |
| Fail Fast | Fehler sofort aufdecken | Ungültiger Zustand oder Eingabe |
| Defensives Programmieren | Systemrobustheit erhöhen | Unerwartete Eingaben oder Zustände |

**Eigene Exception-Klassen für den Bedarf einer benutzerdefinierten Ausnahme erstellen**

Mit benutzerdefinierten Exceptions können Fehler klar und aussagekräftig benannt werden, was die **Lesbarkeit des Codes** verbessert. Sie ermöglichen es, unterschiedliche Fehlerarten **präzise zu unterscheiden** und entsprechend zu behandeln.  
Die Klasse einer benutzerdefinierten Exception muss **von** Exception **erben**.

**try**

{

Console.WriteLine("Wie heißt Ihre Systemrolle?");

var antwort = Console.ReadLine();

if (antwort != "Admin")

throw new WrongRuleFrirstNameException();

Console.WriteLine("Willkomen " + antwort + "!");

}

**catch** (**WrongRuleFrirstNameException** ex)

{

Console.WriteLine(ex.Message);

}

public class **WrongRuleException** : Exception

{

public WrongRuleException(): base("Sie sind kein Admin und nicht berechtig diese Anwendung auszuführen!")

{

}

}

**Hinweis:**

Mit Exceptions sollte man sparsam umgehen, weil deren Erzeugung recht viel Ressourcen (RAM, CPU) auf jener Maschine, auf der sie geworfen werden, erzeugt.

Innerhalb einer langen Iteration (Schleife) sollte man gar intelligente Logik einbauen, damit dieselbe Exception nicht bei jedem fehlerhaften Durchlauf auftritt bzw. künstlich (per throw) geworfen wird.

**Aufgabe**

Schritt 1.)

Erstellen Sie eine Bankkonto-Klasse mit Methoden zum **Einzahlen** und **Abheben**. Erstellen Sie eine eigene **BankingException**, die auftritt, wenn jemand versucht, mehr Geld abzuheben, als auf dem Konto verfügbar ist.

Schritt 2.)

Erstellen Sie mindestens eine weitere, sinnvolle Exception-Klasse (z.B. Wenn der einzuzahlende Betrag kleiner null ist oder/und wenn man einen Betrag über eine zulässige Höhe von 440 Euro abheben will) und implementieren Sie diese an passenden Stellen.

Schritt 3.)

Der Admin Ihrer Anwendung möchte über die Lauffähigkeit und Abstürze Ihrer Anwendung informiert werden.  
Erstellen Sie eine Logging-Komponente, deren Log-Methoden die Fähigkeit haben, Infos, Warnings und Errors zu loggen.  
Die Log-Methode speichert die Einträge in ein JSON-File, das **minütlich** (nach Bedarf) neu angelegt wird.  
Beachten Sie bei Logs von Type Error auf deren starke Aussagekraft, indem sie neben ErrorMessage auch den StackTrace Ihres Exception-Objekts mitspeichert.

**Schlechter Stil: "Exception-orientierte Programmierung"**

Der Begriff *Exception-orientierte Programmierung* bezeichnet eine **Stilrichtung**, bei der Exceptions nicht nur zur Behandlung **außergewöhnlicher Fehlerfälle**, sondern auch für **gewöhnliche Programmlogik oder Kontrolle** verwendet werden. Das ist **meist ungewollt oder ineffizient**.

**Typische Merkmale:**

* Häufige Verwendung von try/catch an Stellen, wo eine Prüfung (z. B. if, TryParse, Contains, ...) genügt hätte.
* Erwartbare oder normale Fehler (z. B. falsche Benutzereingaben, leere Listen, ungültige Indizes) führen zu Ausnahmen.
* Exceptions werden verwendet wie ein **Ersatz für bedingte Logik**.

**Warum ist das problematisch?**

* **Leistung**: Exception-Handling ist vergleichsweise **langsam**, besonders bei häufigem Auftreten.
* **Lesbarkeit**: Der Code wird **schwerer zu verstehen**, weil Fehlerbehandlung und normale Logik vermischt sind.
* **Design**: Es widerspricht dem Prinzip: *"Handle errors before they happen, not after they blow up."*

**Gute Vorgehensweise**

**Wann soll man try/catch verwenden?**

* Für **wirklich unerwartete** Situationen:

Datei nicht gefunden

Netzwerkfehler

Datenbankverbindungsfehler

Fehler beim Deserialisieren externer Daten

**Wann soll man auf try/catch verzichten?**

* Bei **normalen Kontrollstrukturen** oder erwartbaren Fehlfällen:

Eingabe überprüfen (TryParse)

Array-Index prüfen (if (index >= 0 && index < array.Length))

Liste auf Leerheit prüfen (if (liste.Any()))

Objekt auf null prüfen (if (obj != null))

**Beispiele: Schlecht vs. Gut**

**Beispiel 1: Exception-orientiert**

try

{

int zahl = int.Parse(Console.ReadLine());

Console.WriteLine("Ihre Zahl: " + zahl);

}

catch

{

Console.WriteLine("Ungültige Zahl!");

}

**Besser: Logik-basiert**

string eingabe = Console.ReadLine();

if (int.TryParse(eingabe, out int zahl))

{

Console.WriteLine("Ihre Zahl: " + zahl);

}

else

{

Console.WriteLine("Ungültige Zahl!");

}

**Beispiel 2: Zugriff auf Liste ohne Prüfung**

try

{

var erster = meineListe[0];

Console.WriteLine("Erstes Element: " + erster);

}

catch

{

Console.WriteLine("Liste ist leer.");

}

**Besser: Vorher prüfen**

if (meineListe.Count > 0)

{

Console.WriteLine("Erstes Element: " + meineListe[0]);

}

else

{

Console.WriteLine("Liste ist leer.");

}

**Übung: Exception-orientierten Code verbessern**

**Aufgabe**

Im folgenden Programm sollen die Benutzereingabe geprüft und das erste Element einer Liste ausgegeben werden. Der Code verwendet dabei ausschließlich try/catch. **Verbessern Sie ihn so, dass Ausnahmen vermieden werden.**

try

{

Console.Write("Geben Sie eine Zahl ein: ");

int zahl = int.Parse(Console.ReadLine());

List<string> namen = new List<string>();

Console.WriteLine("Erster Name: " + namen[0]);

}

catch

{

Console.WriteLine("Ein Fehler ist aufgetreten.");

}

|  |  |  |
| --- | --- | --- |
|  | Schlechte Praxis (Exception-orientiert) | Gute Praxis |
| Fehlerkontrolle | try/catch für jede Eingabe | TryParse & Bedingungen |
| Listen-Zugriff | Einfach zugreifen + catch | Erst prüfen, dann zugreifen |
| Lesbarkeit | Schwer verständlich | Klar strukturierter Kontrollfluss |
| Performance | Potenziell langsam | Schnell & vorhersehbar |
| Einsatz von Exceptions | Als Logikersatz | Nur für Ausnahmen |