HOCHSCHULE HEILBRONN

insystas
Innovative Systems Technologies & Solutions

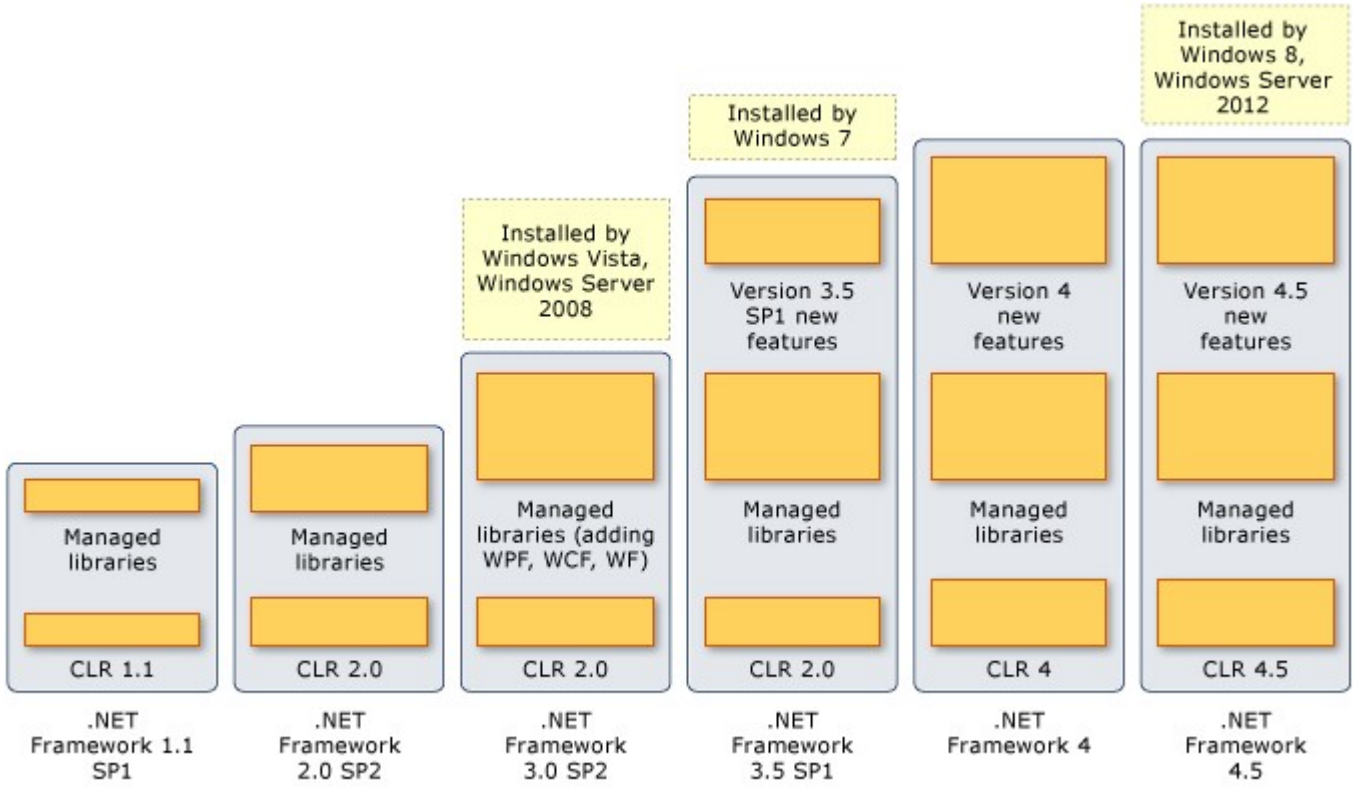
Weiterführende Programmiersprachen

Einführung in die Programmiersprache C#

insystas

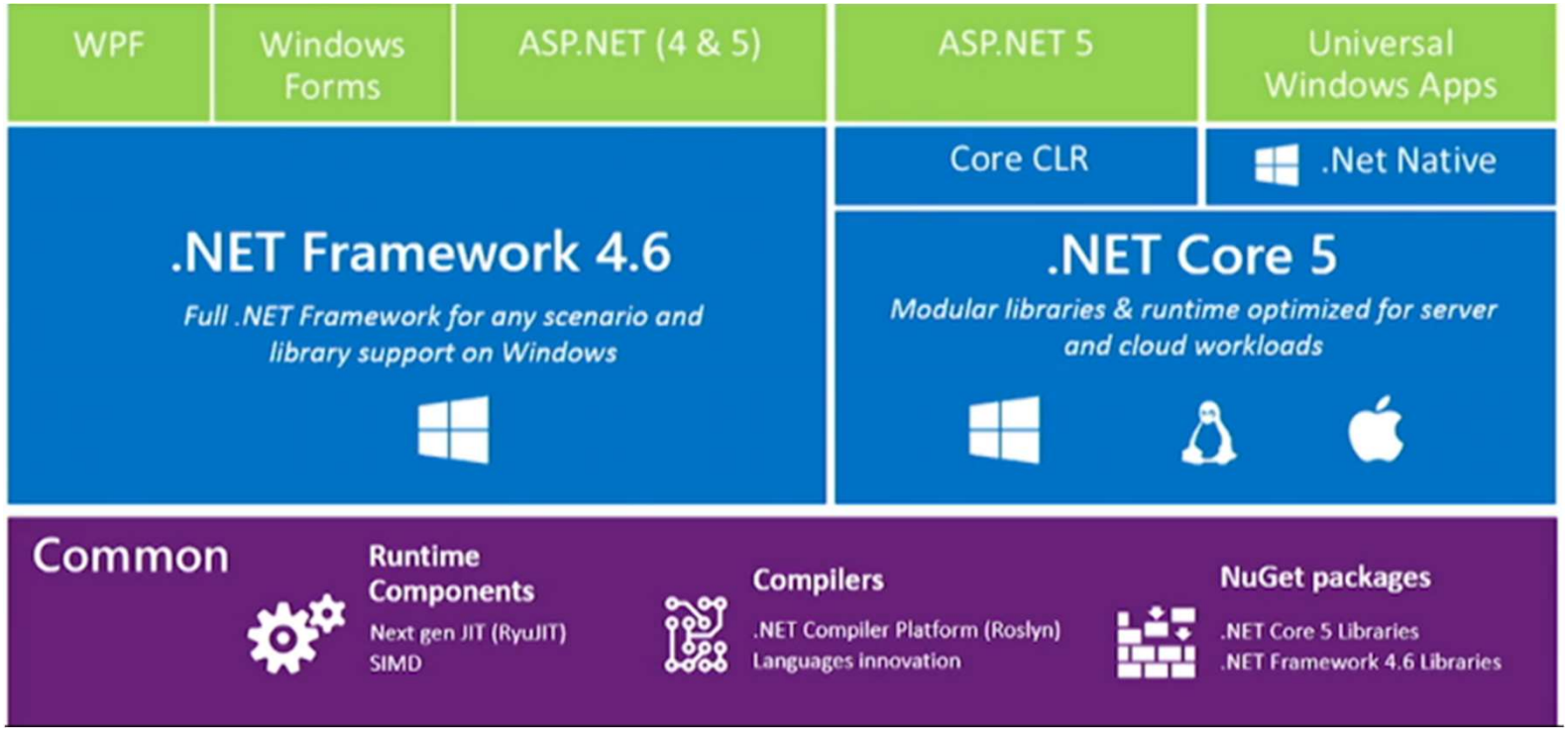
Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Entwicklung Microsoft .NET



