Ausnahmebehandlung – Exception handling

Murphy's Law:

Anything that *can* go wrong *will* go wrong.

Bei vielen Methodenaufrufen ist es realistisch und erforderlich, auf Störungen des normalen Ablaufs vorbereitet zu sein.

Ausnahmebehandlung – Exception handling

- unterschiedliche Funktionen aus unterschiedlichen, getrennt entwickelten Modulen müssen zusammenarbeiten

- in einem Teil einer Anwendung muss auf eine Funktionalität (Funktion) aus einem (Bibliotheks-)Modul zugegriffen werden – wie kann sie mit Laufzeitfehlern dieses Moduls umgehen?

Ausnahmebehandlung – Exception handling

Stellt eine Funktion während ihres Ablaufes eine "unübliche Situation" (Ausnahme, Exception) fest,
kann die Ursache sein:

- Programmierfehler, z. B. versuchter Feldzugriff mit ungültigem Indexwert, Ganzzahldivision durch 0 oder ...
- besondere Umstände: z. B. Speichermangel, unterbrochene Netzwerkverbindung,

so dass die Funktion **nicht mehr in der Lage ist, ordnungsgemäss abzulaufen** und ihre Bestimmung zu erfüllen,
kann sie auf unterschiedliche Art- und Weise reagieren:

- Fehlermeldung ausgeben (wohin?) und das Programm abbrechen

- Einen Fehlerstatus an den Aufrufer als Funktionsergebnis zurückgeben:
  Trennung von Fehlerfeststellung (in der Funktion) und Behandlung des Fehlers (beim Aufrufer)

Ausnahmebehandlung – Exception handling

Probleme der Lösung mit dem Rückgabe-Fehlerwert:

Nur ein Rückgabewert ist möglich: eventuell kann man die Rückgabewerte nicht von den Fehlerwerten trennen

- Diese Art der Fehlerbehandlung ist mühsam, weil bei jedem Aufruf separat abgefragt werden muß, ob er erfolgreich war (Programm wird deutlich länger)
- Vergisst man die Fehlerbehandlung, wird u.U. mit einem falschen Wert weitergerechnet
- Fehlererkennung und Fehlerbehandlung ist nicht völlig getrennt
- Bibliotheksfunktionen sollten bei einem ernsten Fehler dem Aufrufer eine letzte Chance für Aufräumarbeiten geben

Ausnahmebehandlung – Exception handling

Teil a: Fehlerdefinition

Fehlerklasse:
Garfields friends

- An der Unfallstelle wird ein Ausnahmeobjekt der Klasse **Exception** (im Namensraum **System**) oder aus einer **problemspezifischen Unterklasse** erzeugt
- Auf diese Weise können verschiedene Arten von Fehlern unterschieden werden (und man denkt über potentielle Fehler nach und beschreibt/ definiert diese!)
- In einer fehlerbeschreibenden Unterklasse können bei Bedarf weitere Informationen über den Fehler abgelegt werden, z.B. Eingabewerte des fehlerhaften Funktionsaufrufs.

public class COdieExcept: Exception
{

}

public class CJonExcept: Exception
{

}

public class CNermalExcept: Exception
{

}

Ausnahmebehandlung – Exception handling

Die Initiative beim Auslösen einer Exception kann ausgehen …

- von der **CLR**
  Entdeckt die CLR einen Fehler, der nicht zu schwerwiegend ist und vom Anwendungsprogramm prinzipiell behandelt werden kann, dann wirft sie ein Exceptionobjekt, z. B. ein Objekt aus der Klasse **ArithmeticException** bei einer versuchten Ganzzahldivision durch 0.

- vom **Anwendungsprogramm**, wozu auch die verwendeten Bibliotheken gehören (**throw**-Anweisung).

Ausnahmebehandlung – Exception handling

Teil b: Fehlererkennung und -signalisierung

- Ein Teil des Lösungsalgorithmus (inklusive Funktionsaufrufe) wird in geschweifte Klammern eingeschlossen und dieser Block wird mit dem Schlüsselvort „try“ versehen (sog. **try-Block** – kapselt Fehlerentstehung):

```
try
{   /* Loesungsalgorithmus ggf. mit Funktionsaufrufen */
    ...
}
```

- Wird innerhalb dieses geklammerten Teils eine Ausnahmesituation **festgestellt**, so wird mit einer **throw-Anweisung** ein Objekt (Ausdruck: Variable, Konstante eines gewissen Types, Ansnahmeobjekt oder Ausnahme genannt) "geworfen“ (auch aus Bibliothek):

```
if ( sonderfall) throw ausdruck;
```

throw

- Das System bricht hierauf "geordnet" die Abarbeitung des ganzen try-Blockes, der Programmfluss wird unmittelbar **hinter** dem try-Block fortgesetzt
- das mittels throw vom try-Block ausgeworfene Objekt ist noch übrig!

Ausnahmebehandlung – Exception handling

Teil c: Fehlerfangen und -behandlung

Reaktionsmöglichkeiten:

- Exception abfangen und das Problem behandeln. (Es fliegen natürlich keine Objekte durch die Gegend, die mit irgendwelchen Gerätschaften eingefangen werden.)
  Die Laufzeitumgebung prüft, ob die betroffene Methode geeigneten Code zur Behandlung einer Exception enthält (einen sogenannten *Exception-Handler*).
  Wenn ja: Exception-Handler wird ausgeführt, erhält als Aktualparameter die Exception mit Informationen über das Problem.
  Nach der Ausnahmebehandlung kann die Methode ...
    - o ihre Tätigkeit angepasst fortsetzen
    - o oder ihrerseits ein Exception-Objekt werfen (entweder das ursprüngliche oder ein informativeres, neues) und damit die Kontrolle an den Aufrufer zurückgeben.

- oder das Ausnahmeobjekt ignorieren.
  In diesem Fall besitzt eine Methode keinen zum Ausnahmeobjekt passenden Exception-Handler. Die Methode wird beendet, und das Ausnahmeobjekt wird dem Vorgänger in der Aufrufersequenz überlassen.

