

DEPARTMENT WIRTSCHAFTSINFORMATIK

FACHBEREICH WIRTSCHAFTSWISSENSCHAFT

Programmieren für Wirtschaftswissenschaftler

SS 2015

Lucian Ionescu

Blockveranstaltung 16.03.-27.03.2015

1. Einführung

Agenda

- **Organisatorisches**
- Einführung in die Welt der (Programmier-)Sprachen
- Die .NET-Umgebung und Visual Studio
- Das erste Programm

Professur für Wirtschaftsinformatik

Prof. Dr. Natalia Kliewer

Lehrstuhlinhaberin

Natalia.Kliewer@fu-berlin.de

Sprechstunde: Donnerstag 10 – 11 Uhr
Raum 217



Dipl.-Wirt.-Inf. Lucian Ionescu

Wissenschaftlicher Mitarbeiter

Lucian.Ionescu@fu-berlin.de

Sprechstunde: nach Vereinbarung
Raum 215



Lena Wolbeck

Wissenschaftliche Mitarbeiterin

[lena.wolbeck@fu-berlin.de](mailto:lana.wolbeck@fu-berlin.de)

Sprechstunde: Dienstag 14 – 15 Uhr
(mit vorheriger Anmeldung)
Raum 213a



Martin Voges

Studentische Hilfskraft

ktd@zedat.fu-berlin.de

Sprechstunde: nach Vereinbarung
Raum 218



www.wiwiss.fu-berlin.de/kliewer

Ablauf

Mo. 16. März – Fr. 20. März	jeweils 10.00 – 12.30 Uhr Vorlesung (K005) 13.30 – 16.00 Uhr Präsenzübung (Pool 1)	
Mo. 23. März – Fr. 27. März	Projekt-/Hausaufgabenphase (Pool 1) Kernzeiten jeweils von 10.00 – 16.00 Gruppengröße 2-4 Personen	
Fr. 27. März	Präsentationen 12:00 – 14:00	

- **Anwesenheitspflicht** bei allen Terminen!
- Informationen zu Prüfungsleistungen im ersten Foliensatz (Organisatorisches)

Ablauf (2)

- In der Vorlesung werden die Inhalte zunächst theoretisch besprochen
 - Vermittlung der **Grundlagen**
 - **Gemeinsame Anwendung** der Konzepte und Klärung von Verständnisfragen
 - Folien zur jeweiligen Veranstaltung werden zuvor auf der Homepage veröffentlicht:

http://www.wiwiss.fu-berlin.de/fachbereich/bwl/pwo/kliewer/lehre/aktuelle_lehrveranstaltungen_ss15/programmierkurs-c-sharp.html

- In den Tutorien werden Aufgaben zu den besprochenen Themen bearbeitet
 - Vertiefung und **eigenständige Anwendung**
 - In **Zusammenarbeit** in Zweier- oder Dreier-Gruppen
 - Unterstützung bei Fragen
 - **Vorstellung der Ergebnisse** von Teilaufgaben
 - Aufgaben des jeweiligen Tages findet Ihr ebenfalls auf der Homepage

Ablauf (3)

- Was wir von euch erwarten:
 - Bereitschaft, sich eine **algorithmische Denkweise** anzueignen
 - Motivation, eine **neue Sprache** zu lernen
- Was ihr von uns bekommt:
 - **theoretische Grundlagen** und angeleitete **praktische Übungen**
 - meldet euch bei Verständnisproblemen (später ist damit niemandem geholfen)
- Was ihr nach dem Kurs können sollt:
 - für beliebige Fragestellungen **eigenständig** Programmlösungen erstellen können
 - **grundlegendes Verständnis** besitzen, mit dem ihr euch weiterführende Kenntnisse eigenständig aneignen könnt
- Programmieren lernt man nur durch **Anwenden!**

Projektarbeiten

- **Selbständige Bearbeitung** von kleinen Programmierprojekten im Rahmen von Gruppenarbeiten
- **2-4 Teilnehmer** pro Gruppe (je nach Kursgröße)
- Wir stehen bereit zur Besprechung von **Zwischenständen** und offener Fragen
- **Abschlusspräsentationen** am Ende der Projektphase



Inhalte

Grundlagen

Erstes Programm

Ein- und Ausgabe

Objektorientierung

Variablen

Wie speichert man
Werte zur späteren
Verwendung ab?

Operatoren

Wie verarbeitet und
verknüpft man
Werte?

Verzweigungen und Schleifen

Wie bildet man
Bedingungen und
Wiederholungen ab?

Erweiterungen

Datenstrukturen

Wie speichert man
gleichartige Elemente
strukturiert ab?

Methoden und Funktionen

Wie unterteilt man
ein Programm in
Arbeitsschritte?

