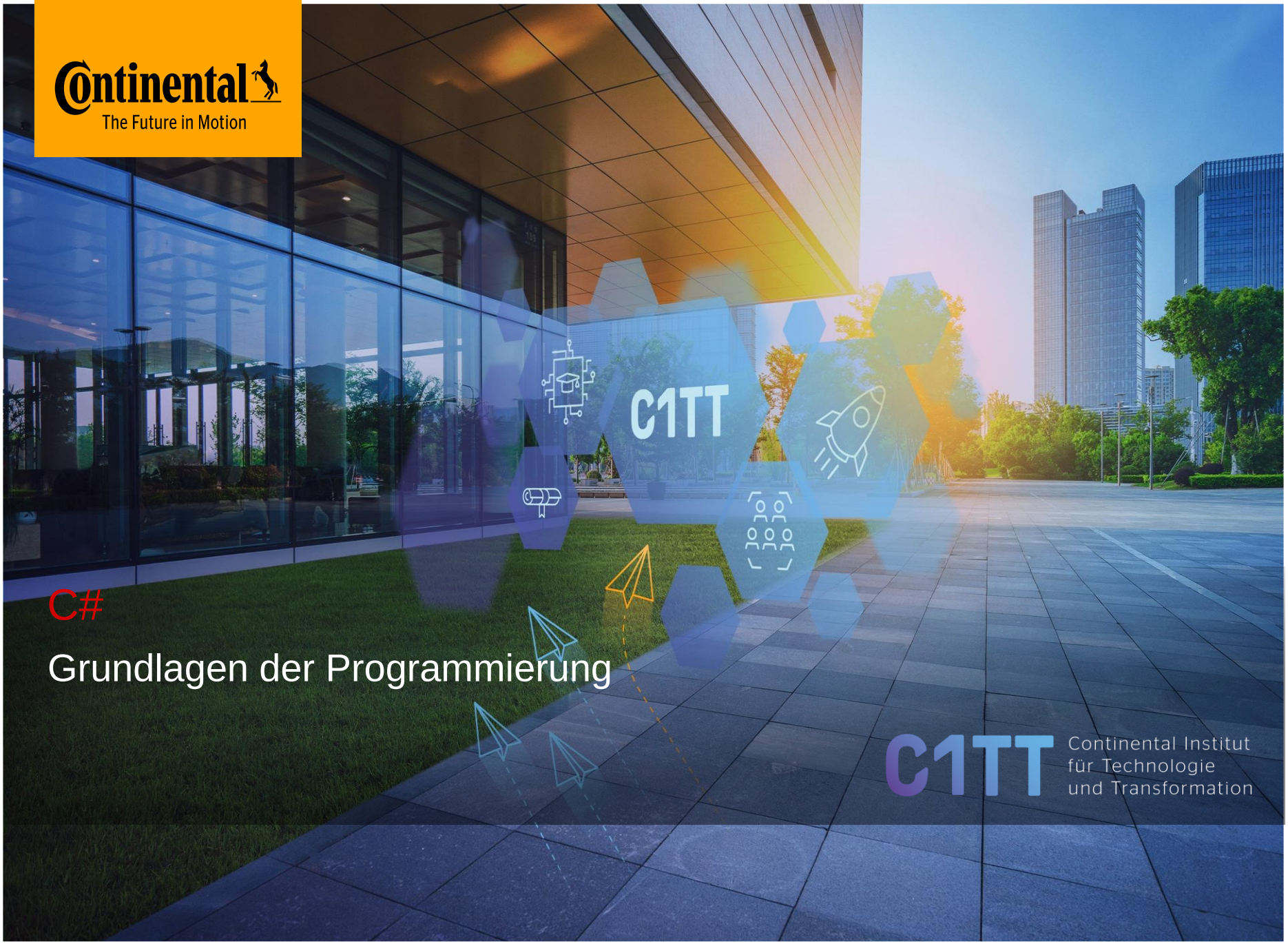
Continental
The Future in Motion
C1TT
C#
Grundlagen der Programmierung
C1TT
Continental Institut
für Technologie
und Transformation

