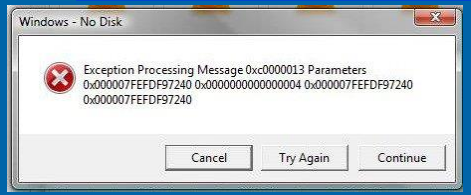
TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Ausnahmebehandlung mit Exceptions



try…catch…finally

Ausnahme- und Fehlerbehandlung mit Exceptions

Fehlercodes

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

- Fehlercodes helfen bei der Diagnose von ungewolltem Verhalten
- weite Verbreitung
- leicht zu verstehen

```
int main(void)
{
    int result = printf("This is a test!");
    if (result > 0)
    {
        printf("%d characters were written!", result);
        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```

- bei vielen Fehlerquellen anwendbar

- schnell unübersichtlich
- kann neben Fehlern auch Status-Codes übertragen
- keine saubere Trennung zwischen Programmcode und Fehlerbehandlung
- viel Code ist für die Implementierung notwendig

```c
// some C-Code:
int result = someFunction();
switch (result)
{
    case ErrorCode1:
        ...
        break;
    case ErrorCode2:
        ...
        break;
    ...
    case ErrorCode993647:
        ...
        break;
    default:
        //alles OK
}
```

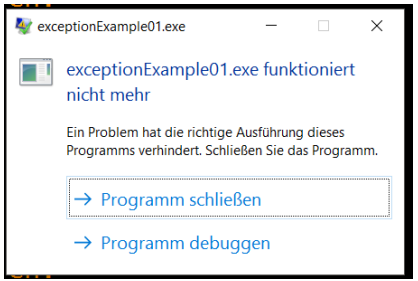
konkretes Beispiel (C-Code)

```c
#include <stdio.h>

int Berechnung(int a, int b, int c, int *Ergebnis) {
    if ((long long) a + (long long) b != a+b) return -2; // arithmetischer Überlauf
    if(c == 0) return -1; // Division durch 0
    *Ergebnis = (a + b) / c;
    return 0;
}

int main() {
    int a = 27;
    int b = +2147483647; // integer max
    int c = 0, d = 0, errorCode = 0;
    errorCode = Berechnung(a,b,c,&d);
    switch (errorCode) {
        case  0: printf("%d\n", d); break;
        case -1: printf("[ERROR] Division durch Null"); break;
        case -2: printf("[ERROR] arithmetischer Ueberlauf"); break;
    }
    return errorCode;
}
```

konkretes Beispiel (C#-Code)

- Funktion berechnet zuerst die Summe der Zahlen a und b
- anschließend wird durch c dividiert
- mögliche Fehler:
  - Division durch Null
  - arithmetischer Überlauf

```
using System;

class ExceptionExample
{
    static int Main()
    {
        int a = 27;
        int b = int.MaxValue;
        int c = 0;
        int d = 0;

        d = (a + b) / c;

        return d;
    }
}
```



Alternative: Exceptions (Ausnahmen)

- gesicherte Code-Blöcke können auf Fehler achten
- Programm wird bei Fehlern nicht augenblicklich geschlossen
- Nutzer bekommt sinnvolle Fehlermeldung
- saubere Trennung zwischen Fehlerbehandlung und restlichem Code

```
try
{
    // Code mit möglicher Ausnahme
}
catch (Exception e)
{
    //Ausnahmebehandlung
}
```

```
using System;

class ExceptionExample
{
    static int Main()
    {
        int a = 27;
        int b = int.MaxValue;
        int c = 0;
        int d = 0;

        try
        {
            d = (a + b) / c;
        }
        catch (Exception e)
        {
            Console.WriteLine("Oje. Ein Fehler! Was ist nur passiert?");
        }
        return d;
    }
}
```

```
>exceptionExample02.exe
Oje. Ein Fehler! Was ist nur passiert?

>echo %errorlevel%
0

>
```

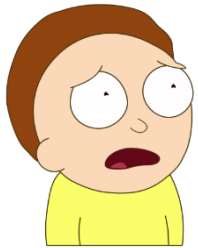
Ausnahme

```
    try
    {
        d = (a + b) / c;
    }
    catch (Exception e)
    {
        Console.WriteLine("Fehler! - " + e.Message);
        d = -5;
    }
    Console.WriteLine("Hier geht es trotzdem weiter");
    return d;
}
```

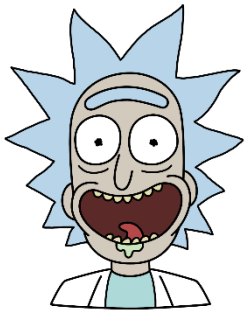
- Fehler wird erkannt
- Informationen werden mitgegeben
- Code wird nach der **Exception** weiter ausgeführt

```
>exceptionExample03.exe
Fehler! - Es wurde versucht, durch 0 (null) zu teilen.
Hier geht es trotzdem weiter

>echo %errorlevel%
-5

>
```

Was ist mit dem arithmetischen Überlauf? Warum wird der nicht erkannt? Der tritt doch zuerst auf.



Der arithmetische Überlauf ist streng genommen kein Fehler. Aber C# bietet es an, darauf zu prüfen, wenn man das möchte. Dazu gibt es den **checked**-Befehl.



Arithmetischer Überlauf in C#

```
try
{
    d = checked(a + b);
}
catch (Exception e)
{
    Console.WriteLine("Fehler! - " + e.Message);
    d = a + b;
}
return d;
```

```
>exceptionExample04.exe
Fehler! - Die arithmetische Operation hat einen Überlauf verursacht.