Später: Wie bildet
man das Verhalten
von Objekten ab?

Klassen

Wie kann man die
reale Situation
modellieren?

Wie bildet man
Objekte und deren
Interaktionen ab?

Grafische Oberflächen

*Wie erstellt man eine
grafische Oberfläche?*

Agenda

- Organisatorisches
- **Einführung in die Welt der (Programmier-)Sprachen**
 - Hardware und Software
 - Algorithmen, Syntax und Semantik
 - Algorithmisches Denken, Abstraktion und Objektorientierung
- .NET-Umgebung und Visual Studio
- Das erste Programm

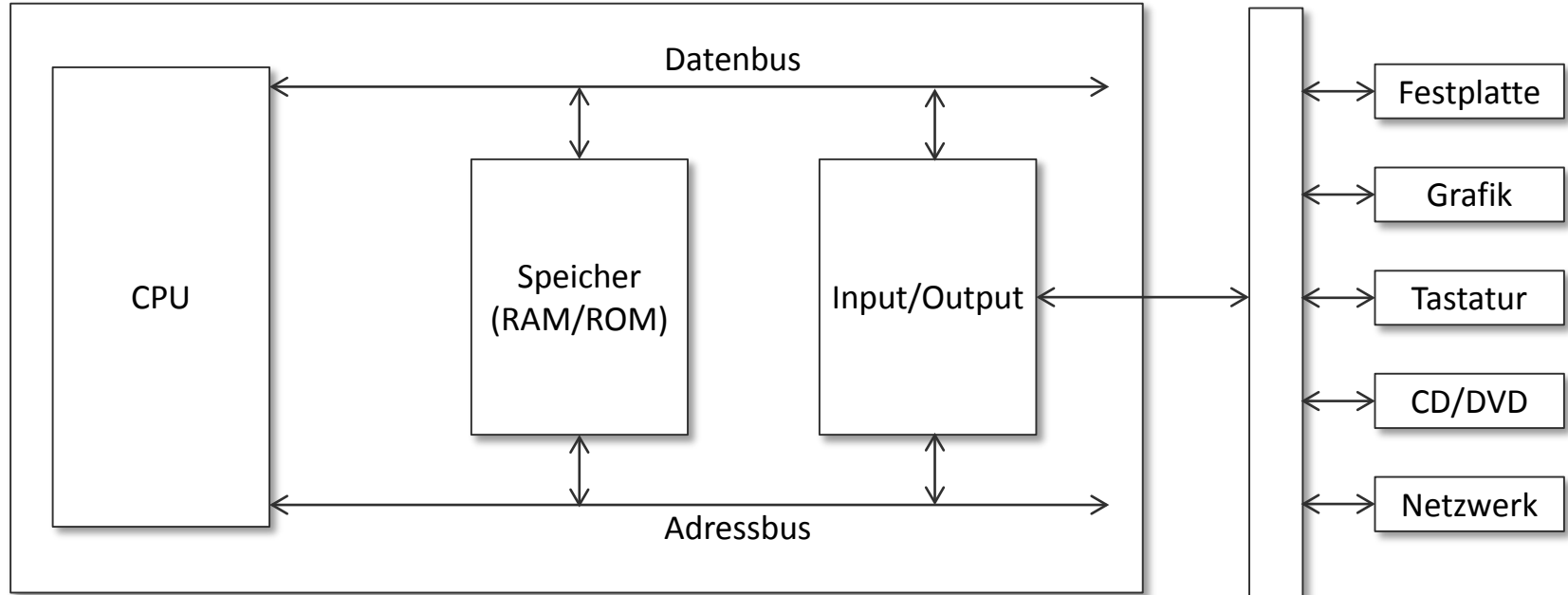
„Eine wundersame Maschine, die es gestattet, ohne große Intelligenz und bei nur mäßiger Leibesanstrengung jedes Werk von hohem wissenschaftlichen oder künstlerischen Wert zu erzeugen.“
Gullivers Reisen (1726)