Continental Institut
für Technologie
und Transformation

C#

Grundlagen der Programmierung

Kapitel 1

EINFÜHRUNG IN .NET

Was ist .NET? (1/3)

- Virtuelle Maschine
  - Isoliert Programm von Hardware und Betriebssystem
  - Ähnlich Java, aber sprachübergreifend
  - Keine Auswirkung auf Programmoberfläche
- Standardisiert von ECMA und ISO

Was ist .NET? (2/3)

- Zu .NET zählt Microsoft
  - .NET Framework und .NET Core, bilden Laufzeitumgebung
  - Entwicklungsumgebung Visual Studio
  - Server-Software wie Windows Server, SQL Server und BizTalk Server
  - Client-Software wie Windows, Windows Phone und Microsoft Office, aber auch Linux und OS X

Was ist .NET? (3/3)

- Zum .NET Framework zählen u.a.:
  - ASP.NET
    - ASP.NET Forms
    - ASP.NET MVC
  - ADO.NET
  - Windows Forms
  - Windows Presentation Foundation (WPF)
  - Windows Communication Foundation (WCF)
  - Windows Workflow Foundation
  - Windows Identity Foundation
  - LINQ
  - Parallel Extensions
  - Entity Framework

- Zu .NET Core zählen:
  - ASP.NET Core
  - Entity Framework Core

Warum .NET? (1/5)

- Das Ende der DLL-Hell
  - Löschen oder Überschreiben gemeinsam genutzter DLLs
  - COM-DLLs konnten nur einmal auf Rechner installiert sein
  - .NET: Registry überflüssig
  - .NET ermöglicht parallele Installation mehrerer DLL-Versionen
  - .NET-Programm enthält Informationen über benötigte DLLs
  - .NET: Umbenennen / Verschieben v. Dateien / Verzeichnissen OK

Warum .NET? (2/5)

- Vereinfachte Installation, Konfiguration und Deinstallation
  - .NET-Programme: keine Registry-Einträge
    -> XCopy-Bereitstelllung
  - ClickOnce-Bereitstellung
  - .NET: Konfigurationsdaten in XML-Datei im Programmverzeichnis
    - Leicht zu vergleichen / versionieren
    - Leicht zu ändern, auch per Skript
  - .NET: Zusätzlich Konfigurationsdateien auf Rechner- oder Unternehmensebene möglich
    -> Admin freut sich
  - .NET: Deinstallation = Verzeichnis löschen

Warum .NET? (3/5)

- Neue Infrastruktur für verteilte Programme
  - COM / DCOM nur im Intranet geeignet
  - COM außerhalb von Windows kein Standard
  - Konkurrenz durch Java
  - Windows Communication Foundation (WCF)
  - .NET Web Services = plattformübergreifender Standard
  - WebAPI
  - Quelloffenenes .NET Core

Warum .NET? (4/5)

- Einheitliches Programmiermodell
  - Windows-Programmierung historisch gewachsen
    → Probleme, z. B.
    - Fehlerbehandlung uneinheitlich
    - Unterschiedl. Typen + Aufrufkonv. der Programmiersprachen
  - .NET: einheitliches Programmiermodell
    - Einheitliche, objektorientierte Bibliothek
    - Vereinheitlichte Fehlerbehandlung
    - Einheitliches, erweiterbares Typsystem
    - Einarbeiten in Programmiersprache = Syntax lernen

Warum .NET? (5/5)

- Mehr Sicherheit
  - Programmierer verwalten Speicher -> Probleme
    - Speicherlecks
    - Buffer-Overrun-Angriffe
  - Speichermanipulationen durch Zeiger -> mehr Probleme
    - Programmierer evtl. überfordert
    - Bösartiger Code möglich
  - .NET: alles wird gut
    - Garbage Collector = keine Speicherlecks
    - .NET-Sprachen typsicher, keine Zeiger = kein bösartiger Code
    - Rollenbasierte + Codezugriffssicherheit auf allen Betriebssystemen