Ausnahmebehandlung – Exception handling

- Direkt hinter dem try-Block steht (mindestens) eine **catch**-Anweisung (im Allgemeinen aber mehrere):

```
try
{  ...
}
catch ( <Garfields friends>)
{
    switch (…)
   { case…:
     case…:
   }
}
catch ( typ2 name)
{  ...
}
...
```




Exception handling: unbehandelte Ausnahmen

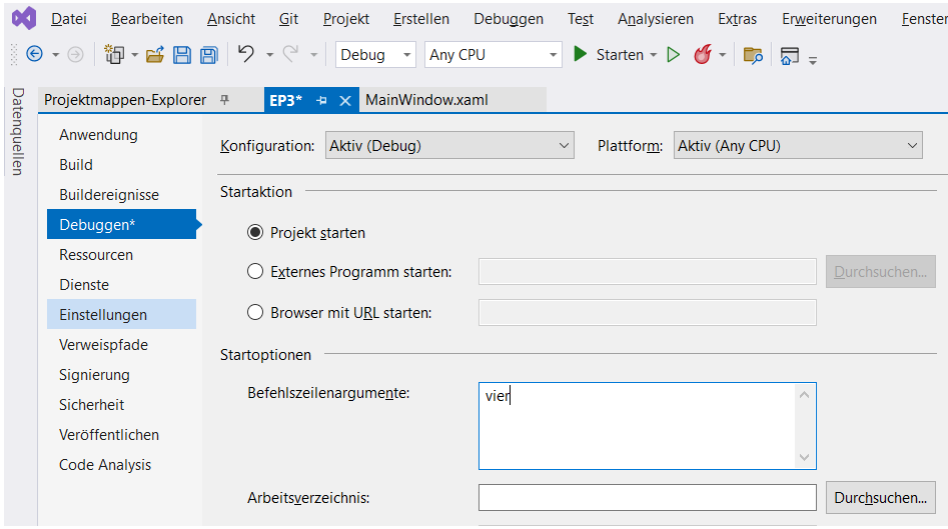
- CLR findet zu einer geworfenen Exception entlang der Aufrufersequenz (bis hinauf zu **Main())** keinen passenden Exception-Handler:

  Programm wird mit einer Fehlermeldung beendet.

- Bsp.:
  - Konsolenprogramm soll die Fakultät einer Zahl berechnen (beim Start als Befehlszeilenargument übergeben)

Hinweis für VS:
*Unter **Projekt > ...-Eigenschaften** kann man auf der Registerkarte **Debuggen** die zu simulierenden **Befehlszeilenargumente** eintragen*

Projekt selektieren->Run->Edit configurations->Program arguments



Main() - Methode beschränkt sich auf die eigentliche Fakultätsberechnung, überlässt die Konvertierung und Validierung der übergebenen Zeichenfolge der Methode Konsole2Int(). Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **Parse()** der BCL-Struktur **Int32**:

Exception handling: unbehandelte Ausnahmen

Bsp. der Fakultätsberechnung:

Main() - Methode beschränkt sich auf die eigentliche Fakultätsberechnung,
überlässt die Konvertierung und Validierung der übergebenen Zeichenfolge der Methode Konsole2Int().

Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **Parse()** der BCL-Struktur **Int32**:

Exception handling: unbehandelte Ausnahmen

```
using System;

class Fakul
{
   static int Kon2Int(string instr)
   {
      int arg = Int32.Parse(instr);
      if (arg >= 0 && arg <= 170)  return arg;
      else  return -1; // Wertebereich!
   }
```

Rückgabewert:
Kon2Int() prüft:

- liegt die aus dem **String**-Parameter ermittelte **int**-Zahl außerhalb des zulässigen Wertebereichs für eine Fakultätsberechnung (mit **double**-Ergebnis-wert)?
- meldet ggf. den Wert -1 als Fehlerindikator zurück.

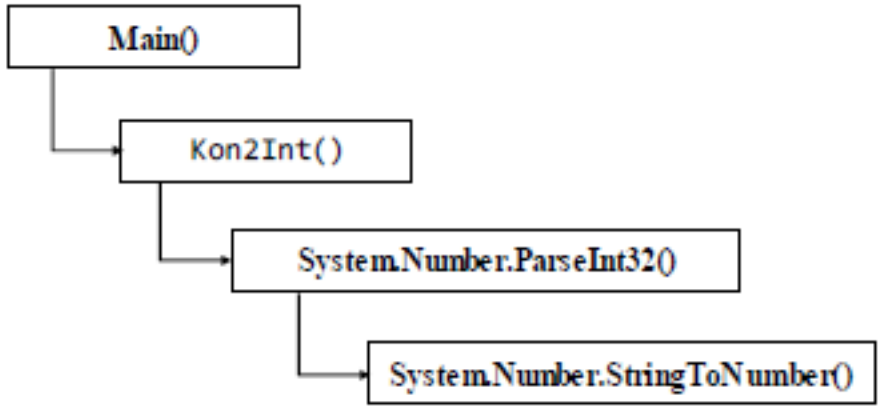
Main() kennt die Bedeutung dieser Rückgabe -> Fakultätsberechnung für ein negatives Argument und ein Argument > 170 kann vermieden werden.

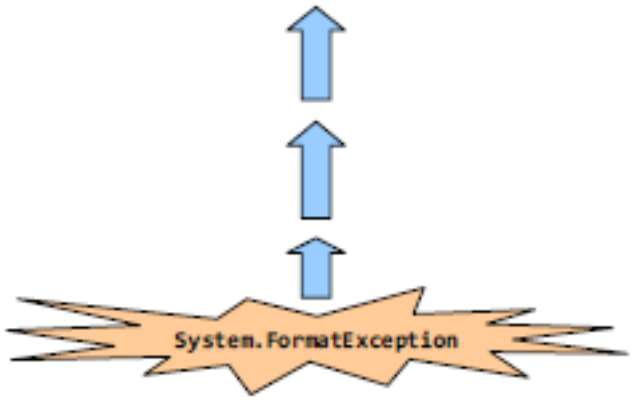
Nicht konvertierbare Zeichenfolge (z. B. „vier“):
BCL-intern aufgerufene Methode **Number.StringToNumber()** wirft **FormatException** - wird vom Laufzeitsystem entlang der Aufrufsequenz an alle beteiligten Methoden bis hinauf zu **Main()** gemeldet

```
   static void Main(string[] args)
   {
      if (args.Length == 0) // IndexOutOfRangeException wird vermieden
      {
         Console.WriteLine("Kein Argument angegeben");
         Console.Read();
         Environment.Exit(1); // Returnwert steht in Umgebungsvariabler
      }                       // ERRORLEVEL zur Verfügung

      int argument = Kon2Int(args[0]);
      if (argument != -1)
      {
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
         fakul = fakul * i;
         Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
      }
      else
      Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
      Console.Read();
   }
}
```

```
>fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

Exception handling: unbehandelte Ausnahmen





Programm endet mit:

```
Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das
falsche Format.
   bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
   bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   bei Fakul.Kon2Int(String s) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 5.
   bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 18.
```

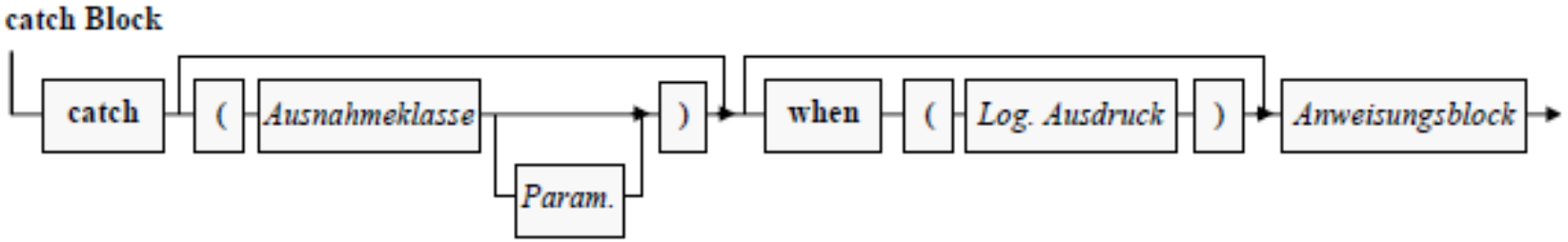
Exceptions im Bsp. abfangen

```
try
{
   // Überwachter Block mit Anweisungen für den regulären Ablauf, wird bei Exception mit goto zu catch() abgebrochen
   // ohne Exception Fortsetzung nach letztem catch()
}
catch (Exceptionklasse1 Parametername)    // Exceptionhandler
{
   // Behandlung von Exceptions aus Exceptionklasse1 oder einer daraus abgeleiteten Klasse, Fortsetzung nach letztem catch()
}
// Optional: abfangen weiterer Ausnahmen:
catch (Exceptionklasse2 Parametername) // Exceptionhandler
{
    // Behandlung von Exceptions aus Exceptionklasse2 oder einer daraus abgeleiteten Klasse, Fortsetzung nach letztem catch()
}
. . .
// Optionaler finally-Block mit Abschluss- bzw. Bereinigungsarbeiten (dann catch-Block nicht unbedingt erforderlich):
finally
{
    Anweisungen, die unabhängig vom Auftreten einer Exception ausgeführt werden sollen (auch nach catch)
}
```

Exceptions im Bsp. abfangen

```
catch (<Exceptionklasse>)    // Exceptionhandler
{
   // ... Exceptionklasse kann ausgewertet werden, wenn vorhanden
}

catch    // Exceptionhandler für generelle Objekte von Exception
{
   // ... Exceptionklasse kann Klasse Exception auswerten/ protokollieren und mit throw; weiterwerfen
}
```

Einschränkung mit *when*:



Ausnahmen im Bsp. abfangen

```
catch ( ExClass )    when 4==a
{
   // ... Exceptionklasse ExClass auswerten, wenn a == 4
}
catch    ( ExClass )    when 5==a
{
   // ... Exceptionklasse ExClass auswerten, wenn a == 5
}
catch    ( ExClass )    when a>3
{
   // ... wird nur ausgeführt, wenn a>5 (nur 1. zutreffendes catch() wird ausgeführt)
}
catch    ( ExClass )
{
   // ... wird nur ausgeführt, wenn a<=3, muss bei gleichen catch() als letztes stehen, wenn ohne Bedingung
}
```

Exceptions im Bsp. abfangen

```
static int Kon2Int(ref string instr)
{
   int arg;
   try
   {    arg = Int32.Parse(instr);          ...
        if (arg < 0 || arg > 170) throw new OverflowException();      // Eingabe war: “-5” (negativ) oder “200” (zu groß für Fakultätsberechnung)
        return arg;
   }
   catch (OverflowException)    {   return -1;   //  Wertebereich verletzt }

   catch (FormatException) when (Double.TryParse(instr, out double d))    // Eingabe war: “4,1” oder “4,0”
   // Wenn die eingegebene Zeichenfolge als Gleitkommazahl interpretierbar ist, wird überprüft, ob sich diese Zahl verlustfrei
   {    // in einen int-Wert wandeln lässt. Dann gewandelte Rückgabe, sonst -2
        arg = (int)d; // arg= ganzzahliger Anteil von d
        if (arg == d)    return ( Int32.Parse(arg.ToString()));
        else  return -2;    // Gleitkommazahl mit Nachkommastelle
    }

    catch (FormatException)                                                  // Eingabe war z.B.: “vier”
    {  return -3; // keine numerische Interpretation der Eingabe möglich
    }
}
```

Exceptions im Bsp. abfangen

Aufgaben eines catch()-Blockes:

- Reparatur (im Beispiel: Double.TryParse(instr, out double d)) und/ oder
- Rückabwicklung: realisierte Effekte des abgebrochenen try-Blockes rückgängig machen und/ oder
- Ersetzung der Exception durch eine informativere Alternative:
  selbst eine Exception werfen, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern und/ oder
- Fehlermeldung und Fehlerprotokollierung
  - Benutzer erhält eine gut verständliche Fehlermeldung.
  - Eintrag in Logdatei: kann dem Software-Entwickler oder Administrator helfen, die Ursache des Fehlers zu finden.
  - Nach einer reinen Fehlermeldung oder –protokollierung: abgefangene Exception erneut werfen mit throw;

case int n when (n >= 0 && n <= 25):

Ausnahmen im Bsp. abfangen