Jonathan Swift

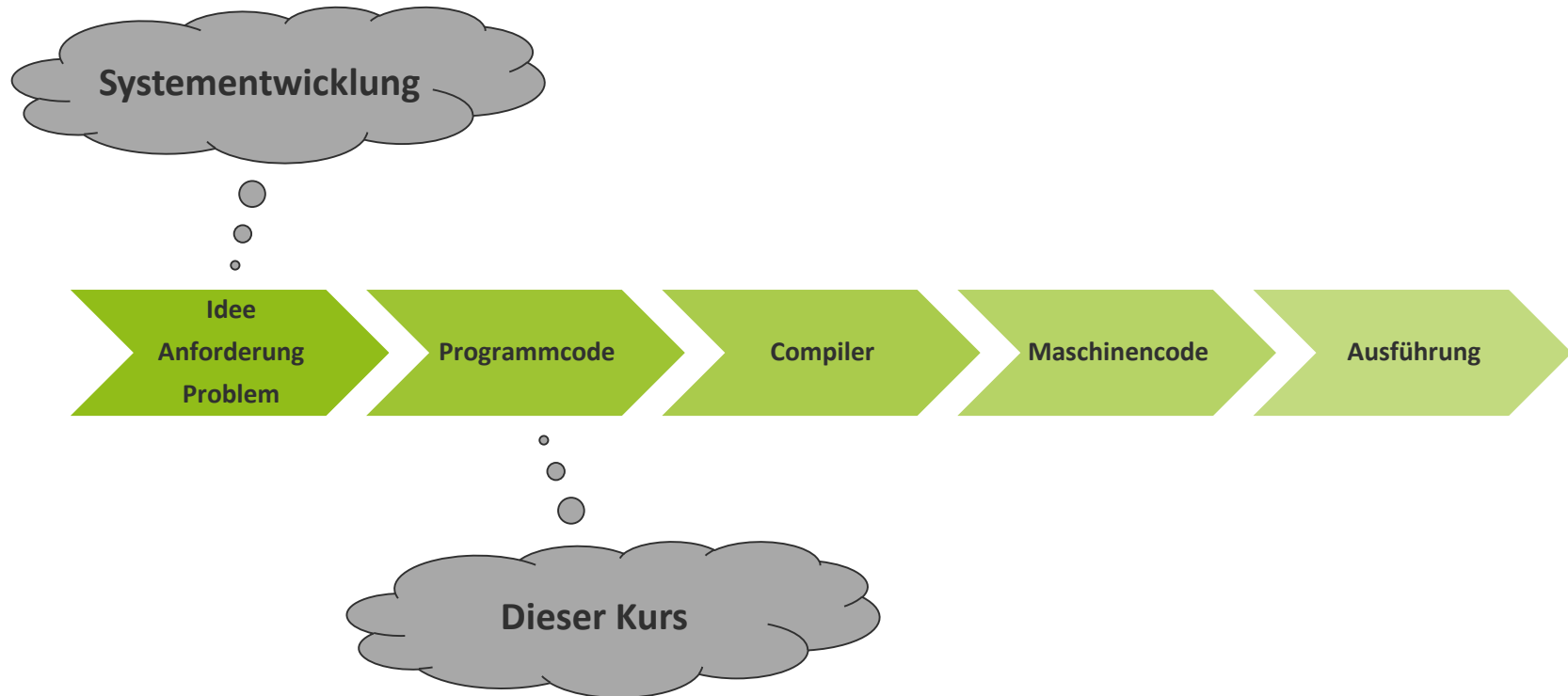
Hardware und Software

- Von Neumann Architektur



- **Anwendersoftware:** Lösung von Problemen aus der externen Welt der Anwender (z.B. Textverarbeitung, Bildbearbeitung, Spiele)
- **Systemsoftware:** zum Betrieb des Rechners, schirmt Hardware vor der Anwendersoftware ab, verwaltet Ressourcen

Grundlegender Software-Entwicklungsprozess



Steuerung von Maschinen – per Algorithmus

- Wohl-spezifizierte **Handlungsvorschrift** zur Lösung eines Problems
- Besteht aus einzelnen Anweisungen, die einen bestimmten Zweck erfüllen
- Starrer **deterministischer Ablauf**

Beispiel-Algorithmus (in sogenanntem *Pseudo-Code*)

Ziel: Berechnung der Summe und des Produkts zweier Zahlen

Eingabe: Zahl_1 und Zahl_2

Ausgabe: Zuerst die Summe und danach das Produkt beider Zahlen

Algorithmus:

1. Lies Eingabe Zahl_1 und lies Eingabe Zahl_2
2. Berechne Summe beider Zahlen
3. Gib berechnete Summe aus
4. Berechne Produkt aus Zahl_1 und Zahl_2
5. Gib berechnetes Produkt aus

Syntax

- Wie unsere natürliche Sprache haben Programmiersprachen **syntaktische** Regeln
 - Geben die **Struktur** der jeweiligen Programmiersprache vor
 - Entsprechen der **Grammatik** der natürlichen Sprache, aber:
 - Der **pragmatische Aspekt** ist unabdingbar und **eindeutig** (ohne Freiheitsgrad)
 - **Keine Fehlertoleranz** bei syntaktisch falschen Anweisungen (Alles-oder-nichts-Prinzip)
- Damit eine **Programmiersprache** vom Computer umgesetzt werden kann, muss man sie in eine sogenannte **Maschinensprache** übersetzen
 - Wird von einem **Compiler/Interpreter** übernommen, der jeweils „seine“ Programmiersprache in Maschinensprache übersetzen kann
 - Funktioniert nur wenn die Syntax korrekt ist