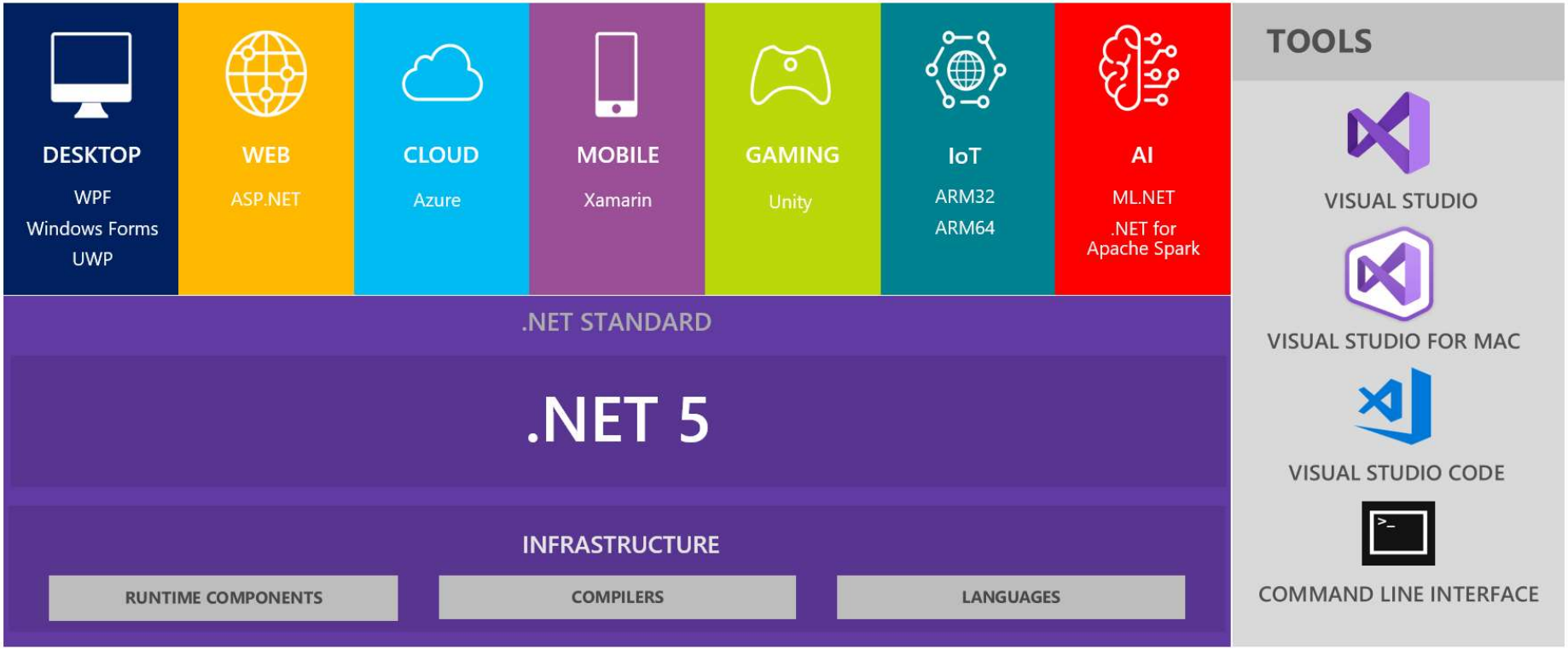
Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Einführung Microsoft .NET



Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

.NET – A unified platform



Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Das .NET Framework

- Das .NET Framework ist eine Softwareplattform
  - Entwicklungsumgebung
    - Visual Studio 2019 Professional e.a. (kommerziell)
    - Visual Studio 2019 Community
      (frei verfügbar, auch kommerziell nutzbar)
  - Klassenbibliotheken
    - Framework Class Library FCL
    - Ausgeliefert als Zwischencode
      (Microsoft Intermediate Language MSIL, IL)
  - Compiler für verschiedene Sprachen
    - C#, Basic
    - Übersetzung in Zwischencode MSIL, der dann von einem JIT Compiler in der Laufzeitumgebung ausgeführt wird
  - Laufzeitumgebung (Common Language Runtime, CLR)

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Verfügbarkeit des .NET Frameworks

- Aktuelle Versionen
  - C# 8.0
  - .NET Framework 4.8 / .NET Core 3.1
  - Visual Studio 2019
- Verfügbarkeit des .NET Framework
  - Einsetzbar ab Windows 98
    - Teilweise eingeschränkter Funktionsumfang
    - Kernbestandteil des Betriebssystems ab Windows Vista
  - Aktuell Code Generierung auch für Linux und MAC OS X
  - Ausführbar unter Linux und MAC OS X: Visual Studio Code

Java und C# - Ähnlichkeiten I

- Objektorientierung
  - Gemeinsame Basisklasse aller Objekte
    - Java: java.lang.Object
    - C#: System.Object
  - Vererbung
    - Ableitung von maximal einer Klasse
    - Implementierung mehrerer Schnittstellen möglich
- Typsicherheit
- Garbage Collection
  - Beide Sprachen verfügen über eine automatische Speicherverwaltung, der Programmierer muss erzeugte Objekte nicht selbst freigeben

Java und C# - Ähnlichkeiten II

- Namensräume
  - Java: Pakete
  - C#: Namensräume (s. auch C++)
- Threads
  - Leichtgewichtige parallele Prozesse
  - Synchronistaion und Kommunikation
- Generizität
- Reflection
  - Zur Laufzeit Zugriff auf Typinformationen
  - Attributierung von Klassen, Methoden, Feldern
    - Java: Annotationen
    - C#: Attribute
- Bibliotheken
  - Object, String, ICollection, Stream

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Übernahmen aus C++

- Überladen von Operatoren
- Zeigerarithmetik in „systemnahen“ Klassen
  - Werden in C# als *unsafe* gekennzeichnet
- Übernahme einiger syntaktischer Details
  - siehe z.B. Vererbung

Java und C# - Unterschiede I

- Referenzparameter
  - In Java Übergabe nur *call by value*
  - C#: auch *call by reference* möglich
    - Realisierbarkeit von Ausgangsparametern
- Objekte auf dem Stack
  - Java: alle Objekte liegen auf dem Heap
  - C#: Instanzierung auch auf dem Stack möglich
    - » Belasten die Garbage Collection nicht
- Einheitliches Typsystem
  - Im Gegensatz zu Java sind in C# alle Datentypen (auch z.B. *int, char*) vom Typ *object* abgeleitet

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

C# - „syntactic sugar“

- Eigenschaften der Sprache, die ihre Mächtigkeit nicht erhöhen, aber einfachere und bequemere Lösungen ermöglichen, als das in anderen Sprachen möglich wäre
  - Properties, Events
    - Properties: spezielle Felder eines Objekts, auf die automatisch mit get- und set-Methoden zugegriffen wird
    - Events: Definition von Ereignissen, die von Komponenten ausgelöst und von anderen behandelt werden
  - Indexer
    - Ein Index-Operator wie bei Array-Zugriffen kann selbst definiert werden
  - Delegates
    - ähnelt den Funktionszeigern in C/C++
  - *foreach*-Schleife
    - bequemes Iterieren über Arrays, Listen, Mengen
  - Iteratoren
  - Lambda-Ausdrücke
  - Query-Ausdrücke
    - SQL-ähnliche Abfragen auf Hauptspeicherdaten (Arrays, Listen)