```
static void Main(string[] args)
{
   if (args.Length == 0)
   {       Console.WriteLine("Kein Argument angegeben");  Console.Read();  Environment.Exit(1);  }
   int argument = Kon2Int(ref args[0]);
   switch (argument)
   {
          case int arg when arg >= 0: // case mit Bereich, z.B. auch: case int n when (n >= 0 && n <= 25):
                                       double fakul = 1.0;
                                      for (int i = 1; i <= argument; i++)
                                           fakul = fakul * i;
                                      Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
                                      break;
          case -1:  Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
                    break;
          case -2:  Console.WriteLine("Die Eingabe ist keine ganze Zahl: " + args[0]);
                    break;
          case -3:  Console.WriteLine("Die Eingabe ist nicht numerisch interpretierbar: " + args[0]);
                    break;
   }
}
```

Ausnahmen: finally-Block

Wird unter fast allen Umständen ausgeführt:

- Nach der ungestörten Ausführung des **try**-Blocks
- Nach einer Exceptionbehandlung in einem **catch**-Block (auch beim Verlassen des **catch**-Blocks durch eine neue Exception)
- Nach dem Auftreten einer unbehandelten Exception im **try**-Block
- Beim Verlassen der **try**-Anweisung durch eine **goto**-Anweisung im **try**-Block oder in einem **catch**-Block
- Beim Beenden der Methode durch eine **return**-Anweisung im **try**-Block oder in einem **catch**-Block

Ausnahmen: finally-Block

Die Ausführung des finally-Blocks wird nur verhindert durch:

- Der **try**-Block oder ein **catch**-Block hängt in einer Endlosschleife.
- Im **try**-Block oder in einem **catch**-Block wird die statische Methode **Exit()** der Klasse **Environment** aufgerufen und somit der komplette Prozess terminiert.

Der **finally**-Block ist der ideale Ort zur Freigabe von unmanaged Ressourcen wie Datei-, Datenbank- und Netzwerkverbindungen mit <ressource>.Dispose() .

Anderes Beispiel für das Exception-Handling

```
using System;
class Sequenzen
{
   static int Calc(String arg) // berechnet 10 % arg
   {
      int erg = 0;
      try
      {
         Console.WriteLine("try-Block von Calc()");
         erg = Convert.ToInt32(arg);
         erg = 10 % erg;
      }
      catch (FormatException)
              { Console.WriteLine(" FormatException in Calc(), Ergebnis==0 ");
}
      finally { Console.WriteLine("finally-Block von Calc()"); }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
}
```

```
   static void Main(string[] args)
   {
      try
      {
         Console.WriteLine("try-Block von Main()");
         Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
      }
      catch (ArithmeticException) // z. B. Ganzzahldivision durch 0
              { Console.WriteLine("ArithmeticException-Handler in Main()"); }
      finally { Console.WriteLine("finally-Block von Main()"); }
      Console.WriteLine("Nach try-Anweisung in Main()");
   }
}
```

Beispiele auf den nächsten Folien:

a) Normaler Ablauf: „8“
b) Exception in Calc(), die dort auch behandelt wird
c) Exception in Calc(), die in **Main()** behandelt wird
d) Exception in **Main()**, die nirgends behandelt wird

Beispiel a) Normaler Ablauf („8“)

```
using System;
class Sequenzen
{
   static int Calc(String instr)
   {
      int erg = 0;
      try
      {
        Console.WriteLine("try-Block von Calc()");
        erg = Convert.ToInt32(instr);
        erg = 10 % erg;
      }
      catch (FormatException)
              { Console.WriteLine(" FormatException in Calc(), Ergebnis==0 ");
}
      finally { Console.WriteLine("finally-Block von Calc()"); }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
}
```

```
   static void Main(string[] args)
   {
      try
      {
         Console.WriteLine("try-Block von Main()");
         Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
      }
      catch (ArithmeticException) // z. B. Ganzzahldivision durch 0
              { Console.WriteLine("ArithmeticException-Handler in Main()"); }
      finally { Console.WriteLine("finally-Block von Main()"); }
      Console.WriteLine("Nach try-Anweisung in Main()");
   }
}
```

Ausgaben:

try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % 8 = 2
finally-Block von Main()
Nach try-Anweisung in Main()

Bsp. b) Exception in Calc(), die dort behandelt wird („Acht“)

```
using System;
class Sequenzen
{
   static int Calc(String instr)
   {
      int erg = 0;
      try
      {
        Console.WriteLine("try-Block von Calc()");
        erg = Convert.ToInt32(instr); // loest FormatException aus
        erg = 10 % erg; // loest bei erg==0 ArithmeticException aus
      }
      catch (FormatException)
            { Console.WriteLine("FormatException in Calc(), Ergebnis==0"); }
      finally { Console.WriteLine("finally-Block von Calc()"); }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
}
```

```
static void Main(string[] args)
{
    try
    {
        Console.WriteLine("try-Block von Main()");
        Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
    }
    catch (ArithmeticException) // z. B. Ganzzahldivision durch 0
            { Console.WriteLine("ArithmeticException-Handler in Main()"); }
    finally { Console.WriteLine("finally-Block von Main()"); }
    Console.WriteLine("Nach try-Anweisung in Main()");
}
}
```

Ausgaben:

try-Block von Main()
try-Block von Calc()
FormatException-Handler in Calc(), Ergebnis==0
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % acht = 0
finally-Block von Main()
Nach try-Anweisung in Main()

Bsp. c) Exception in Calc(), die in Main() behandelt wird („0“)

```
using System;
class Sequenzen
{
   static int Calc(String instr)
   {
      int erg = 0;
      try
      {
        Console.WriteLine("try-Block von Calc()");
        erg = Convert.ToInt32(instr); // loest evtl. FormatException aus
        erg = 10 % erg; // loest bei erg==0 ArithmeticException aus
      }
      catch (FormatException)
              { Console.WriteLine("FormatException in Calc(), Ergebnis==0"); }
      finally { Console.WriteLine("finally-Block von Calc()"); }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
}
```

```
static void Main(string[] args)
{
    try
    {
        Console.WriteLine("try-Block von Main()");
        Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
    }
    catch (ArithmeticException) // z. B. Ganzzahldivision durch 0
            { Console.WriteLine("ArithmeticException-Handler in Main()"); }
    finally { Console.WriteLine("finally-Block von Main()"); }
    Console.WriteLine("Nach try-Anweisung in Main()");
}
}
```