Grenzen der korrekten Syntax

Wenn der Knull nicht gepramelt hat, dann haben entweder das Fipi oder die Gluka geurzt. Wenn der Knull nicht geixt hat, dann hat, falls das Dapi nicht gelüllt hat, die Gluka gepramelt. Wenn der Akro nicht geurzt hat, dann hat das Fipi entweder gepramelt oder gewatzelt. Wenn weder der Knull noch das Dapi geixt haben, dann hat die Gluka geurzt. Wenn das Dapi nicht gewatzelt hat, dann hat, falls der Knull nicht geurzt hat, das Fipi gelüllt. Jeder hat etwas getan, keine zwei taten dasselbe, wer hat was getan?

Der Knull hat geixt, die Gluka hat geurzt, das Dapi hat gewatzelt, das Fipi hat gepramelt, und der Akru hat gelüllt.

Syntax (und Logik) ist be„deutungslos“.

Semantik

- **Absicht** des Entwicklers/Anwenders, d.h. was soll das Programm machen?
 - Kann nicht maschinell überprüft werden
- Analogie zu natürlichen Sprachen: Die **Aussage eines Satzes**
 - Gibt dem Programm die **Bedeutung**
 - Fehler in der Semantik werden nicht vom Computer entdeckt
- Um korrekte Semantik herzustellen sollte man algorithmisch denken können.
- Beispiele in menschlicher Sprache
 - Morgen ist heute gestern.
 - Eins plus eins ist gleich fünf.
 - „Wenn ich lügend sage, dass ich lüge, lüge ich oder sage ich Wahres?“ „Du sagst Wahres.“
„Wenn ich Wahres sage und sage, dass ich lüge, lüge ich?“ „Du lügst offenbar.“
(Eubulides, 4. Jhd v. Chr.)

Semantik

- Semantik = Informationsgehalt
 - gibt dem Programm die **Bedeutung**
 - Fehler in der Semantik werden nicht vom Computer entdeckt
 - Beispiele in menschlicher Sprache:
 - Morgen war heute gestern.
 - Eins plus eins ist fünf.
 - π ist genau 3.
- Um korrekte Semantik herzustellen sollte man algorithmisch denken können

Semantik (2)

Afugrnud enier Sduite an enier Elingshcen Unvirestiät ist es eagl, in wleher Rienhnelfoge die Bcuhtsbaen in eniem Wrot sethen, das enizg wcihitge dbaei ist, dsas der estre und lzete Bcuhtsbae am rcihgiten Paltz snid. Der Rset knan ttolaer Bölsdinn sien, und du knasnt es torztedm onhe Porbelme lseen. Das ghet dseahlb, wiel wir nchit Bcuhtsbae für Bcuhtsbae enizlen lseen, snodren Wröetr als Gnaezs. Smtimt's?

Semantik ist nicht (immer) an eine Syntax gebunden.

Zusammenfassend...

Die **Syntax** definiert, wie Sprachelemente zusammenhängen und verwendet werden dürfen.

Die **Semantik** gibt dem Programm die Bedeutung.

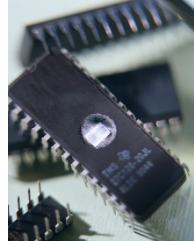
Programmieren ist das Formulieren einer Problemlösung in Befehlsanweisungen mithilfe einer Sprache, die automatisch in Befehle übersetzt wird, die ein Computer ausführen kann.

Ein **Algorithmus** ist eine Reihe von zusammengehörigen Befehlen, die einen bestimmten Zweck erfüllen.

Mensch vs. Maschine

Computer: schnell, aber dumm

- Physikalische Grenzen
- Eigenständigkeit?



Menschliches Gehirn: klug, aber langsam

- besitzt ca. 10 Milliarden Neuronen und 60 Billionen Synapsen
- Hochkomplexer, nicht-linearer, paralleler Computer

	Computer	Gehirn
Geschwindigkeit	Logische Gatter (Silizium) 10^{-9} s	Neuronale Ereignisse 10^{-3} s
Energieeffizienz	10^{-6} J/Operation	10^{-16} J/Operation
Stärken	$a+b$, $a*b$, ..., if-then-else, Enumeration, ...	Mustererkennung, Wahrnehmung, Bewegungskontrolle, ...
Eigenschaften	robust, fehlertolerant	fehleranfällig

