C#
Exceptions

BITLC | Tom Selig
20. Januar 2026

Grundlagen

- Exceptions (Ausnahmen) treten auf, wenn das Programm zur Laufzeit versucht eine unzulässige Operation durchzuführen.
- Das können sehr unterschiedliche Operationen sein, z. B.:

  Eine benötigte Datei ist nicht vorhanden

  Fehlerhafte Anmeldung an einem Server

  Eine Objektvariable ist nicht initialisiert

  Es wird versucht durch 0 zu teilen

- In diesen Fällen löst die Laufzeitumgebung eine Ausnahme aus und „wirft“ (`throw`) ein `Exception`-Objekt, das an einer anderen Stelle im Code „gefangen“ (`catch`) und behandelt werden kann.

Grundlagen

- Wenn eine Exception geworfen wird, wird zunächst geprüft, ob die Exception innerhalb der betroffenen Methode gefangen wird.
- Ist dies der Fall, kann sie dort behandelt werden und das Problem ist behoben.
- Ist dies nicht der Fall, wird die Exception an die aufrufende Methode weitergegeben und dort erfolgt wieder eine Prüfung.
- Dies geht so weiter, bis die Exception in der Main-Methode angekommen ist (Call-Stack).
- Wird die Exception dort ebenfalls nicht gefangen, wird das Programm abgebrochen und eine Fehlermeldung ausgegeben.

Exceptions fangen

- Exceptions können (und sollten) im Code gefangen und behandelt werden.
- Die geschieht durch einen `try`-`catch`-Block:

```
try {
    // Potenziell "gefährlicher" Code
}
catch {
    // Code zur Behandlung der Exception
}
```

- Code, der potenziell eine Exception auslösen könnte, wird im `try`-Block gekapselt.
- Der folgende `catch`-Block enthält den Code zur Behandlung.

Beispiel

Eingabe von Buchstaben

zahl2 könnte 0 sein

Eingabe größer oder kleiner Wertebereich

```
static void Main(string[] args) {
    Console.Write("Erste Zahl: ");
    int zahl1 = int.Parse(Console.ReadLine());
    Console.Write("Zweite Zahl: ");
    int zahl2 = int.Parse(Console.ReadLine());

    Console.WriteLine(zahl1 / zahl2);
}
```

Potenziell *„gefährlicher“* Code wird gekapselt

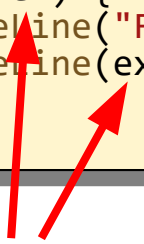
Code zur Behandlung der Exception

```
static void Main(string[] args) {
    try {
        Console.Write("Erste Zahl: ");
        int zahl1 = int.Parse(Console.ReadLine());
        Console.Write("Zweite Zahl: ");
        int zahl2 = int.Parse(Console.ReadLine());

        Console.WriteLine(zahl1 / zahl2);
    }
    catch {
        Console.WriteLine("Es ist ein Fehler aufgetreten!");
    }
}
```

Exceptions auswerten

- Egal was der Benutzer jetzt eingibt, das Programm läuft auf jeden Fall fehlerfrei durch.
- Allerdings ist die Meldung des Programms nicht wirklich aussagekräftig.
- Der Benutzer weiß nicht, was er falsch gemacht hat.
- Und der Code weiß es ebenfalls nicht …?!?

```
static void Main(string[] args) {
    try {
        Console.Write("Erste Zahl: ");
        int zahl1 = int.Parse(
            Console.ReadLine());
        Console.Write("Zweite Zahl: ");
        int zahl2 = int.Parse(
            Console.ReadLine());

        Console.WriteLine(zahl1 / zahl2);
    }
    catch (Exception ex) {
        Console.WriteLine("Fehler:");
        Console.WriteLine(ex.Message);
    }
}
```



Das geworfene Exception-Objekt

Exceptions auswerten

- Jedes `Exception`-Objekt, das von der Laufzeitumgebung im Fehlerfall erzeugt wird, hat eine Reihe von Properties.
- Die wichtigsten davon sind:

  `Message`:
  String mit der Beschreibung der Ausnahme

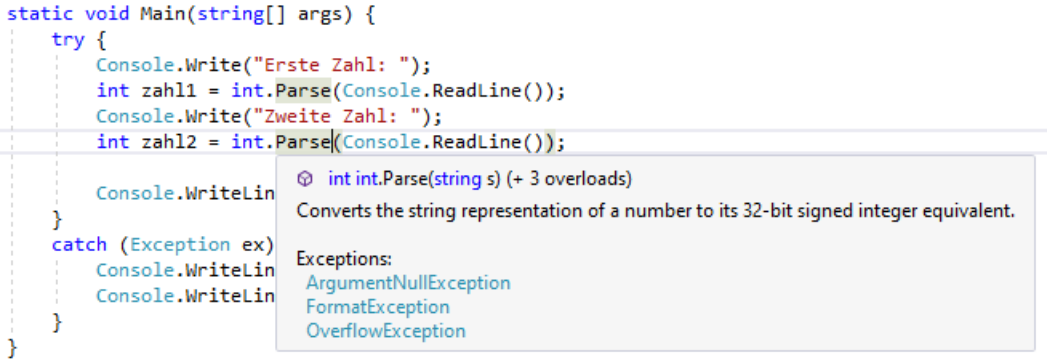
  `StackTrace`:
  String mit der Aufruf-Reihenfolge aller Methoden bis zur Ausnahme

  `InnerException`:
  `Exception`, deren Behandlung die aktuelle `Exception` ausgelöst hat

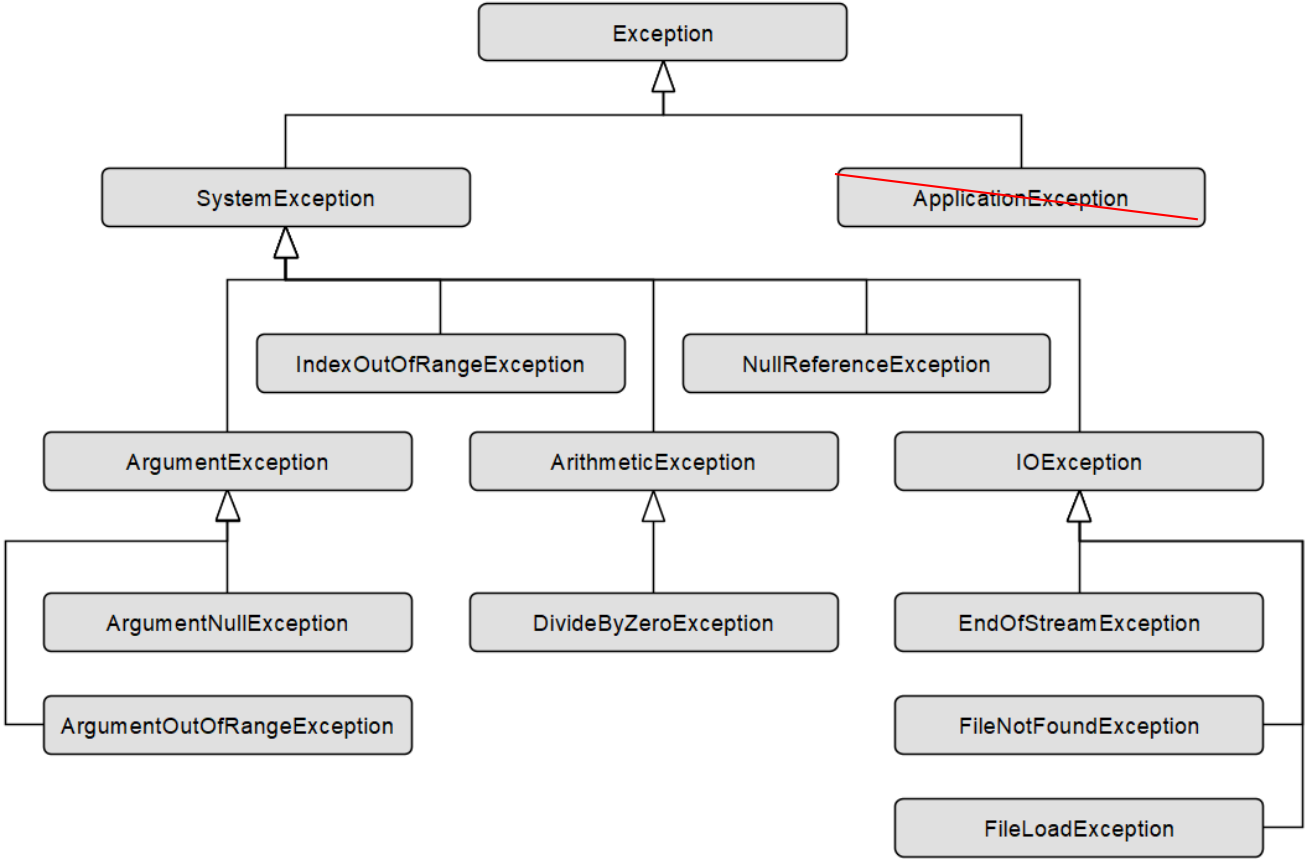
  `Data`:
  Dictionary mit Zusatzinformationen zur Ausnahme