```
try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
ArithmeticException-Handler in Main()
finally-Block von Main()
Nach try-Anweisung in Main()
```

Bsp. d) Exception in Main(), die nirgends behandelt wird (-)

```
using System;
class Sequenzen
{
   static int Calc(String instr)
   {
      int erg = 0;
      try
      {
         Console.WriteLine("try-Block von Calc()");
         erg = Convert.ToInt32(instr); // loest evtl. FormatException aus
         erg = 10 % erg; // loest ArithmeticException aus
      }
      catch (FormatException)
               { Console.WriteLine("FormatException in Calc(), Ergebnis==0"); }
      finally { Console.WriteLine("finally-Block von Calc()"); }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
}
```

```
static void Main(string[] args)
{
    try
    {
        Console.WriteLine("try-Block von Main()");
        Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
    }                                   // wirft IndexOutOfRangeException
    catch (ArithmeticException) // z. B. Ganzzahldivision durch 0
            { Console.WriteLine("ArithmeticException-Handler in Main()"); }
     finally { Console.WriteLine("finally-Block von Main()"); }
     Console.WriteLine("Nach try-Anweisung in Main()");
}
}
```

```
try-Block von Main()
finally-Block von Main()
```

Ausnahmebehandlung – geschachteltes try

try-Blöcke sind schachtelbar:

```
try
{           // Anfang try-Block1
    ...
    try
    {         // Anfang try-Block2
            ...
    }         // Ende try-Block2
    catch  (int  i)  { ... }  // Abfangen von Fehlern aus Block2
    catch (char c) { ... }
    ...
}             // Ende try-Block1
catch  (  float  f) { ... }  // Abfangen von Fehlern aus Block1

catch ( double *dp) { ...}
...
```

- Wird ein Exceptionobjekt am Ende eines try-Blockes nicht abgefangen, so wird es an den übergeordneten try-Block weitergeleitet.

Returnwerte versus Exceptionhandling

```
static void Main()
{
   int returncode;
   returncode = M1();
   // Behandlung von pot. M1() - Fehlern
   if (returncode == 1)
   {    // ...
    Environment.Exit(11);
   }
   ...
   returncode = M2();
   // Behandlung von pot. M2() - Fehlern
   if (returncode == 1)
   {   // ...
      Environment.Exit(21);
   }
   ...
   returncode = M3();
   // Behandlung von pot. M3() - Fehlern
   if (returncode == 1)
   {
      // ...
      Environment.Exit(31);
   }
   ...
}
```