Algorithmisches Denken

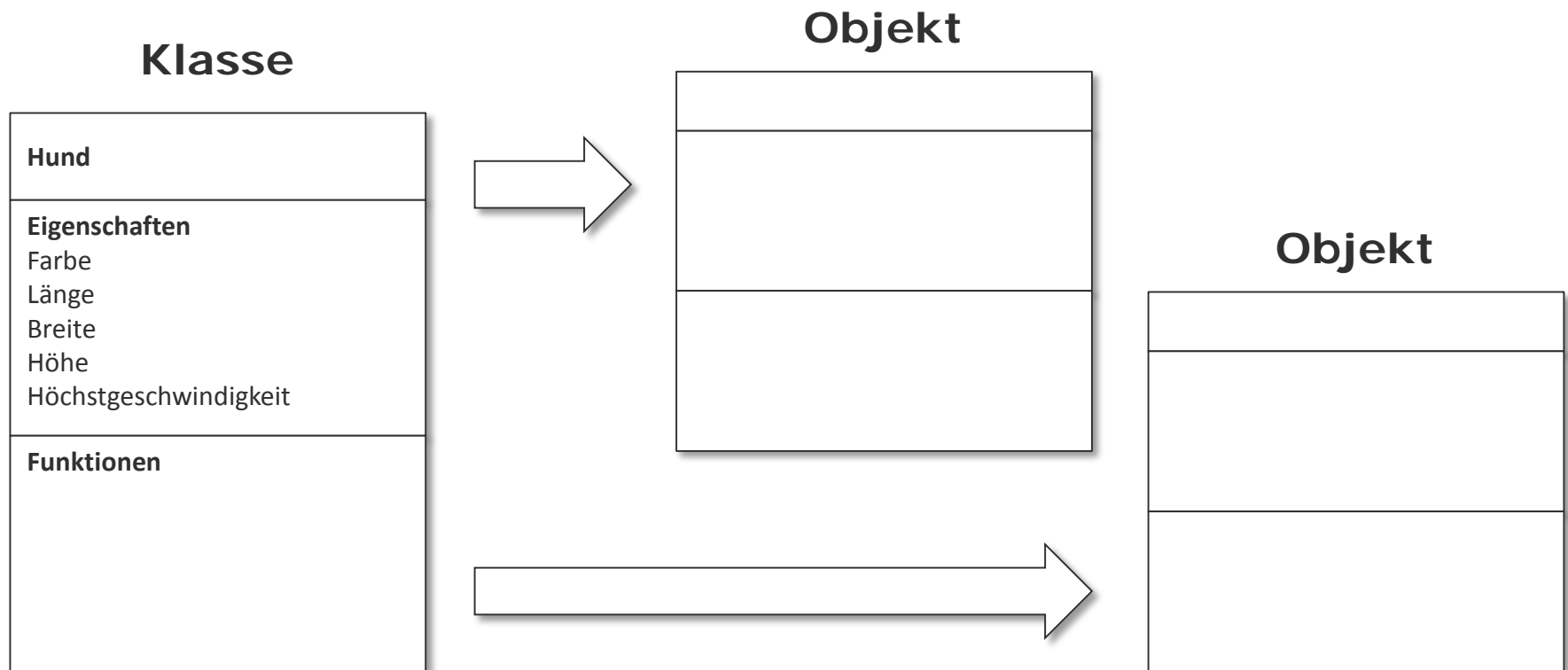
- Folge: Man muss einem Computer in exakten Schritten vorgeben was er durchführen soll – per Algorithmus!
- Algorithmisches Denken bedeutet eine **Problemstellung** zu **analysieren** und dann **Anweisungen** zu **formulieren** die für die Problemlösung notwendig sind
 - kein starres Konzept, sondern eine erlernbare Fähigkeit
 - von einfachen hin zu komplexeren Problemstellungen
- Die Anweisungen sollten möglichst alle auftretenden Möglichkeiten behandeln (Robustheit, **Fehlertoleranz**)
 - „was wäre wenn“-Szenarien beachten!
 - siehe Beispiel aus der Einführungsveranstaltung (Brot backen)
- Es gibt beim Programmieren nicht *die eine* Lösung, es ist vielmehr ein **kreativer Prozess**

Abstraktes Denken

- „Abstraktion [...] bezeichnet meist den **induktiven** Denkprozess des **Weglassens** von Einzelheiten und des Überführens auf etwas **Allgemeineres** oder **Einfacheres**.“ (Wikipedia, 03.2015)
- In Bezug auf die Programmierung
 - Welche Problemstellungen treten auf?
 - Welche **Objekte**, Interaktionen, Prozesse werden auftreten?
 - Was sind die wirklich **relevanten** Eigenschaften und Beziehungen der Objekte?
 - **Pragmatismus**: nur das modellieren, was zielführend ist – Beispiele?