Exception-Klassen

- Das Programm gibt jetzt eine Fehlermeldung aus, die dem Benutzer sagt was er falsch gemacht hat.
- In .NET gibt es verschiedene Arten von `Exception`-Klassen, auf die man individuell reagieren kann.
- Alle `Exception`-Klassen stammen von der Klasse `Exception` ab, daher kann man mit dieser Klasse alle anderen Typen fangen.
- Welche Typen von Exceptions eine Methode werfen kann, ist aus der Dokumentation und der IntelliSense ersichtlich.



Exception-Hierarchie (Auszug)



Exceptions auswerten

- Nach dem `try`-Block können mehrere `catch`-Blöcke folgen.
- Diese können unterschiedliche Typen von `Exception` fangen.
- Es wird aber immer nur der **erste** passende `catch`-Block ausgeführt.
- Die Reihenfolge der Blöcke ist daher wichtig (vgl. `Exception` Hierarchie).

```
static void Main(string[] args) {
    try {
        //
        // Gleicher Code wie bisher
        //
    }
    catch (FormatException ex) {
        // Behandlung von FormatException
    }
    catch (OverflowException ex) {
        // Behandlung von OverflowException
    }
    catch (DivideByZeroException ex) {
        // Behandlung von DivideByZeroExcept.
    }
    catch (Exception ex) {
        // Behandlung anderer Exceptions
        Console.WriteLine("Fehler:");
        Console.WriteLine(ex.Message);
    }
}
```

`finally`-Block

- Manchmal muss noch Code ausgeführt werden, auch wenn ein Fehler aufgetreten ist.
- Z. B. wenn noch eine Datei geschlossen werden muss.
- Dies kann in einem `finally`-Block geschehen.
- Dieser wird immer ausgeführt, egal ob ein Fehler aufgetreten ist oder nicht.

```
static void ReadFile() {
    FileStream fs = File.Open("file.txt",
        FileMode.Open, FileAccess.Read);
    try {
        StreamReader sr = new
            StreamReader(fs);
        int zahl1 = int.Parse(sr.ReadLine());
        int zahl2 = int.Parse(sr.ReadLine());
        Console.WriteLine(zahl1 / zahl2);
    }
    catch (Exception ex) {
        Console.Write("Fehler: ");
        Console.WriteLine(ex.Message);
        Console.ReadLine();
    }
    finally {
        if (fs != null) {
            fs.Dispose();
        }
    }
}
```