Übliche Namenskonventionen

Namenskonvention für…	Beispiele	
	Java	C#
Klassen- und Interfacenamen	Car IVehicle	Car IVehicle
Methodennamen	showTree	ShowTree
Packages (Java) bzw. Namensräume (C#)	java.util	System.Windows.Forms
Properties	-	PropertyName
Felder und Variablen	varName redCar	varName redCar

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Namensräume

- Java: Package

```
package meinNamensraum; // erste Zeile der Datei
```

- C#: Namensraum

```
namespace MeinNamensraum
{
   // der Namensraum umschließt die enthaltenen Elemente…
}
```

Verwendung von Bibliotheken

- Bibliothek verfügbar machen
  - Java: Aufnehmen der Klasse /der JAR-Datei zur Umgebungsvariablen *classpath*
  - C#: Referenz auf die Bibliothek zum Projekt hinzufügen (rechter Mausklick auf das Projekt, dann „Verweis hinzufügen…“)
- Namensräume verwenden
  - Java/C#: voll qualifizierten Namen verwenden
    oder:
  - Java: Importieren des Packages
  - C#: Verwenden des Namensraums über *using*

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Konsole: Ein- und Ausgaben

- Ausgabe

```
Console.Write(…) / Console.WriteLn(…)
```

- Eingabe

```
Console.Read(…) / Console.ReadLn(…)
```

Zeilenumbrüche in C#:
…erhalten Sie, wenn Sie die Konstante `System.Environment.NewLine` in einen String oder eine Konsolen-Ausgabe einbauen!

Konvertieren eines Strings in primitive Datentypen

- Der primitive Datentyp *int* entspricht einem Objekt der Klasse *Int32* (alias)

```
Int32.Parse(String s)                          oder
Int32.TryParse(String s, out int result)       oder
Convert.ToInt32(String s)
```

- Analog funktioniert die Konvertierung für andere primitive Datentypen

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Sichtbarkeit in C#, C++ und Java

Java	C#	C++	Erklärung
public	public	public	Zugriff für alle
private	private	private	Zugriff nur innerhalb der jeweiligen Klasse *C#: Default für Felder, Methoden und Properties*
protected			Zugriff nur aus abgeleiteten Klassen und aus Klassen desselben Packages
	protected	protected	Zugriff nur aus abgeleiteten Klassen
<Kein Modifizierer>			Zugriff nur aus Klassen desselben Packages *Java: Default für Klassen, Felder, Methoden und Properties*
	protected internal		Zugriff nur aus abgeleiteten Klassen und aus Klassen derselben Assembly (-> entspricht JAR in Java)
	internal		Zugriff nur innerhalb derselben Assembly *C#: Default für Klassen*

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Die Main-Methode

- Einstiegspunkt in ein startbares C# Programm
  - DLLs haben keinen Einstiegspunkt
    - » Eine evtl. vorhandene Main-Methode wird ignoriert
- Vier mögliche Signaturen:

```
static void Main()                  oder
static int Main()                   oder
static void Main(string[] args)     oder
static int Main(string[] args)
```

- Achtung:
  Die Sichtbarkeit von Main spielt für die Ausführung in C# keine Rolle
  - vgl. Java: *public* static void main(string[] args)
- In einer Klasse darf maximal eine Main-Methode enthalten sein!

Mehrere Main-Methoden

- Mehrere Klassen in einem Package (Java) bzw. einem Assembly (C#) enthalten Main-Methoden
  - Java
    Beim Start der Applikation wird die Klasse übergeben, deren Main-Methode verwendet werden soll:

```
java test.MyClass
```

  - C#
    Die Klasse, deren Main-Methode verwendet werden soll, muss zur Compilierzeit über den Schalter /main bzw. im Visual Studio über die Projekteigenschaften eingestellt werden:
    Menüpunkt *Projekt -> Eigenschaften*
    -> Reiter *Anwendung* -> Auswahl *Startobjekt*

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Automatisierte Generierung der Dokumentation aus dem Quellcode

- Java
  - Eigene Dokumentation:
    ```
    /**
     *
     */
    ```
  - Tool *javadoc* erstellt daraus HTML
  - Dokumentation der Java Klassenbibliothek (JFC)
    Java SDK Dokumentation
- C#
  - Eigene Dokumentation
    ```
    /// <summary>
    ///
    /// </summary>
    ```
  - keine direkte Toolunterstützung zur Weiterverarbeitung durch Microsoft
    aber: freie Tools verfügbar, z.B. *Sandcastle*
  - Dokumentation der .NET Klassenbibliothek (FCL)
    Microsoft Developer Network MSDN

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Formatierung von Textausgaben

- Java:
  - über System.out.format und Platzhalter wie %f, %d, …
- C#:
  - Im String selbst über `String.Format(…)` oder direkt bei der Ausgabe über `Console.Write(…)`
  - {0…}, {1…}, … {n…} dienen als Platzhalter für den 1. bis (n+1). angehängten Wert
  - weiter konfigurierbar:
    - welcher Typ
    - Länge
    - Ausrichtung
    - wie viele Nachkommastellen…
  - Format:

```
…(„Text mit Platzhaltern {0, 4:f2} und {1} bla {2,5} Text“, wert1, wert2, wert3);
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

C# Format-Platzhalter

{ Index [, Ausrichtung][:Format]}

- Index
  Die Position des Wertes in der Parameterkette, der in den Platzhalter eingefügt werden soll (beginnend mit 0)
  - Ein Wert kann mehrfach ausgegeben werden, indem sein Platzhalter mehrmals im formatierten String angegeben wird
  - Zu viele an den formatierten String angehängte Werte werden ignoriert
  - Wird ein Platzhalter für einen Index verwendet, den es in den angehängten Werten nicht gibt, führt das zu einem Laufzeitfehler
- optional: Ausrichtung
  Eine positive (rechtsbündig) oder negative (linksbündig) Ganzzahl, die angibt, wie viele Stellen für die Ausgabe des Platzhalters verwendet werden sollen
- optional: Format
  Ein Buchstabe, optional gefolgt von einer Zahl. Durch das Format wird bestimmt, wie der in den Platzhalter einzufügende Wert zu formatieren ist

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Der Format-Anteil des Platzhalters

- D
  - Decimal
  - Ausgabeformatierung für eine Ganzzahl, ggfs. mit führendem „-“ bei negativen Zahlen
  - Eine (optionale) Zahl dahinter beschreibt, wie viele Stellen für die Zahl mindestens verwendet werden sollen. Eventuell fehlende Stellen werden mit führenden Nullen aufgefüllt
- F
  - Fixed-Point
  - Ausgabe von Fließkommazahlen, ggfs. mit führendem „-“ bei negativen Zahlen
  - Eine (optionale) Zahl dahinter gibt an, wie viele Nachkommastellen ausgegeben werden sollen (-> gerundet!)
- P
  - Percent
  - Ausgabe von Prozentwerten, inklusive des „%“-Zeichens
  - Standard sind 2 Nachkommastellen, diese können aber über eine optionale Zahl hinter dem P auch angegeben werden
- …

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Beispiele zur C# Text-Formatierung

```
Console.WriteLine(„Dieses {0, -10} kostet {1,10:f2} Euro“, „Auto“, 20945);
```

Ausgabe auf der Konsole:

```
Dieses Auto       kostet   20945.00 Euro
```

```
Console.WriteLine(„{0:P} des Bruttoinlandprodukts“, 23.8563);
```

Ausgabe auf der Konsole:

```
23,86 % des Bruttoinlandprodukts
```

```
string formatString = „#{0,D2}: {1}“;
Console.WriteLine(formatString, 1, „VS 2010 herunterladen“);
Console.WriteLine(formatString, 2, „Umgebung einrichten“);
```

Ausgabe auf der Konsole:

```
#01: VS 2010 herunterladen
#02: Umgebung einrichten
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Partielle Klassen

- Aufteilung einer Klasse auf mehrere Quelldateien
- Eingeführt, um von Visual Studio generierten Code vom Entwicklercode zu trennen
  - z.B. Aufteilung einer Klasse *MyForm* in die Dateien *MyForm.cs* (Entwicklercode) und *MyForm.Designer.cs* (von Visual Studio generierter Code
- Die tatsächliche Klasse besteht dann aus der Vereinigungsmenge der n partiellen Klassen
  - Doppelte Deklarationen, widersprüchliche Modifizierer o.ä. Fehler entdeckt der Compiler
- Schlüsselwort *partial*
  - kommt nach dem Modifizierer der Klasse:

```
public partial class AClass { … }
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Werte- und Referenztypen in C#

- Standardmäßige Behandlung lokaler Variablen auf dem Stack:
  - Wertetypen
    - werden direkt auf dem Stack abgelegt
    - dazu gehören in C#
      - „primitive Datentypen“ wie int, double…
      - *struct* und *enum*
  - Referenztypen
    - nur die Referenz liegt auf dem Stack
      - „Inhalte“ werden auf dem Heap abgelegt
    - dazu gehören in C#
      - *string* und Arrays
      - alle übrigen klassenbasierten Objekte
- In Java sind die „primitiven“ Datentypen die einzigen Wertetypen, alles andere sind klassenbasierte Objekte und damit also Referenztypen

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Call By Value

- Die zu übergebenden Parameter werden auf den Stack kopiert
- Die Methode arbeitet mit der Stack-Kopie des Parameters
  - bei Wertetypen: Kopie des Wertes
  - bei Referenztypen: Kopie der Referenz
    Achtung:
    Sowohl die Original-Referenz als auch die Kopie zeigen auf denselben Heap-Bereich, also „auf denselben Inhalt“
    -> Inhalte können verändert werden!!!
- Die Übergabe großer Wertetypen ist daher teuer
  - Rechenzeit: die Daten müssen kopiert werden
  - Speicherplatz: die

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Call By Value

- Die zu übergebenden Parameter werden auf den Stack kopiert
- Die Methode arbeitet mit der Stack-Kopie des Parameters
  - bei Wertetypen: Kopie des Wertes
  - bei Referenztypen: Kopie der Referenz
    Achtung:
    Sowohl die Original-Referenz als auch die Kopie zeigen auf denselben Heap-Bereich, also „auf denselben Inhalt“
    -> Inhalte können verändert werden!!!
- Die Übergabe großer Wertetypen ist daher teuer
  - Rechenzeit: die Daten müssen kopiert werden
  - Speicherplatz: die Daten sind mehrfach vorhanden

!!! Java kennt nur Call By Value !!!

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Call By Reference

- Es wird eine Referenz auf die Parameter übergeben
- Die Methode arbeitet (über eine Indirektion) mit den Originalwerten
  - bei Wertetypen: der Wert selbst
  - bei Referenztypen: die Original-Referenz
- Sehr effiziente Art der Parameterübergabe, aber: die Originalwerte und –Referenzen können verändert werden!

Übergabe von Methodenparametern in C#

- Default: Call By Value
  - Verhalten identisch zu Java
- Call By Reference explizit mit den Schlüsselwörtern *ref* und *out*
  - *ref*
    Übergebener Parameter muss initialisiert sein
  - *out*
    Übergebener Parameter ist i.a. nicht initialisiert und muss in der aufgerufenen Methode initialisiert werden
  - Die Schlüsselwörter müssen sowohl in der Methodensignatur aufgeführt werden als auch beim Aufruf

    Beispiel:

```
public void DoSomething(ref int a) { … }
…
obj.DoSomething(ref eineGanzzahl);
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Properties

- Properties = Eigenschaften
  Es geht also um den gekapselten Zugriff auf die Eigenschaften von Objekten
  - Java
    - Coding Pattern, stammt ursprünglich aus der Java Beans Spezifikation
      - Felder privat machen
      - Zugriff über öffentliche *get* und/oder *set* Methoden
  - C#
    - Integriertes Sprachmittel mit eigener Syntax
    - Ermöglicht komfortablen, aber dennoch gekapselten Zugriff auf Felder

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

C# Properties: Beispiel

```
public class AClass {
  private string descr;
  public string Description {
    get {
      if (desc != null) {
        return desc;
      }
      return String.Empty;
    }
    set { desc = value; }
  }
  public static void Main() {
    // zugriff auf das Property Description wie auf ein Feld
    AClass a = new AClass();
    a.Description = „Beschreibung Instanz a“; // set accessor
    String s = a.Description;                 // get accessor
  }
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

C# Properties: Allgemeine Syntax

```
private Datentyp feld;
[Zugriffsmodifizierer] Datentyp PropertyName {
  [Zugriffsmodifizierer] get {
    // Anweisungen
    return feld;
  }
  [Zugriffsmodifizierer] set {
    // Anweisungen
    feld = value;
  }
}
```

```
[Zugriffsmodifizierer] Datentyp PropertyName {
  [Zugriffsmodifizierer] get; [Zugriffsmodifizierer] set;
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

C# Properties: Weitere Infos

- Propertynamen:
  - beginnen mit einem Großbuchstaben
  - Propertyname und Feldname können übereinstimmen, müssen es aber nicht
- impliziter Parameter *value*
  - Wird beim Aufruf des set-Accessors übergeben
  - *value* hat den Datentyp des Properties

C# Properties: Weitere Infos

- read only/write only Properties
  - es ist möglich, nur *get* oder nur *set* zu definieren
  - Dies gilt nicht für die verkürzte Schreibweise!
- Zugriffsmodifizierer
  - Zugriffsmodifizierer (private|public|protected|internal|internal protected) vor dem Datentyp des Properties:
    - gilt als Default für *get* und *set Zugriffe*
  - Default kann überschrieben werden:
    - Angabe des Zugriffsmodifizierers direkt vor den Accessoren
    - allerdings: ein innerer Modifizierer muss restriktiver sein als der äußere!