Abstraktes Denken und Objektorientierung

- Um ein Problem zu lösen, sollte man sich auf das **Wesentliche** fokussieren
- Wie kann man die reale Problemstellung möglichst sinnvoll abbilden?
 - Modellierung von **Objekten** und deren **Interaktionen**!
 - Man unterscheidet zwischen **Klassen** und **Objekten**!



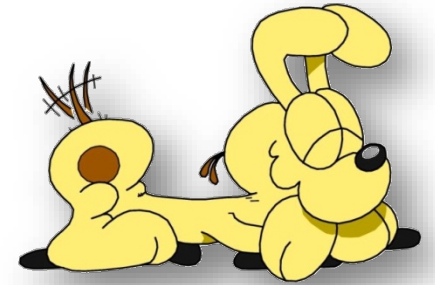
Objektorientierte Programmierung

- Anforderungen beim Programmieren:
 - Lesbarkeit
 - Wiederverwendbarkeit
 - Wartbarkeit
 - Sicherheit
- Früher:
 - prozedurales Programmieren
 - Herunterschreiben von mehreren tausend Zeilen mit Anweisungen
- Heute:
 - realitätsnahe, aber **vereinfachte Modellierung** der realen Welt
 - Realität kann schematisch in **Objekte** eingeteilt werden, die miteinander agieren
 - **Pragmatismus** spielt eine große Rolle – Modellierung geschieht immer zielabhängig, es wird also niemals *die* Modellierung geben



Objektorientierte Programmierung (2)

- C# ist eine Objektorientierte Programmiersprache
 - es gibt **Klassen**, die bestimmte Eigenschaften und Funktionen besitzen
 - jedes **Objekt** ist eine **Instanz** einer bestimmten Klasse
 - Objekte können mit anderen Objekten **interagieren**
- Analogie zur realen Welt – **Klasse Hund**
 - Hunde haben die **Eigenschaften** Größe und Farbe
 - Hunde haben die **Funktionen** *Bellen()*, *Fressen()*, *Spielein(Hund)*
 - Funktionen können komplexe Abläufe abbilden, z.B. dass nur Hunde miteinander spielen, die dieselbe Fellfarbe besitzen
- **Instanzen** der Klasse Hund
 - Bello und Hulk sind Objekte der Klasse Hund
 - *Bello* ist ein 60cm großer Hund mit *rotem* Fell
 - *Hulk* ist ein 70m großer Hund mit *grünem* Fell
 - im Programmablauf kann auf Funktionen und Eigenschaften der Objekte zugegriffen werden, z.B. *Bello.Spielen(Hulk)*;



Agenda

- Organisatorisches
- Einführung in die Welt der (Programmier-)Sprachen
- **.NET-Umgebung und Visual Studio**
 - Grundbegriffe
 - Visual Studio starten
- Das erste Programm

Grundbegriffe

- Quellcode
 - **Programmcode** in der jeweiligen Programmiersprache
- Assembler
 - **maschinennahe** Sprache, nicht für jeden Computer identisch!
- Maschinensprache
 - **maschinenlesbare** Sprache, systemspezifisch
- IDE (= Integrated Development Environment)
 - z.B. Visual Studio
 - Programm, dass die Entwicklung von Code und Programmen auf unterschiedlichste Weise unterstützt, wichtigste Funktionen
 - **Text-Editor** mit automatischer Erkennung der Syntax (ähnlich der Rechtschreibprüfung)
 - Compiler (**übersetzt** Quellcode in maschinenlesbaren Code)
 - Debugger (Werkzeug zur **Diagnose** von Fehlern im Programmcode)