Exceptions werfen

- Exceptions können nicht nur von der Laufzeitumgebung geworfen werden.
- Der Anwendungscode kann selbst Exceptions werfen.
- Dies geschieht mit dem Schlüsselwort `throw`.
- Das kann am Beginn einer Methode sinnvoll sein, um die Parameter zu prüfen.

```
static void Main(string[] args) {
    string[] strings = ["abc", "def", "ghi"];
    PrintString(strings, 1);
    PrintString(strings, 7);
    PrintString(null, 2);
}


static void PrintString(string[] arr, int p) {
    if (arr == null) {
        throw new ArgumentNullException(
            nameof(arr));
    }
    if (p < 0 || p > arr.Length -1) {
        throw new ArgumentOutOfRangeException(
            nameof(p),
            p,
            "Ausserhalb der Array-Grenzen");
    }

    Console.WriteLine(arr[p]);
}
```

Exceptions werfen

- Auch innerhalb des `catch`-Blocks kann eine `Exception` geworfen werden.
- Dies kann entweder eine neue `Exception` sein, oder aber die gefangene `Exception` wird weiter geworfen.
- Aber Vorsicht: Es gibt einen Unterschied zwischen `throw` und `throw ex`!

```
static void Main(string[] args) {
    Console.Write("Datei: ");
    string filename = Console.ReadLine();
    try {
        PrintFile(filename);
    }
    catch (Exception ex) {
        Console.WriteLine(ex.Message);
    }
}

static void PrintFile(string filename) {
    StreamReader sr = null;
    try {
        sr = File.OpenText(filename);
        Console.WriteLine(sr.ReadToEnd());
    }
    catch (Exception ex) {
        Logger.Log(ex);
        throw;
    }
}
```

Eigene `Exception`-Klassen

- Sollten die vordefinierten `Exception`-Klassen nicht ausreichen, können eigene `Exception`-Klassen erstellt werden.
- Diese sollten nach Möglichkeit den folgenden Regeln entsprechen:

  Abgeleitet von `Exception` (`ApplicationException` gilt als veraltet)

  Der Bezeichner endet mit `Exception`

  Drei Konstruktoren analog zur Klasse `Exception`
- Darüber hinaus können beliebige weitere Eigenschaften und/oder Methoden codiert werden.
- Für die Speicherung von Zusatzinformationen kann auch die `Data`-Eigenschaft genutzt werden.

Eigene Exception-Klassen

```
class MyException : Exception {

    // Konstruktoren
    public MyException()
    {
    }

    public MyException(string message) : base(message)
    {
    }

    public MyException(string message, Exception inner) : base(message, inner)
    {
    }
}
```

Eigene Exception-Klassen

```
class MyException : Exception {

    // Property, das die Data-Eigenschaft der Basis-Klasse nutzt
    public string? MyInfo {
        get { return Data["MyInfo"] as string; }
        private set { Data["MyInfo"] = value; }
    }

    // Konstruktoren
    public MyException() { }

    public MyException(string message) : base(message) { }

    public MyException(string message, Exception inner) : base(message, inner) { }

    public MyException(string message, string myInfo) : this(message, myInfo, null) { }

    public MyException(string message, string myInfo, Exception inner)
        : base(message, inner) {
        MyInfo = myInfo;
    }
}
```

Schlussbemerkungen

- Wann sollte man Exception-Handling einsetzen, und wann nicht?

  So oft wie nötig, so wenig wie möglich!

- Exceptions sind teuer! Sie kosten Rechenzeit, Speicherplatz und somit auch Performance.
- Viele Situationen lassen sich auch anders lösen.

  Das benötigt meist weniger Rechenleistung und Speicherplatz.

- Exception-Handling sollten niemals ohne guten Grund verwendet werden. Insbesondere nicht, damit unerwartete Fehler geschluckt werden und das Programm nicht abstürzt.