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Konstruktoren

- Syntax (Java und C#)

```
[Zugriffsmodifizierer] MyClass ([0..n Parameter]) {
  // Initialisierung etc.
}
```

  - kein Rückgabewert
  - mehrere Konstruktoren möglich

- Verkettung
  - ein Konstruktor der Klasse ruft einen anderen Konstruktor derselben Klasse auf
  - Schlüsselwort: *this*

Java

```
public MyClass() {
  this(„logfile.txt“);
  // sonstige Initialisierungen etc.
}
```

C#

```
public MyClass() : this(„logfile.txt“) {
  // sonstige Initialisierungen etc.
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Destruktoren

Java

- Verwendet die *finalize()*-Methode als eine Art Destruktor
- keine praktische Relevanz in Java, da weder die Reihenfolge der Ausführung, noch die Ausführung selbst garantiert sind!

C#

- Destruktor wird automatisch aufgerufen, wenn der Garbage Collector ein Objekt aus dem Speicher entfernt
  - auch beim Beenden des Programms
- Syntax:

```
~Klassenname() { … }
```

  - kein Zugriffsmodifizierer
  - keine Parameter

- Destruktoren verursachen Mehraufwand bei der Garbage Collection
  - keine leeren Destruktoren erzeugen!

Garbage Collection

- Die Ausführung der Garbage Collection kann erzwungen werden
  - Java: Aufruf von System.gc();
  - C#: Aufruf von System.GC.Collect();
  - Manueller Aufruf sollte aber vermieden werden, da er zu Leistungseinbußen führen kann
  - Relevanz bei Multithreading Anwendungen

Method Overloading
Das Überladen von Methoden

- Mehrere Methoden können denselben Namen haben, müssen sich aber in der Parameterliste unterscheiden
  - Anzahl und/oder Reihenfolge der Parametertypen müssen sich unterscheiden
  - Nicht ausreichend sind:
    - Unterschiede im Rückgabewert
    - Unterschiede bei der Benennung der Parameter

Arrays

- Sammlung von Daten desselben Typs
- Intern sind Arrays Klassen, die sich von *System.Array* ableiten

  Beispiel:

```
int[] myArray = new int[] {2, 1};
if (myArray is Array)
  Console.WriteLine(„myArray ist ein Array!“);
```

- Die statischen Methoden der Klasse *Array* können auf die eigenen Arrays angewendet werden
  - Löschen (Elemente auf Default-Werte zurücksetzen)

```
Array.Clear(myArray, intAb, intAnzahl);
```

  - Kopieren

```
int[] second = (int[]) myArray.Clone();
```

  - Sortieren

```
Array.Sort(myArray);
```

Interface IComparable muss implementiert sein

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Arrays

- Sind Referenztypen
- Erstes Element hat den Index 0
- Jedes in enem Array enthaltene Feld wird automatisch mit dem Default-Wert seines Datentyps initialisiert

Eindimensionale Arrays

- 3 Möglichkeiten zu Deklarieren und Initialisieren

```
int[] a = new int[2];
int a = new int[] {0, 0};
int[] a = {0, 0};
```

- Zugriff über `name[index]`

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Mehrdimensionale Arrays

- Die Dimensionen werden durch ein Komma innerhalb der eckigen Klammen getrennt
- „rechteckige“ Form der Arrays, d.h. die Anzahl der Elemente ist innerhalb einer Dimension für alle Einträge gleich
- Beispiel:
  - 2-dimensionale Arrays für Tabellen
  - 3-dimensionale Arrays für‘s Koordinatensystem
  - …

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Jagged Arrays

- jagged: gezackt / zerklüftet
  - Array, das selbst Arrays enthält
  - im Allgemeinen nicht „reckeckig“
  - können beliebig tief geschachtelt werden
  - Hinweis:
    Auch die Java-Arrays können „jagged“ sein

Jagged Arrays

Deklaration und Initialisierung zweidimensionaler jagged Arrays

```
int[] multi = new int[3][];
multi[0] = new int[3];
multi[1] = new int[2];
multi[2] = new int[7];
```

bzw.

```
int[][] multi = new int[][] { new int[3], new int[2], new int[7] };
int[][] multi =              { new int[3], new int[2], new int[7] };
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

foreach-Schleife

- Zum Durchlaufen aller Elemente einer *Collection*
- vgl. Java:

```
for (Datentyp  element : collectionName)
```

- Schlüsselwörter: *foreach* und *in*
- Voraussetzung:
  Die *Collection* muss das Interface *IEnumerable* implementieren
  - ist bereits gegeben bei
    - allen Arrays
    - den Standard-Collections in den Namespaces *System.Collections* und *System.Collections.Generic*
- Syntax:

```
foreach (Datentyp element in collectionName) {
  // Operation auf element
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Beispiele: foreach-Schleife über Arrays

```
string[] mainzelmaennle
  = new string[] {„Anton“, „Berti“, „Conni“, „Det“, „Edi“, „Fritzchen“ };
foreach (string m in mainzelmaennle) {
  Console.Writeln(m);
}
```

```
int [,] multiArray = new int [,] { {1, 11}, {2, 22} };
foreach (int num in multiArray) {
  Console.WriteLine(num); // iteriert über alle 4 Elemente
}
```

```
int[][] jagged = new int[2][] { new int[5], int[3] };
foreach (int[] innerArray in jagged) {
  Console.Writeln(innerArray); // iteriert nur über äußeres Array
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Exceptions

- Repräsentation eines Ausnahmezustands durch ein Exception-Objekt
- Basisklasse: *System.Exception*
- Die Behandlung von Exceptions wird nicht erzwungen („unchecked“)
  - C# kennt keine *throws*-Klausel!
- Exceptions können…
  - in **eigenen Exception-Klassen** implementiert werden
  - in beliebigen Methoden (und dort an beliebigen Stellen) **ausgelöst** werden
  - im *catch*-Block **behandelt** werden
- Beispiele einiger vordefinierter Exception-Klassen:
  - *System.IO.IOException*: Ausnahme im Umfeld einer Dateioperation
  - *System.IndexOutOfRangeException*: falls Sie versuchen, auf ein Feld eines Arrays zuzugreifen, das nicht existiert (z.B. Index zu groß)
  - *System.InvalidCastException*: bei ungültigen Typumwandlungen

Auslösen und Behandeln von Exceptions

- Auslösen einer Exception

  throw new XYZException();

- Behandeln einer Exception

```
try {
        // kritischer Code, der eine Exception
        // auslösen könnte
}
catch (<ExceptionKlasse <Variablenname>) {
        // Behandeln der Ausnahme und/oder
        // ergänzen und wieder auslösen
}
[finally {
        // Nacharbeiten; werden (wenn vorhanden)
        // auf jeden Fall ausgeführt!
}]
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Exception Handling: der *try*-Block

- Umschließen Sie nicht zu viele Anweisungen in einem *try*-Block!
  Beispiel:

```
try {
        for (int i=-5; i<=5; i++) {
                Console.WriteLine((aNumber/i).ToString());
        }
}
catch (DivideByZeroException ex) …
```

  Bei einer Division durch 0 wird die ganze Schleife abgebrochen!

  besser:

```
for (int i=-5; i<=5;i++) {
        try {
                Console. WriteLine((aNumber/i).ToString())
        }
        catch (DivideByZeroException ex) …
```

  Bei einer Division durch 0 wird nur der fehlerhafte Aufruf abgebrochen.

Exception-Handling: Der *catch*-Block

- Es können mehrere *catch*-Blöcke nacheinander folgen
  - beachten Sie aber, dass speziellere Exceptions vor den allgemeineren stehen müssen, da der erste passende *catch*-Block die Exception behandelt
  - Nach dem Abarbeiten eines *catch*-Blocks ist die Exception nicht mehr vorhanden, es sei denn, sie wird mit *throw* erneut ausgelöst
- Syntax, falls *System.Exception* in einem *catch*-Block bearbeitet werden soll:

```
…
catch (Exception ex) {
        // Behandlung der Ausnahme
}
…
catch {
        // Behandlung der Ausnahme
        // kein Exceptionname ➔ kein Zugriff auf die Exception
}
```

- Weiterwerfen einer Exception:

```
… catch (Exception ex) { throw ex; }
… catch { throw; }
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Exception-Handling: der *finally*-Block

- ist optional
- dient zum Aufräumen der „Überbleibsel“
  - z.B. Schließen einer Datei, Löschen temporärer Dateien…
- wird auf jeden Fall ausgeführt:
  - wenn *keine* Exception auftritt
  - wenn *eine behandelte* Exception auftritt
  - wenn *eine unbehandelte* Exception auftritt