>echo %errorlevel%
-2147483622

>
```

Gezielt Exceptions fangen

```
try
{
    d = checked((a + b) / c);
}
catch (OverflowException e)
{
    Console.WriteLine("[ERROR] arithmetischer Überlauf! - " + e.Message);
    d = -2;
}
catch (DivideByZeroException e)
{
    Console.WriteLine("[Error] Division durch Null - " + e.Message);
    d = -1;
}
Console.WriteLine("Hier geht es trotzdem weiter");
return d;
```

Viele Funktionen werfen **Exceptions**. Hier gibt es Hilfe dazu:
https://msdn.microsoft.com/de-de/

Exceptions folgen einer klaren Hierarchie. Somit kann man Fallback-Lösungen implementieren oder auch Fehler weiterreichen.

```
catch (OverflowException e)
{
    Console.WriteLine("[ERROR] arithmetischer Überlauf! - " + e.Message);
    d = -2;
}
catch (DivideByZeroException e)
{
    Console.WriteLine("[ERROR] Division durch Null - " + e.Message);
    d = -1;
}
catch (Exception e)
{
    Console.WriteLine("[ERROR] Ein unbekannter Fehler - " + e.Message);
    d = -3;
}
Console.WriteLine("Hier geht es trotzdem weiter");
return d;
```

Hierarchie

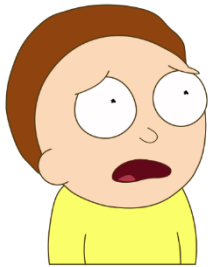
```
>exceptionExample07.exe
[ERROR] Es wurde versucht, durch 0 (null) zu teilen.

>echo %errorlevel%
-1

>
```

Exceptions hangeln sich nach oben, bis ein passender **catch-Block** gefunden wurde.

Wird kein passender **catch-Block** gefunden, kommt es zur Behandlung in der Laufzeitumgebung.

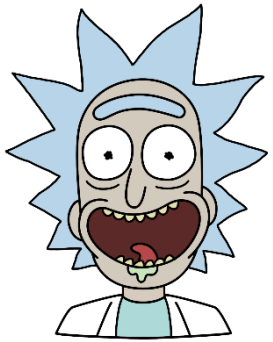
```
static int Berechnung(int a, int b)
{
    try
    {
        return a / b;
    }
    catch (OverflowException e)
    {
        Console.WriteLine("[ERROR] " + e.Message);
        return -2;
    }
}

static int Main()
{
    try
    {
        return Berechnung(2, 0);
    }
    catch (DivideByZeroException e)
    {
        Console.WriteLine("[ERROR] " + e.Message);
        return -1;
    }
}
```

Finally



Was ist, wenn ich in einer Funktion nach einem Fehler unbedingt noch etwas machen muss? Zum Beispiel eine Datei schließen oder eine Server-Verbindung schließen?
Aber meine **Exception** will ich woanders behandeln.

Auch dafür gibt es eine tolle Möglichkeit in C#. **catch** ist nicht das einzige, was auf **try** folgen kann. Es gibt noch **finally**.
Damit wird ein Code-Block definiert, der definitiv ausgeführt wird, auch wenn es eine **Exception** gibt.



Beispiel: Serveranfrage

```
>exceptionExample08.exe
Verbindungsaufbau zum Server
Verbindung wird sauber geschlossen
Ein Fehler ist aufgetreten
Danke!

>echo %errorlevel%
0

>
```

```
static void ServerAnfrage(int a, int b)
    {
        try
        {
            Console.WriteLine("Verbindungsaufbau zum Server");
            int test = a / b; // Hier ein Fehler
            Console.WriteLine("Hier ist die eigentliche Anfrage");
        }
        finally
        {
            Console.WriteLine("Verbindung wird sauber geschlossen");
        }
    }

    static void Main(string[] args)
    {
        try
        {
            ServerAnfrage(2, 0);
            Console.WriteLine("Hier passiert Arbeit mit den Serverdaten");
        }
        catch (Exception)
        {
            Console.WriteLine("Ein Fehler ist aufgetreten");
        }
        Console.WriteLine("Danke!");
    }
```

Effizienz und Overhead

- Ausnahmebehandlung verursacht in C# nur minimalen Overhead
- Beim Eintreten eines Ausnahmezustandes, verringert sich die Effizienz stark

Entwurfsrichtlinien - Zusammenfassung

- **Exceptions** sollten nur Fehler markieren und beim üblichen Codeablauf nicht auftreten
- In Rückgabewerten sollten keine Fehlerinformationen weitergeleitet werden (Hier gibt es Ausnahmen von der Regel)
- **Finally-Blöcke** sollten vor allem bei Ressourcenverwendung (Dateisystem, Netzwerk, Geräte) genutzt werden.
- sinnvolle Fehlerbehandlung hilft beim Programmieren