```
static void Main()
{
    try
    {
        M1();
        M2(); // wird bei Exception in M1() nicht mehr ausgeführt
        M3(); // wird bei Exception in M1() oder M2() nicht mehr ausgeführt
    }
    catch (ExA a)
    {
        // Behandlung von Ausnahmen aus der Klasse ExA
    }
    catch (ExB b)
    {
        // Behandlung von Ausnahmen aus der Klasse ExB
    }
    catch (ExC c)
    {
        // Behandlung von Ausnahmen aus der Klasse ExC
    }
}
```

Returnwerte versus Exceptionhandling

Nachteile der Nutzung für Fehlerrückgabe:

- Ungesicherte Beachtung von Rückgabewerten.
  Gut gesetzte Rückgabewerte nützen nichts, wenn sich die Aufrufer nicht darum kümmern.

- Umständliche Weiterleitung von Fehlern.
  Wenn ein Fehler nicht an Ort und Stelle behandelt werden soll, muss die Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden.

- Beschränkung auf Methoden
  Die Rückgabe eines Fehlerindikators ist keine Option bei Eigenschaften, Indexern, Ereignissen oder überladenen Operatoren.

Returnwerte versus Exceptionhandling

Vorteile von Exceptions

- **Bessere Lesbarkeit des Quellcodes**
  Bessere Trennung zwischen den Anweisungen für den normalen Programmablauf und den Exceptionbehandlungen im Quellcode.

- **Garantierte Beachtung von Ausnahmen**
  Exceptions können nicht ignoriert werden. Reagiert ein Programm nicht darauf, wird es vom Laufzeitsystem beendet.

- **Automatische Weitermeldung bis zur bestgerüsteten Methode**
  Oft ist der unmittelbare Verursacher nicht gut gerüstet zur Behandlung einer Ausnahme, z. B. nach dem vergeblichen Öffnen einer Datei. Dann entscheidet eine „höhere“ Methode über das weitere Vorgehen und erfragt z. B. beim Benutzer eine alternative Datei.

- **Bessere Fehlerinformationen für den Aufrufer/ Ausgabe**
  Über ein Exception-Objekt kann der Aufrufer sehr genau über den aufgetretenen Fehler informiert werden (bei einem traditionellen Rückgabewert nicht der Fall).

Exception handling in Bibliotheken

Bei der Entscheidung für eine Technik zur Fehlerkommunikation innerhalb von Applikationen ist u.a. die Wahrscheinlichkeit für das Auftreten des Fehlers relevant:

- Tritt ein **Fehler mit erheblicher Wahrscheinlichkeit** auf:

  Aufrufer sollte mit dem Problem rechnen und Methode per Rückgabewert kommunizieren.

- Bei **Fehlern mit geringer Wahrscheinlichkeit** haben häufige, meist überflüssige Kontrollen eine Leistungseinbuße zur Folge. Hier sollte man es besser auf eine Exception ankommen lassen.

Exception handling in Bibliotheken

Implementierung einer Bibliotheksmethode für einen größeren Nutzerkreis:
Microsoft *Design Guidelines:* **DO NOT** return error codes.

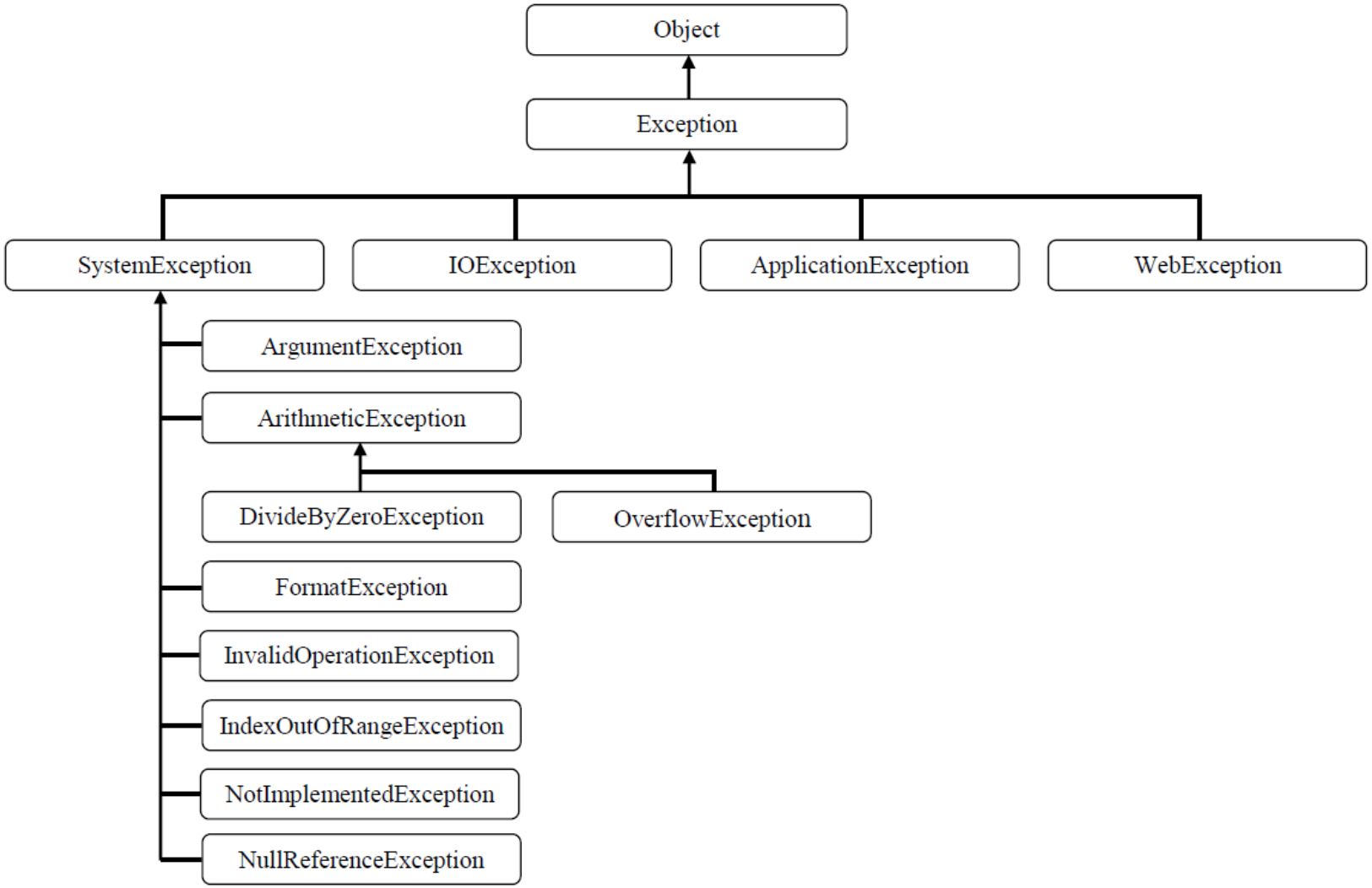
Exceptions are the primary means of reporting errors in frameworks.

- Entwickler einer Bibliotheksfunktion:
  wirft Fehlerobjekt mittels throw , wenn in seiner Funktion eine nicht behebbare Ausnahmesituation erkannt wird

- Anwendungsentwickler:
  ruft Bibliotheksfunktionen in einem try-Block auf, kann seinen Algorithmus als Ganzes entwickeln, ohne immer wieder an die Behandlung von Fehlern denken zu müssen.
  Muss sich im Anschluss an den try-Block um alle während des Algorithmus möglicherweise aufgetretenen Fehler kümmern.

Exceptionklassen in der BCL

Kleiner Ausschnitt aus im .NET - Framework vordefinierten Exceptionklassen (Klassenhierarchie):



Exception in der BCL: Properties der Klasse *Exception*

Data	Auflistung von Schlüssel-Wert-Paaren mit zusätzlichen benutzerdefinierten Informationen zur Ausnahme. Autor kann Zusatzinformationen zur Exception in beliebig langer Schlüssel-Wert-Liste mit Elementen vom Strukturtyp **DictionaryEntry** mit der **IDictionary**-Methode **Add()** einfügen.
HelpLink	Link zur Hilfedatei, die dieser Ausnahme zugeordnet ist.
HResult	Codierter Wert, der einer bestimmten Exception zugeordnet ist.
InnerException	Viele **catch**-Blöcke sind Informationsvermittler und werfen selbst eine Ausnahme für einen leichter verständlichen Unfallbericht. Ruft die Exception-Instanz ab/ übergibt die, die die aktuelle Ausnahme verursacht hat.
Message	Meldung, mit der die aktuelle Ausnahme beschrieben wird.
Source	Name der Anwendung oder des Objekts, die/ das den Fehler verursacht hat.
StackTrace	Darstellung der unmittelbaren Frames in der Aufrufliste (Dateinamen und Zeilennummern, wenn für die Projekt-Erstellungskonfiguration die **Debuginformationen** nicht abgeschaltet waren).
TargetSite	Methode, die die aktuelle Ausnahme auslöste.

Exception in der BCL: Properties der Klasse *Exception*

ToString()-Methode nutzbar für Properties eines **Exception**-Objekts :

- den Namen der Ausnahmeklasse
- die **Message**-Zeichenfolge (die beim Erzeugen der Ausnahme formulierte Fehlermeldung)
- die **StackTrace**-Zeichenfolge (die Aufrufreihenfolge)

Exception-Klassen der BCL

Ausnahme	Bedingung
ArgumentException	Ein Nicht-NULL-Argument, das an eine Methode übergeben wird, ist ungültig.
ArgumentNullException	Ein Argument, das an eine -Methode übergeben wird, ist null.
ArgumentOutOfRangeException	Ein Argument liegt außerhalb des Bereichs der gültigen Werte.
DirectoryNotFoundException	Ein Teil eines Verzeichnispfads ist ungültig.
DivideByZeroException	Der Nenner in einem Ganzzahl- oder Decimal Divisionsvorgang ist 0.
DriveNotFoundException	Ein Laufwerk ist nicht verfügbar oder nicht vorhanden.
FileNotFoundException	Eine Datei ist nicht vorhanden.
FormatException	Ein Wert hat kein geeignetes Format, um von einer Konvertierungsmethode wie Parseaus einer Zeichenfolge konvertiert zu werden.
IndexOutOfRangeException	Ein Index befindet sich außerhalb der Grenzen eines Arrays oder einer Auflistung.
InvalidOperationException	Ein Methodenaufruf ist im aktuellen Zustand eines Objekts ungültig.
KeyNotFoundException	Der angegebene Schlüssel für den Zugriff auf ein Element in einer Auflistung wurde nicht gefunden.
NotImplementedException	Eine Methode oder ein Vorgang wird nicht implementiert.
NotSupportedException	Eine Methode oder Operation wird nicht unterstützt.
ObjectDisposedException	Ein Vorgang wird für ein Objekt ausgeführt, das verworfen wurde.
OverflowException	Ein Arithmetik-, Umwandlungs- oder Konvertierungsvorgang führt zu einem Überlauf.
PathTooLongException	Ein Pfad- oder Dateiname überschreitet die maximale systemdefinierte Länge.
PlatformNotSupportedException	Der Vorgang wird auf der aktuellen Plattform nicht unterstützt.
RankException	Ein Array mit der falschen Anzahl von Dimensionen wird an eine -Methode übergeben.
TimeoutException	Das einem Vorgang zugewiesene Zeitintervall ist abgelaufen.
UriFormatException	Es wird ein ungültiger URI (Uniform Resource Identifier) verwendet.

Exceptions werfen (Fehlersignalisierung)

Insbesondere sollte man an Methoden und Konstruktoren übergebene Parameterwerte routinemäßig prüfen und ggf. die Ausführung durch das Werfen einer Exception abbrechen.

Auslösen einer Exception: **throw**-Anweisung, enthält eine Referenz auf ein Exceptionobjekt (aus der Klasse **System.Exception** oder einer abgeleiteten Klasse):