Überladen von Operatoren

- Operatoren
  - unäre: + - ! ~ ++ --
  - binäre: + - * / % & | ^ >> <<
  - relationale: < > <= >= == !=
- Überladen, um sie für eigene Klassen verwenden zu können, z.B.
  - myObj++;
  - myObj3 = myObj1 + myObj2;
  - if (myObj1 > myObj2) { …
- Beispiele für sinnvolle Operatoren-Überladungen
  - + bei Zeichenketten („Konkatenation“)
  - + - * / bei Klassen, die andere Zahlensysteme repräsentieren (Dualzahlen, römische Zahlen, Vektoren, Matritzen)

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Überladen von Operatoren

- Überladen ist auch mehrfach für unterschiedliche Übergabeparameter möglich!
  - z.B. Multiplikation bei Vektoren
    Vector v2 = v1 * 7; // … operator* (Vector, int)
    Vector v3 = v1 * v2; // … operator* (Vector, Vector);
  - Operatoren nur überladen, wenn semantisch erschließbar!
    - Was bedeutet es, wenn eine Bilder-Klasse den Operator „-“ überlädt? ➔ nicht sinnvoll!
  - Bei manchen Operatoren wird ein paarweises Überladen erzwungen:
    == und !=
    < und >
    <= und >=

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Überladen von Operatoren: Syntax

- Schlüsselwort *operator*
- Operatorüberladungen sind
  - immer *static*
  - in der Regel *public*
- Wählen Sie einen sinnvollen Typ für den Rückgabewert
  - z.B. für + und – Rückgabewert vom Typ Ihrer Klasse
  - <= und >=, != und == Rückgabewert *bool*
- Parameter
  - bei unären Operatoren: 1 Parameter vom Typ der eigenen Klasse
  - binäre und relationale Operatoren: 2 Parameter
    - mind. der erste muss vom Typ der eigenen Klasse sein
    - 1. Parameter entspricht dem Objekt vor dem Operator

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Einschub: Implizite und explizite Konvertierung

- 2 Arten von Konvertierung
  - **Implizite** Konvertierung: automatisch
    z.B. *short* nach *int*:
    short s = 7;
    int i = s;
  - **Explizite** Konvertierung: muss im Code explizit durchgeführt werden
    z.B. *int* nach *short*
    int i = 7;
    short s = (short) i;

Operator Overloading
Konvertierungsbeispiele

```
class HexNumber {
    ...
    public static implicit operator int(HexNumber hex) {
        return hex.num;
    }
    public static implicit operator long(HexNumber hex) {
        return hex.num;
    }
    public static implicit operator short(HexNumber hex) {
        return hex.num;
    }
    public static explicit operator string(HexNumber hex) {
        return hex.ToString();
    }
}

static void Main() {
    HexNumber h1 = new HexNumber(0x3F87);
    ...
    string str = (string)h1;
    int i = h1;
    Console.WriteLine("HexNumber h1 = {0} (= {1} als int)", str, i);
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Vererbung

- Klassen können in C# genauso wie in Java
  - von genau einer Basisklasse erben
  - beliebig viele Interfaces implementieren

Vererbungssyntax

- Java
  class DerivedClass extends BaseClass
  implements Interface1, Interface2
  - Java unterscheidet zwischen Klassenvererbung (*extends*) und Schnittstellenvererbung (*implements*)
  - Werden mehrere Schnittstellen implementiert, so werden diese durch Komma getrennt
- C#
  class DerivedClass : BaseClass,
  Interface1, Interface2
  - C# unterscheidet syntaktisch nicht zwischen Klassen- und Schnittstellenvererbeung
    - Basisklasse und/oder Schnittstellen werden mit „:“ an die Klassendefinition angehängt
    - Basislasse und (ggfs. mehrere) Schnittstelle(n) werden jeweils durch Komma getrennt

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Polymorphie („Vielgestaltigkeit“)

- Ein Objekt einer abgeleiteten Klasse kann immer in einer Referenz vom Typ der Basisklasse bzw. vom Typ einer geerbten Schnittstelle gespeichert werden
  - implizierter „Upcast“
  - Beispiel:
    BaseClass base = new DerivedClass();
- Um das Objekt nun wieder in einer Referenz des Typs der abgeleiteten Klasse zu speichern, muss exlpizit gecastet werden
  - explizierter „Downcast“
  - Beispiel:
    DerivedClass derived = (DerivedClass) base;
  - Zuvor sollte allerdings getestet werden, ob es sich bei dem Objekt tatsächlich um ein Objekt der abgeleiteten Klasse handelt!
    ➔ Laufzeitfehler vermeiden!

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Einschub: Typprüfung und Casts

- Java
  - Schlüsselwort zur Typprüfung: *instanceof*
  - Beispiel:
    ```
    DerivedClass derived = null;
    if (base instanceof DerivedClass) {
      derived = (DerivedClass) base;
    }
    ```
- C#
  - Schlüsselwort zur Typprüfung: *is*
  - Alternative zum Cast: Schlüsselwort *as*
    - liefert *null* (keine Exception!), falls der CAst nicht möglich ist
  - Beispiel:
    ```
    DerivedClass derived = null;
    if (base is DerivedClass) {
      derived = (DerivedClass) base;
    }
    DerivedClass2 d2 = base as DerivedClass;
    ```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Aufruf von Basisklassen-Methoden und -Konstruktoren

- Aufruf von Basisklassen-Methoden
  - Java: super.*Methodenname*(…);
  - C#: base.*Methodenname*(…);
- Default-Konstruktoren einer Basisklasse … werden automatisch beim Erzeugen der abgeleiteten Instanz aufgerufen, falls …
  - die abgeleitete Klasse nicht explizit einen anderen Basisklassen-Konstruktor aufruft
  - der Default-Konstruktor vorhanden und von der abgeleiteten Klasse aus sichtbar ist

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Aufruf von Basisklassen-Konstruktoren

- Sonstige Konstruktoren
  - Java
    - Schlüsselwort: *super*
    - Aufruf muss in der 1. Zeile des Konstruktors stehen
    - Beispiel:
      ```
      public DerivedClass (int id) {
              super(id);
              // sonstige Initialisierungen
      }
      ```
  - C#
    - Schlüsselwort: *base*
    - Aufruf wird, durch „:“ getrennt, an die Deklaration angehängt
    - Beispiel:
      ```
      public DerivedClass (int id) : base(id) {
              // sonstige Initialisierungen
      }
      ```

Aufruf von Basisklassen-Destruktoren

- Java
  - verwendet die *finalize()* Methode als eine Art Destruktor
    - explizierter Aufruf des Basisklassen-Finalisierer aus der eigenen *finalize()* Methode heraus
    - *aber: Keine praktische Relevanz in Java*
- C#
  - Nach dem Abarbeiten eines Destruktors wird automatisch der Destruktor der Basisklasse aufgerufen
    ➔ kaskadiertes Abarbeiten entlang der Vererbungskette

Überschreiben geerbter Methoden

- In Vererbungshierarchien ist es oft sinnvoll, eine Methode der Basisklasse durch eine speziellere Variante in der abgeleiteten Klasse zu überschreiben
- Zur Laufzeit soll dann die spezialisierte Variante der Methode aufgerufen werden, d.h.
  - die Methode, die in der Objektklasse selbst definiert ist oder
  - diejenige, die im Vererbungszweig am weitesten „unten“ liegt

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Überschreibbare Methoden in Java

- Ob eine Methode einer Basisklasse überschreibbar ist oder nicht, wird bei deren Deklaration entschieden:
  - Methoden sind per Default *virtuell*, also überschreibbar
  - Verbieten des Überschreibens: Schlüsselwort *final*
- In der abgeleiteten Klasse kann die virtuelle Methode dann überschrieben werden (muss aber nicht!)
  - Keine besondere Syntax nötig

Überschreibbare Methoden in C#

- Deklaration von Methoden und Properties der Basisklasse
  - Methoden und Properties sind per Default *nicht virtuell* (nicht überschreibbar)
  - Erlauben des Überschreibens durch Schlüsselwort *virtual*
  - Beispiel:

    public virtual string ToString() { … } // überschreibbar

- Überschreiben einer virtuellen Methode in einer abgeleiteten Klasse
  - benötigt das Schlüsselwort *override*
  - Die überschreibende Methode/das überschreibende Property ist nach unten hin in der Vererbungshierarchie ebenfalls virtuell (implizit)
  - Beispiel:

    public override string ToString() { … }

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Laufzeitverhalten virtueller Methoden und Properties

- Auszuführender Code wird zur Laufzeit bestimmt
  (anhand des Typs des Objekts, nicht der Referenz!)
- Es wird jeweils die am stärksten spezialisierte Variante ausgewählt
  (so weit unten wie möglich in den Vererbungshierarchie)
- Beispiel:
  ToString() ist eine virtuelle Methode!