Nutzung des .NET Framework (1/4)

Continental Institut für Technologie und Transformation

- .NET-Programme schreiben
  - Framework Class Library (FCL) = Funktionsbibliothek
  - Zugriff auf Windows-API oder COM-Komponenten möglich (aber Sicherheit + Performance leiden)
  - Gleiche Plattform auf allen Systemen

Nutzung des .NET Framework (2/4)

Continental Institut für Technologie und Transformation

- .NET-Programme installieren
  - XCopy-Bereitstellung
  - .EXE, .DLL = Intermediate Language (IL) + Metadaten
    - Plattformunabhängigkeit
    - Disassemblierbar, Problem?
  - Assembly
    - Enthält Beschreibung benötigter Bibliotheken und Rechte
    - Besteht aus einer oder mehreren Dateien
    - Konfigurationsmöglichkeiten unabhängig vom Betriebssystem
    - Global Assembly Cache (GAC)

Nutzung des .NET Framework (3/4)

- .NET-Programme ausführen >
  - Grafik im Kapitel *.NET-Referenz*
  - In Common Language Runtime (CLR) laden
    - Metadaten in Assembly -> Bibliotheken, Rechte
    - Konfigurationsdateien auswerten
  - Just-in-Time kompilieren (JITten)
    - IL -> Betriebssystemspezifischer Code
    - Sichtbarkeit + Typ von Variable bekannt -> Garbage Collection möglich
    - CLR erkennt Programmabsicht -> kann abbrechen
    - Zeiger, z. B. in C# -> unsicherer Code, braucht höchste Ausführungsrechte
    - API-Funktionen, COM-Komponenten → unverwalteter Code, noch schlimmer

Nutzung des .NET Framework (4/4)

Continental Institut für Technologie und Transformation

- .NET-Programme ausführen
  - Anwendungsdomäne
    - Isoliert wie Prozess -> Sicherheit
    - Leichter als Prozess -> Geschwindigkeit, Sparsamkeit
    - Auch auf Betriebssystemen ohne Prozesse möglich
  - CLR-Host
    - Nötige Erweiterung für Programm / Betriebssystem
    - Entscheidet Verhältnis .NET-Programm / Anwendungsdomäne / Prozess
    - MS liefert CLR-Hosts für Windows, IIS und IE
    - Mono-Projekt liefert CLR-Hosts für Windows, FreeBSD, Mac OS X, Linux + für Apache-Webserver

Kapitel 3

KONSOLENANWENDUNGEN

Zweck einer Konsolenanwendung

- Vereinfacht das Programmieren
- Konsole fehlt aber z. B. auf Windows Phone
- Heute meist nur noch für Test- oder Demonstrationsprogramme
  - RAD erleichtert Programmieren grafischer Oberflächen
  - Nutzer sind Besseres gewöhnt
  - Programme für Programmierer

Anatomie einer Konsolenanwendung

- Vorlage Konsolenanwendung in Visual Studio
- *`Program.cs`*, Name änderbar
- Elemente im Projektmappen-Explorer ansehen
- Dateien und Verzeichnisse im Explorer ansehen
- Beschreibung der Dateien und Verzeichnisse siehe Tabelle
- *`Program.cs`*:
  - Namensraum = Projektname
  - Klasse `Program`
  - Methode `Main()`
  - Eigene Felder + Definitionen auf Klassenebene oberhalb `Main()`

Funktionsweise einer Konsolenanwendung

- Start -> `Main()`
- Ende `Main()` -> Ende Programm
- Kommandozeilenargumente in `args` Parameter
  - 1. Parameter nicht Anwendungsname
- `Main(): int` statt `void` möglich
- Kommunikation über `Console`-Klasse
- Standard-Eingabe, -Ausgabe und -Fehlerausgabe umleitbar
  - `StartInfo`-Eigenschaft der `Process`-Klasse
  - `StandardInput`-, `StandardOutput`- und `StandardError`-Eigenschaften