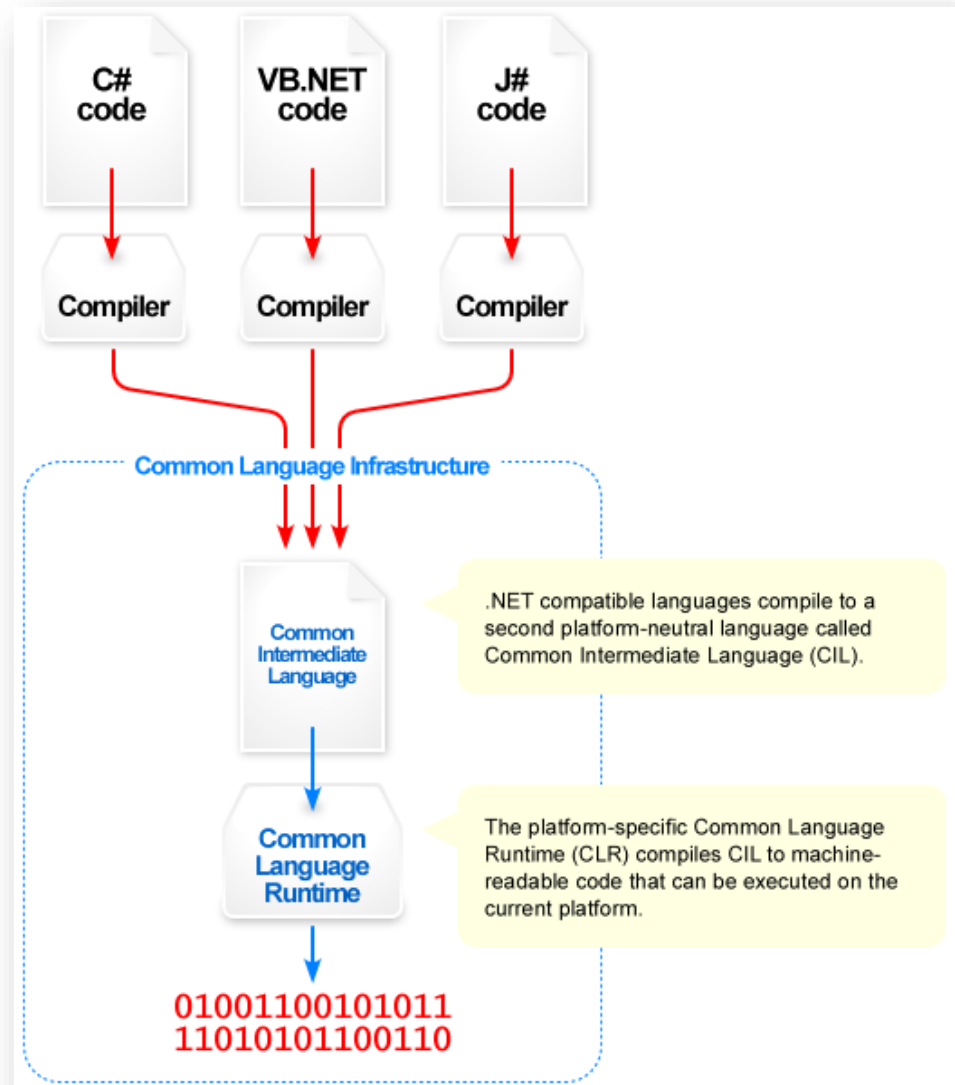
```
Console.WriteLine("¡Diga!");
```

```
movb $0x61, %al  (AT&T Syntax)
mov al, 61h      (Intel Syntax)
```

```
10110000 01100001
```

Grundbegriffe in .NET

- .NET ist die Plattform mit der Visual Studio arbeitet
- Viele Vorteile, u.a.:
 - Unterstützung unterschiedlicher Programmiersprachen
 - Möglichkeiten der Code-Generierung
 - Sicherheitsaspekte
- In unserem Kurs von sekundärer Bedeutung



Microsoft Visual Studio

- Quelldatei (engl. source file)
 - den **Code** beinhaltende Datei (endet bei C# mit .cs)
- Projekt (engl. project)
 - ein Projekt fasst Quelldateien zusammen, die zu einem **Anwendungspaket** gehören
 - aus einem Projekt erzeugt Visual Studio eine Assembly (*.exe, *.dll o.ä.)
 - unterschiedliche Projekte können miteinander agieren, Beispiel:
 - ein Projekt, welches die grafische **Oberfläche** eines Programms beinhaltet
 - ein Projekt, welches die **Logik** des Programms beinhaltet
 - Vor-/Nachteile?
- Projektmappe (engl. *solution*)
 - eine Projektmappe fasst Projekte zusammen, die gemeinsam eine **Anwendung** bilden

Visual Studio herunterladen

- Über das MS DreamSpark-Programm erhältlich (ehemals MSDNAA-Programm)
- Version: Professional
- Sprache könnt ihr selbst wählen, Englisch bietet sich aber an, da es intuitiver ist

The screenshot shows the download page for Microsoft Visual Studio Professional 2013 32-bit Web Installer (German) via DreamSpark. The page has a green header with the title. Below the header, there is a product image on the left and a list of details on the right. The details include the manufacturer (Microsoft Corporation), platform (Windows), delivery type (Download), and availability for academic users. A 'Kostenlos' (Free) badge is present, along with a 'In den Warenkorb' (Add to cart) button and a 'Sind Sie berechtigt?' (Are you eligible?) link. Social media icons for Twitter and Facebook are also visible. At the bottom, there are tabs for 'Beschreibung' (Description), 'Systemanforderungen' (System requirements), and 'Sind Sie berechtigt?'. The 'Beschreibung' tab is active, showing a brief description of Visual Studio Professional 2013.