```
try
{
   argument = Kon2Int(args[0]);  // 188
}
catch (Exception e) {   Console.WriteLine(e.Message); }
```

```
Argument ausserhalb [0, 170]
Parametername: instr
Der tatsächliche Wert war 188.
```

```
static int Kon2Int(string instr)
{
   int arg;
   arg = Int32.Parse(instr);
   if (arg < 0 || arg > 170)
         throw new ArgumentOutOfRangeException (nameof(instr),     arg,          "Argument ausserhalb [0, 170]");
                                                  // Parametername, Parameterwert, Errormessage
   else return arg;
}
```

Exceptions werfen (Fehlersignalisierung)

throw als Ausdruck:

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  static double Log2(double arg) {`
`  return arg > 0 ?`
`  Math.Log(arg) / Math.Log(2) :`
`  throw new ArgumentException("arg must be positive");`
`  }`
 	
`  static void Main(string[] args) {`
`  Console.WriteLine(Log2(8));`
`  Console.WriteLine(Log2(0));`
`  }`
`}`
	`3`
	`Unbehandelte Ausnahme:`
	`System.ArgumentException:`
	`arg must be positive`

throw; // ohne Parameter: wirft zuletzt gefangene Exception erneut

Gerade behandelte Exception wird erneut geworfen, wenn ...

- von einem Aufrufer lediglich ein Protokolleintrag geschrieben werden soll, ohne in die Problembehandlung einzugreifen,
- sich nach einem Lösungsversuch herausstellt, dass der **catch**-Block das Problem nicht aus der Welt schaffen kann, sodass die Ausnahme an den Vorgänger in der Aufrufverschachtelung übergeben werden muss.

StackTrace-Informationen bleiben im Exception-Objekt erhalten!

Eigene Exceptions definieren

Microsofts Empfehlungen für selbst definierte Ausnahmeklassen:

- Als Basisklasse sollte **System.Exception** verwendet werden, z. B.: `public class BadFaculArgException : Exception { ... }`
- Der Klassenname sollte mit dem Wort *Exception* enden.
- Die folgenden *allgemeinen Konstruktoren für Exceptions* sollten mit **public** - Verfügbarkeit implementiert werden:
  - o Ein parameterfreier Konstruktor, z. B.: `public BadFaculArgException() { }`
  - o Ein Konstruktor mit einem **string**-Parameter für die Fehlermeldung, z. B.:
    `public BadFaculArgException(string message) : base(message) { }`
  - o Ein Konstruktor mit einem **string**- Parameter für die Fehlermeldung und einem **Exception**-Parameter für eine innere Exception, die zuvor gefangen wurde und nun in eine informativere Exception als Anlage aufgenommen wird, z. B.:

```
public BadFaculArgException(string message, Exception innerException) : base(message, innerException) { }
```

Eigene Exceptions definieren

```
using System;
public class BadFaculArgException : Exception   // Ableitung von Exception, bad argument in function faculty()
{
   int type=-1, value = -1;
   string input;
   public BadFaculArgException() { } // parameterfreier Konstruktor
   public BadFaculArgException(string message) : base(message) { }  // mit String-Parameter für Fehlermeldung
   public BadFaculArgException(string message, Exception innerException) : base(message, innerException) { } // mit der inner exception
   public BadFaculArgException(string message, string input_, int type_, int value_, Exception innerException) : base(message, innerException)
   {
      input = input_;                                  // zu konvertierende Zeichenfolge
      if (type_ >= 0 && type <= 3)  type = type_;      // numerischer Indikator für die Fehlerart:
      if (type_ == 4 && (value_ < 0 || value_ > 170))  // 0: Unbekannt
      {                                                 // 1: Argument hat den Wert null, 2: Zeichenfolge kann nicht konvertiert werden
         type = type_;                                 // 3: int-Überlauf , 4: int-Wert außerhalb [0, 170]
         value = value_;
      }
   }
   public string Input {get {return input;}}
   public int Type {get {return type;}}
   public int Value {get {return value;}}
}
```

Eigene Exceptions: Nutzung in Kon2Int()