Eingaben auswerten und Text ausgeben

- `Console`-Klasse aus Namensraum System
- Methoden `ReadLine()`, `Read()`, `WriteLine()` und `Write()`
- Lesen in Schleife
- Mit `ReadLine()` auf Eingabe des Benutzers warten

Demo

EINGABEN AUSWERTEN UND TEXT AUSGEBEN

Kommandozeilenparameter auswerten

Continental Institut für Technologie und Transformation

- Kommandozeilenparameter in `Main()` im Parameter `args`
- `args`: String-Array mit d. Leerzeichen getrennten Teilen der Kommandozeile
- Kommandozeilenparameter beim Aufruf aus IDE:
  - Projekt, Eigenschaften, Debuggen, Startoptionen, Befehlszeilenargumente

Demo

KOMMANDOZEILENPARAMETER AUSWERTEN

Features für Konsolenanwendungen in .NET

- Vorder- + Hintergrundfarbe des Textes einstellbar
- Ausgabepuffer (= bestehende Ausgabe) verschieben / löschen
- Position, Größe, Titelleistentext des Konsolenfensters per Code änderbar
- Noch mehr kosmetisches ...

Kapitel 4

PROGRAMMIEREN MIT DEM .NET FRAMEWORK

Warum C#?

- Microsoft beginnt Mitte der 90er Jahre, .NET zu entwickeln
  - Bestehende Sprachen für spezielle Compiler, Bibliotheken und BS entwickelt → passten nicht zu .NET
  - Bestehende Sprache umzuarbeiten rechtlich nur für VB möglich -> Probleme mit alten VB-Programmierern
- Also technische / juristische Notwendigkeiten → C# nichts Neues, Best-of bekannter OO-Programmiersprachen
- C#
  - Features von Delphi durch Anders Hejlsberg
  - Folgt seinem Namen entsprechend C-ähnlicher Syntax
  - Wenig gemeinsam mit Sprache Java; .NET viel gemeinsam mit Plattform Java
  - „ßi-scharp“ gesprochen
  - C# ist um einen Halbton erhöhtes C -> C# Weiterentwicklung von C
  - C# → Syntax möglicher Sprachkonstrukte
  - (.NET → Typsystem + Klassenbibliothek aller Sprachen)

Ein einfaches C#-Programm (1/2)

- Hallo Welt!

```
class Min {
     static void Main() {
          System.Console.WriteLine("Hallo Welt!");
     }
}
```

Ein einfaches C#-Programm (2/2)

- Programm zeigt:
  - Keine globale Funktionen oder Variablen
  - (Nur wenigen Definitionen außerhalb einer Klasse möglich)
  - C# keine Hybridsprache wie C++ oder Delphi
  - Methode `Main()`
    - Außer in DLLs immer nötig
    - Andere Signatur möglich
    - Zugriffsmodifizierer wie `private` und `public` werden ignoriert
    - C# unterscheidet Groß- und Kleinschreibung, `Main()` != `main()`
    - .NET-Klassenbibliothek in Namensräume untergliedert, immer angeben

Besonderheiten von C# (1/2)

- Wenig Überraschungen, nur Details
  - Keine globale Funktionen oder Variablen
  - Keine lokalen statischen Variablen, private Felder benutzen
  - Lokale Variablen werden nicht automatisch initialisiert, nur Felder
  - Groß- und Kleinschreibung wird unterschieden
  - Umlaute in Bezeichnern erlaubt, aber keine Sonderzeichen wie $ oder #
  - Lokale Variablen lassen sich blockweise deklarieren, z. B. in `if`-Block
  - C# definiert Alias-Namen für FCL-Typen, z. B. `int` für `System.Int32`
  - Alles ist ein Objekt, auch einfachere Typen wie Integer
  - C# / .NET Framework unterscheidet zwischen Wert- und Referenztypen
  - Strukturen können Methoden + Ereignisse enthalten + Schnittstellen implementieren
  - Partielle Klassen, Schnittstellen, ...
  - Generische Klassen, Schnittstellen, ...
  - `Nullable` Types (Werttypen, die den Wert `null` annehmen können)
  - Autoimplemented Properties
  - Delegaten