  ```
  Object[] obj = { new Object(), „Hallo“ };
  Console.WriteLine(„obj[0]={0}, obj[1]={1}“, obj[0], obj[1]);
  ```

  Ausgabe:
  obj[0] = System.Object, obj[1] = Hallo

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Verdecken von Methoden und Properties

- Eine Methode (analog: ein Property) der Basisklasse kann in C# durch eine Methode der abgeleiteten Klasse verdeckt werden
  - Explizit durch Verwenden des Schlüsselworts *new*
  - Implizit, indem man beim Überschreiben in der abgeleiteten Klasse das Schlüsselwort *override* weglässt/vergisst
    - Warnung des Compilers, um auf evtl. unbeabsichtigtes Verdecken hijnzuweisen
    - Implizit *new*
  - Bedeutung des Verdeckens
    - Die Methode in der abgeleiteten Klasse stellt eine vollkommen andere Methode dar, die nur zufälligerweise dieselbe Signatur hat wie die Methode der Basisklasse. Dabei ist es egal, ob die Methode in der Basisklasse virtuell ist oder nicht.
    - Beispiel: Verdecken der ToString()-Methode:

      ```
      public string ToString() { … } bzw.
      public new string ToString() { … }
      ```

- **In Java gibt es kein Verdecken!**
  Nicht-virtuell (== final) ➔ Compiler-Fehler beim Versuch, eine finale Methode zu überschreiben

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Abstrakte Methoden und Properties

- Schlüsselwort *abstract*
- Abstrakte Elemente erzwingen ihr Überschreiben
- Abstrakte Properties
  - Deklaration, aber keine Implementierung der *get*- und/oder *set*-Accessoren
  - Beispiel:
    public abstract string Description { get; set; }
- Abstrakte Methoden
  - nur Deklaration, haben aber keinen Methodenrumpf
  - Beispiel:
    public abstract void Show();
- Sobald eine Klasse mind. ein abstraktes Element enthält, muss sie ebenfalls als abstrakt deklariert werden!
- Eine abstrakte Klasse muss aber nicht zwangsläufig abstrakte Elemente enthalten

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Abstrakte Klassen

- Schlüsselwörter *abstract* und *class*
- Von abstrakten Klassen können keine Objekte angelegt werden
- Objekte sind nur für (nicht abstrakte) abgeleitete klassen möglich, die alle abstrakten Methoden und Properties implementieren
  - auch hier: Schlüsselwort *override* verwenden!
- Können folgende Elemente enthalten
  - abstrakte Methoden
  - implementierte Methoden (Überschreiben nicht erforderlich!)
  - Felder
  - abstrakte Properties
  - implementierte Properties (Überschreiben nicht erforderlich!)

Interfaces

- Schlüsselwort *interface*
- alle Methoden sind implizit *public*
  - müssen auch *public* implementiert werden
- Daten sind nur in folgender Form erlaubt:
  - Java: nur Felder, die *final* sind
  - C#: nur Properties, die keine Implementierung haben
- Interface-Vererbung
  - Interfaces erben andere Interfaces
  - können nicht von Klassen erben
- Verwenden Sie Interfaces anstelle abstrakter Klassen, die ausschließlich abstrakte Methoden enthalten

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Unveränderliche Methoden und Klassen

- Java
  - Schlüsselwort *final*
- C#
  - Schlüsselwort *sealed*

Eine unveränderliche Klasse kann nicht als Basisklasse verwendet werden ➔ Ableitung nicht möglich!

Delegates

- Delegate = Referenz auf Methode
  - verweist auf *Programmcode* im Hauptspeicher
  - im Gegensatz dazu: Objektreferenz (-> Verweis auf Daten)
  - „typisierter Funktionszeiger“ (vgl. Funktionszeiger in C/C#)
- Ist ein Typ!
  - vgl. andere Typen wie Klassen Strukturen, Aufzählungen, Interfaces…
  - Definition direkt im Namespace möglich
  - Schlüsselwort *delegate*
  - erbt von *System.Delegate*
- Ähnliche Vorgehensweise wie im Kontext von Klassen und Objekten

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Delegate -Beispiel

```
delegate int DelegateRechne(int i1, int i2);

class DelegateExample
{
    static int Add(int i1, int i2)
    {
        return i1 + i2;
    }
    static int Sub(int i1, int i2)
    {
        return i1 - i2;
    }
    public static void TestDelegate()
    {
        DelegateRechne rechne = new DelegateRechne(Add);
        int result = rechne(10, 20);
        Console.WriteLine(result);
    }
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Multicast Delegates

- Ein Delegate kann auch mehrere Methoden enthalten
  - man spricht dann von einem „Multicast Delegate“
  - Hinzufügen von Methoden zum Delegate:
    - 1. Methode mit dem Zuweisungsoperator =
      DelegateRechne rechne = new DelegateRechne(Add);
      oder
      DelegateRechne rechne = Add;
    - Jede weitere Methode mit dem überladenen Operator +=
      rechne += Sub;
  - entfernen von Methoden vom Delegate mit dem überladenen Operator -=
    - rechne -= Add;
  - Beim Aufruf des Delegates werden dann alle enthaltenen Methoden nacheinander ausgeführt
  - Nur die zuletzt ausgeführte Methode liefert ihren Rückgabewert zurück, alle anderen Rückgabewerte gehen verloren
    - daher haben Delegates häufig den Rückgabewert *void*

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Multicast Delegate Beispiel

```
delegate int DelegateRechne(int i1, int i2);

class DelegateExample
{
    static int Add(int i1, int i2)
    {
        return i1 + i2;
    }
    static int Sub(int i1, int i2)
    {
        return i1 - i2;
    }
    public static void TestDelegate()
    {
        DelegateRechne rechneMulti = Add;
        rechneMulti += Sub;
        int result = rechneMulti(10, 20);
        Console.WriteLine(result);
    }
}
```


Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Anwendung von Delegates

- Hauptanwendungsgebiet: EventHandler-Methoden im GUI-Umfeld
  - **Java:** wird über Listener Interfaces gelöst
    - die Klasse, die im Ereignisfall benachrichtigt werden möchte, muss das passende Listener Interface implementieren
  - **C#:** wird über Events gelöst
    - basieren auf Delegates
    - siehe Folgevorlesungen

typeof

- Operator zur Ermittlung von Typinformationen zur Laufzeit
  - kann nicht überschrieben werden
  - nimmt einen Parameter (den zu untersuchenden Typ)
  - Beispiel:
    System.Type type = typeof(Demo2.Book);
- Für Instanzen wird typeof nicht benötigt
  - dort wird die von *System.Object* geerbte Methode *getType()* verwendet
  - Beispiel:
    System.Type type = myBook.GetType();
- System.Type
  - Liefert Informationen zu einem Typen
    - getFields()
    - getConstructors()
    - getMethods()
    - getProperties()
    - …

Benutzerdefinierte Wertetypen

- Können vom Entwickler zusätzlich zu den vordefinierten C# Wertetypen wie int, float, double, bool, … angelegt werden
- Dabei handelt es sich um
  - Aufzählungen (*enum*)
  - Strukturen (*struct*)

Aufzählungen (*enum*)

- Eine Aufzählung erzeugt eine Menge benutzerdefinierter Konstanten und gruppiert sie zu einem Typen
- Ein Beispiel:
  - Annahme:
    Ein Autohersteller verkauft seine Autos nur in 3 Farben silber, schwarz und weiß. Der Aufzählungstyp *Autofarbe* hätte dann die Ausprägungen *silber, schwarz, weiß*
  - Definition der Aufzählung:
    public enum Autofarbe { Silber, Schwarz, Weiss }
  - Verwendung der Aufzählung:
    Auto auto = new Auto();
    auto.Farbe = Autofarbe.Silber;

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Aufzählungen
Unterschiede zwischen Java und C#

- Java Aufzählungen
  - sind Referenztypen
  - sind typsicher
  - können beliebige Werte repräsentieren, nicht nur Zahlen
  - sind intern von der Basisklasse *java.lang.Enum* abgeleitet und können
    - Methoden und Attribute haben
    - Interfaces implementieren
- C# Aufzählungen
  - sind Wertetypen
  - sind nicht zu 100% typsicher
  - repräsentieren Zahlenwerte
  - sind intern von der Basisklasse *System.Enum* abgeleitet, können aber nicht erweitert werden (sind unveränderlich)

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Verwendung von Aufzählungen

- Schlüsselwort *enum*
- erzeugt einen neuen Datentyp (Wertetyp)
- kann außerhalb einer Klasse (global) definiert werden
- Intern steht jedes Element einer Aufzählung für einen ganzzahligen Wert
  - statt des *enum* kann nicht einfach eine Ganzzahl verwendet werden ➔ Compilerfehler
  - Es ist aber möglich, eine Zahl in einen *enum* durch explizites Casten umzuwandeln und umgekehrt

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Strukturen

- sind Wertetypen, die eine kleine Gruppe von Variablen zusammenfassen
- alle Wertetypen des .NET Frameworks sind intern ebenfalls als Strukturen implementiert
- Strukturen sind Klassen sehr ähnlich…
  - sie können Attribute, Methoden und Properties enthalten
  - sie können Interfaces implementieren

Strukturen – Unterschiede zu Klassen

- Schlüsselwort *struct*
- all Strukturen erben direkt von der Klasse *System.ValueType* ➔ keine eigene Basisklasse möglich
- von Strukturen kann nicht geerbt werden
  - sie können also nicht abstrakt sein
- Strukturen besitzen immer einen Default-Konstruktor, der nicht überschrieben oder verändert werden kann
  - verschwindet auch dann nicht, wenn weitere Konstruktoren mit Parametern codiert werden
  - daraus ergibt sich in eigenen Konstruktoren ein ungewohntes Verhalten:
    Bevor Sie z.B. einem Property einen Wert zuweisen können, muss es zuerst über den Default-Konstruktor initialisiert werden
    ➔ Default-Konstruktor muss explizit vorverkettet werden!
- Strukturen haben keinen Destruktor

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Initialisieren von Strukturen

- 3 Möglichkeiten
  - Default-Konstruktor:
    Coordinate pointA = new Coordinate();
  - Sonstiger benutzerdefinierter Konstruktor
    Coordinate pointB = new Coordinate(2,4);
  - ohne Initialisierung:
    Coordinate pointC;
    // selbst initialisieren
    pointC.x = 2;
    pointC.y = 4;
    - funktioniert nur für Strukturen!
      - und nur, wenn die Struktur keine Properties besitzt
      - nicht bei „normalen“ Klassen möglich!

Indizierer (Indexer Properties)

- Indizierer werden meist dann eingesetzt, wenn eine KLasse mehrere Objekte gleichen Typs verwaltet
  - einzelne Elemente sind über [] zugreifbar
- Ist eine Art Property
  - aber: Schlüsselwort *this* anstelle eines Bezeichners
  - Implementiert die Accessoren *get/set*
  - haben einen Typ (dient hier als eine Art Rückgabewert)
  - haben zusätzlich noch Parameter, in eckigen Klammern []
- Parameter der Indizierer können von einem beliebigen Typ sein
  - nicht zwangsweise Integer!
  - auch mehrere Parameter möglich (vgl. n-dim. Array)
- pro Klasse können auch mehrere Indizierer mit unterschiedlichen Parametern implementiert werden
- Syntax:
  [Zugriffsmodifizierer] Typ this[1 .. n Parameter] {
  get { /* Implementierung; return Instanz o.g. Typs */ }
  set { /* Implementierung; value hat o.g. Typ */ }
  }

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Das Schlüsselwort *static*

- funktioniert in weiten Teilen genau wie in Java
- bei Klassen
  - die Klasse enthält nur statische Methoden, Properties und Attribute (müssen explizit als *static* gekennzeichnet werden!)
  - In Java kann eine Klasse nicht *static* sein!
- bei Methoden und Properties
  - die Methode (analog: das Property) kann ohne Instanz der Klasse aufgerufen werden
  - Zugriff über *Klassenname.Methodenname*
  - hat nur Zugriff auf andere statische Elemente der Klasse
- bei Attributen
  - statische Attribute sind nur genau einmal pro Klasse vorhanden (nicht pro Objekt!)
  - Zugriff über *Klassenname.Attributname*

Dipl.-Ing. Thomas Marczinkowsky - thomas.marczinkowsky@insystas.de

Konstanten in C#

- Konstanten
  der Wert einer Konstanten kann, nachdem er einmal zugewiesen wurde, nicht mehr geändert werden
- 3 Arten von Konstanten
- (1) Deklaration mit dem Schlüsselwort *const*
  - sind implizit *static*
    (ohne dass Sie das Schlüsselwort *static* angeben müssen)
  - Verarbeitung zur Compilezeit
  - Zuweisung des Werts nur direkt bei der Deklaration möglich
  - Beispiel:
    const double PI = 3.14159265;

Konstanten in C#

- (2) Deklaration mit dem Schlüsselwort *readonly*
  - Verarbeitung zur Laufzeit
  - Zuweisung des Werts
    - entweder bei der Deklaration
    - oder im Konstruktor
- (3) Deklaration mit dem Schlüsselwort *enu*m
  - der Aufzählungstyp *enum* definiert ebenfalls Konstanten!
    (enum = Gruppen von Konstanten)