```
static int Kon2Int(ref string instr)
{
   int arg;
   try
   {    arg = Int32.Parse(instr);
        if (arg < 0 || arg > 170) throw new OverflowException(); return arg;
   }
   catch (OverflowException)     {   return -1;   //  Wertebereich verletzt }

   catch (FormatException) when (Double.TryParse(instr, out double d))
   {
         arg = (int)d; // arg= ganzzahliger Anteil von d
        if (arg == d)    return ( Int32.Parse(arg.ToString()));
        else  return -2;    // Gleitkommazahl mit Nachkommastelle
    }

    catch (FormatException)                  // Eingabe war z.B.: “vier”
   {  return -3; // keine numerische Interpretation der Eingabe möglich
    }
}
```

Eigene Exceptions: Nutzung in Kon2Int()

```
try
{
    arg = Convert.ToInt32(instr);
}
catch (ArgumentNullException e)
{
     throw new BadFaculArgException("Kein Argument vorhanden", null, 1, -1, e);
}
catch (FormatException e) when (Double.TryParse(instr, out double d)) // double!
{
     arg = (int)d; // d: gültiger double, arg: ganzzahliger Anteil
     if (arg == d) return ( Int32.Parse(arg.ToString()));
     else    throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);  // Fehlerart 2: double.x
}
catch (FormatException e)    {  throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);  } // Fehler 2: Text
catch (OverflowException e) {  throw new BadFaculArgException("Ganzzahl-Überlauf", instr, 3, -1, e);  }  // Fehler 3: Wert > int32

if (arg < 0 || arg > 170)          throw new BadFaculArgException("Wert außerhalb [0, 170]", instr, 4, arg); // Fehler 4
else  return arg; // nur, wenn keine oder keine neue Exception geworfen
```

```
static int Kon2Int(ref string instr)
{
   int arg;
   try
   {   arg = Int32.Parse(instr);
       if (arg < 0 || arg > 170) throw new OverflowException(); return arg;
   }
   catch (OverflowException)    {  return -1;  //  Wertebereich verletzt }

   catch (FormatException) when (Double.TryParse(instr, out double d))
   {
        arg = (int)d; // arg= ganzzahliger Anteil von d
       if (arg == d)    return ( Int32.Parse(arg.ToString()));
       else  return -2;    // Gleitkommazahl mit Nachkommastelle
    }

    catch (FormatException)                  // Eingabe war z.B.: “vier”
   {  return -3; // keine numerische Interpretation der Eingabe möglich
    }
}
```

Eigene Exceptions definieren: Nutzung

```
static void Main(string[] args)
{
   int argument = -1;
   if (args.Length == 0)
   {
      Console.WriteLine("Kein Argument angegeben");
      Console.Read();
      Environment.Exit(1);
   }
   try
   {
      argument = Kon2Int(ref args[0]);
      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
      fakul = fakul * i;
      Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
   }
```

```
   catch (BadFaculArgException e)
   {
      Console.WriteLine($"Message:\t{e.Message}");
      Console.WriteLine($"Fehlertyp:\t{e.Type}\nEingabe:\t{e.Input} ");
      Console.WriteLine($"Wert:\t\t{e.Value}");
      if (e.InnerException != null)
         Console.WriteLine($"Orig. Message:\t{e.InnerException.Message}");
      Console.Read();
      Environment.Exit(1);
   }
}
```

Programmstart mit dem Befehlszeilenargument „vier“:

Message: Fehler beim Konvertieren
Fehlertyp: 2
Eingabe: vier
Wert: -1
Orig. Message: Die Eingabezeichenfolge hat das falsche Format.

Übung 1

Im Beispielprogramm zur Demonstration von möglichen Sequenzen bei der Ausnahmebehandlung verzichtet die Methode Calc() darauf, die potentiell von der Methode **Convert.ToInt32()** zu erwartende **OverflowException** abzufangen.
Bleibt die Ausnahme unbehandelt? Wenn nicht: warum nicht?

```
using System;
public class S
{
   public static int Calc(String instr)
   {
      int erg = 0;
      try
      {
         Console.WriteLine("try-Block von Calc()");
         erg = Convert.ToInt32(instr);
         erg = 10 % erg;
      }
      catch (FormatException)
            {  Console.WriteLine("FmtEx-Handler in Calc()"); }
      finally { Console.WriteLine("finally-Block von Calc()");  }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
   }
```

```
   static void Main(string[] args)
   {
      try
      {
         Console.WriteLine("try-Block von Main()");
         Console.WriteLine("10 % " + args[0] + " = " + S.Calc(args[0]));
      }
      catch (ArithmeticException)
      {
         Console.WriteLine("ArithmeticException-Handler in Main()");
      }
      finally
      {
         Console.WriteLine("finally-Block von Main()");
      }
      Console.WriteLine("Nach try-Anweisung in Main()");
   }
}
```

Lösung 1

Übung 2

Rechnen mit Gleitkommazahlen:
Dabei produziert C# in kritischen Situationen *keine* Ausnahmen, sondern operiert mit speziellen Grenzwerten wie **Double.POSITIVE_INFINITY** oder **Double.NaN**.
-> Fehlersuche ist schwer, wenn mit diesen Werten weitergerechnet wird, und am Ende das Ergebnis **NaN** auftaucht
Beispiel: Methode Log2() zur Berechnung des dualen Logarithmus greift auf die BCL-Methode **Math.Log()** zurück und liefert daher bei ungeeigneten Argumenten (<= 0) als Rückgabewert **Double.NaN:**

```
using System;
class Prog
{
   static double Log2(double arg) { return Math.Log(arg) / Math.Log(2); }    // Log10(arg)/ Log10(2)

   static void Main()
   {
      double a = Log2(-1);   // == NaN
      double b = Log2(8);    // == 3
      Console.WriteLine(a*b);  // „NaN“
   }
}
```

Erstellen Sie eine Version der Methode Log2(), die bei ungeeigneten Argumenten (<0) eine sprechende **ArgumentOutOfRange - Exception** wirft. Fangen Sie diese in Main(), und geben Sie sie aus.

Übung 3

Werfen und fangen Sie statt der ArgumentOutOfRange-Exception eine eigene, davon abgeleitete Exception.