Besonderheiten von C# (2/2)

Continental Institut für Technologie und Transformation

- Sichtbarkeit lokaler Variablen durch Ort der Deklaration bestimmt
- Sichtbarkeit von Feldern durch Zugriffsmodifizierer definiert
  - `public`: Uneingeschränkter Zugriff
  - `protected`: Zugriff innerhalb des Typs und davon abgeleiteter Typen
  - `internal`: Zugriff innerhalb der Assembly
  - `protected internal`: Innerhalb Assembly wie `public`, außerhalb wie `protected`
  - `private`: Zugriff innerhalb des Typs

Wert- und Referenztypen (1/3)

- Werttypen
  - Kleine, kurz benutzte Typen wie Integer, Aufzählungen und Strukturen
  - Schnell zugreifbar
  - Variablen enthalten Wert selber
- Referenztypen
  - Größere, langfristig verwendete Typen wie Klassen und Strings
  - Variablen enthalten nur Verweis auf eigentlichen Wert
  - Garbage Collector, keine Freigabe nötig

Wert- und Referenztypen (2/3)

- Vergleichen von Referenztyp-Variablen
  - Verglichen werden Referenzen, nicht Objekte selber
  - `Equals()` + `ReferenceEquals()`, Vorsicht bei Basisimplementierung
- Zuweisen von Referenztypen an Variablen
  - Kopie der Referenz wird übergeben
  - Kopie der Referenz verweist auf dasselbe Objekt wie Original
  - Veränderungen wirken sich direkt auf das Objekt aus

Wert- und Referenztypen (3/3)

- Zuweisung erzeugt evtl. im Hintergrund aus Werttyp einen Referenztyp
  - Boxing
  - Beim Zugriff dann umgekehrter Weg - Unboxing
  - Beides automatisch, kostet aber Performance
  - Generische Auflistungsklassen
- Strings
  - Sind Referenztypen
  - Verhalten sich aber wie Werttypen
    - Verglichen werden ihre Werte
    - Beim Zuweisen werden sie kopiert
- Strukturen
  - Große Strukturen -> aufwändige Kopieroperationen beim Zuweisen
  - Statt großer Strukturen entsprechende Klassen einsetzen

Demo

ZUWEISEN UND VERGLEICHEN VON WERT- UND REFERENZTYPEN

Demo

AUSWIRKUNGEN DES BOXING

Strings

- Zeichen in .NET
  - Unicode
  - Zeichenketten: System.String, z. B. "Ein String"
  - Einzelne Zeichen: System.Char, z. B. 'c'
- System.String
  - Referenztyp mit Werttyp-Semantik
    - String-Vergleich -> Vergleich der Werte, kein Vergleich der Referenzen
    - Verkettung möglich -> Kopieroperationen, **StringBuilder** benutzen!
  - Compare(), Equals(), StartsWith(), IndexOf()
  - Zugriff auf Zeichen mit Array-Syntax
  - Format() ersetzt Platzhalter in String durch formatierte Werte
  - Zeilenumbruch in String durch \r\n oder Environment.NewLine
  - Rückstrich zugleich Escape-Zeichen
    - string s = "c:\\";
    - string s = @"c:\";

Demo

STRING-VERKETTUNG MIT DER STRINGBUILDER-KLASSE

Typumwandlung und -abfrage

Continental Institut für Technologie und Transformation

- C# / .NET Framework ist typsicher -> Compiler findet viele Fehlerquellen
- Implizite Typumwandlung, z. B. von `Int32` auf `Int64`, automatisch
- Explizite Typumwandlung
  - Harte Typumwandlung, z. B. `(Int32)i` – Laufzeitfehler möglich!
  - Typsichere Umwandlung mit ***`as`*** – ***`null`***, wenn Umwandlung fehlschlägt
- Typabfrage
  - Operator **`is`**, z. B. **`if (i is Int32)`** ...;
  - Type-Objekt
    - Instanz: Methode `GetType()`
    - Typ: `typeof()`-Operator
- Typen wie z. B. `System.Boolean` und `System.Int32` besitzen statische Methode `Parse()` zum Instanziieren aus `String`

Aufzählungen (Enum)

- Aufzählung
  - Satz von Konstanten
  - Typsicherheit
  - Selbstdokumentierender Code
  - `Enum`–Klasse bietet verschiedene Methoden zum Auswerten
  - Als Bitfeld möglich

WEITERES

Arrays

- Arrays
  - Werden zur Aufnahme eines bestimmten Typs deklariert
    - Kein Boxing und Unboxing für Werttypen
    - Aber: generische Auflistungsklassen
  - Mehrdimensionale und unregelmäßige Arrays möglich
  - Keine dynamischen Arrays
  - Unterstützen `foreach`-Schleife
  - Untere Grenze festlegbar, aber lieber bei 0 bleiben
  - Sind Referenztypen
    - Zuweisung kopiert nur Verweis auf Array, nicht dessen Elemente
    - `Copy()`, `CopyTo()` und `Clone()`
    - Trotzdem enthält Array-Kopie Verweise auf **dieselben** Instanzen von Referenztypen
  - Zum Sortieren: `Sort()`, `Reverse()`
  - Zum Suchen: `BinarySearch()`, `IndexOf()`
  - Anzahl der Elemente: `Length`, `GetLength()`
  - `IndexOutOfRangeException`

Ausnahmen

- Laufzeitfehler
  - NET Framework erzeugt Ausnahme
  - Programm in undefiniertem Zustand
- Ausnahmebehandlung >
  - Ohne Behandlung im Programm bricht es die CLR i. A. ab
  - Trotzdem Ausnahme nur behandeln, wenn Programm wieder in definierten Zustand gebracht werden kann
  - Auch behandelte Ausnahmen protokollieren
    - Exception Management Application Block
    - FCL fängt leider viele Ausnahmen ab
- Exception-Klasse
  - Ausnahme ist Instanz von Exception oder abgeleiteter Klasse
  - Exception-Klasse bietet zusätzliche Informationen, z. B. Aufrufliste
  - Von FCL-Ausnahmen eigene Klassen ableitbar, siehe Online-Hilfe

Ausnahmebehandlung

- `try` ... `catch`-Anweisung
  - Gefährdete Anweisungen in `try`-Block fassen
  - Evtl. aufgetretene Ausnahme im folgenden catch-Block verarbeiten
- `catch`-Block
  - Gibt Ausnahme automatisch frei
  - Wird nur ausgeführt, wenn Ausnahme aufgetreten ist
  - Programmausführung geht in Anweisung nach `catch`-Block weiter
  - Typ der zu behandelnden Ausnahme angeben, schließt abgeleitete Typen ein
  - Nicht Exception = generell alle Ausnahmen behandeln
  - Weitere `catch`-Blöcke möglich, spezifische zuerst
- `throw`-Anweisung
  - Ausnahme im `catch`-Block wieder auslösen
    - ursprünglichen Aufrufliste geht verloren -> `InnerException` setzen
  - Ausnahmen generell auslösen, dazu Instanz erzeugen

Ressourcenschutz-Blöcke

- `try` ... `finally`- / `using`- Anweisung
  - Gefährdete Anweisungen in `try`-Block fassen
  - Im `finally`-Block unverwaltete Ressource freigeben
  - `finally`-Block wird immer durchlaufen, auch bei `return`
  - `using`-Anweisung noch einfacher
  - `finally`-Block / `using`-Anweisung gibt Ausnahme nicht frei