Microsoft Visual Studio Professional 2013 32-bit Web Installer (German) - DreamSpark - Download

Hersteller: Microsoft Corporation

Plattformen: Windows

Liefertyp: Download

Verfügbar für: Akademische Benutzer
Vorrätig

Kostenlos

In den Warenkorb

Sind Sie berechtigt?

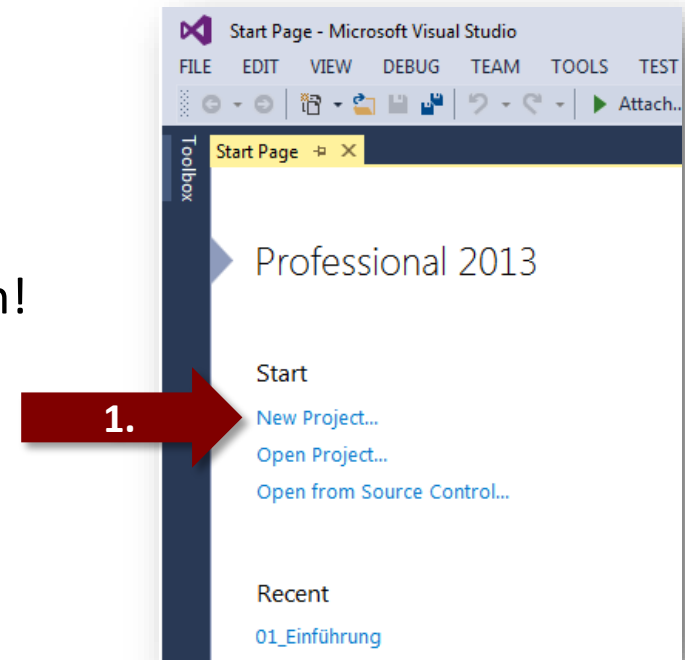
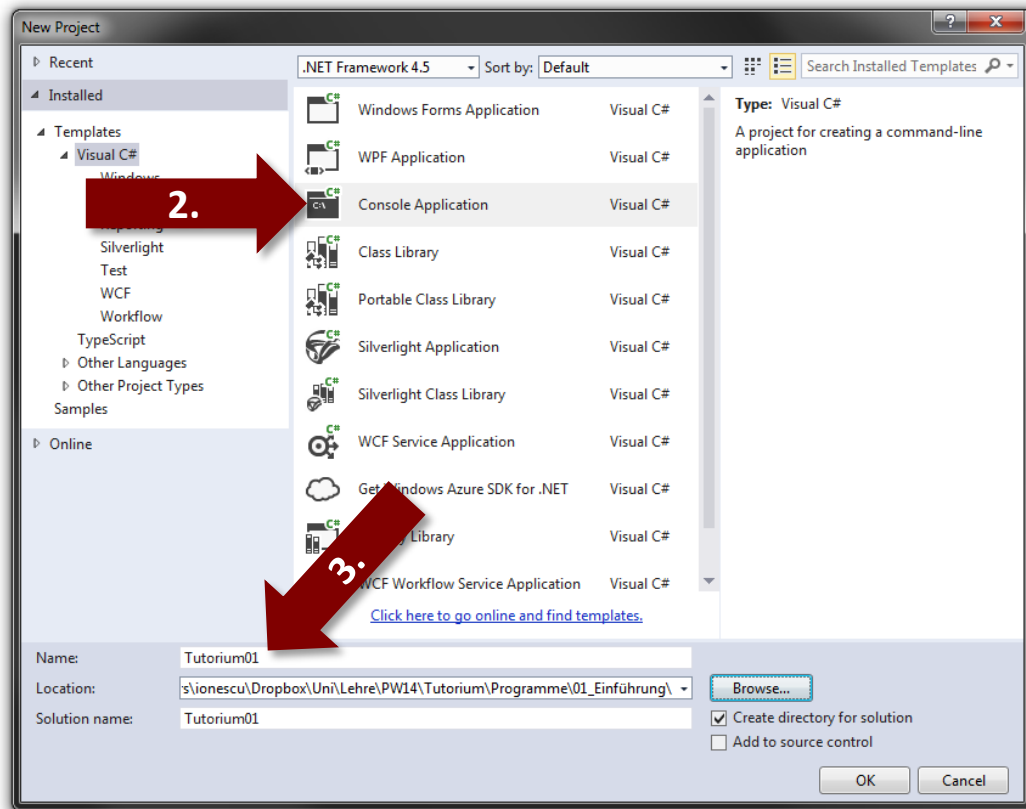
[Beschreibung](#) [Systemanforderungen](#) [Sind Sie berechtigt?](#)

Microsoft Visual Studio Professional 2013

Visual Studio Professional 2013 can help you create applications that have innovative user experiences that delight your customers. The integrated environment makes complex tasks easier so that you can focus on achieving your goals. By using Visual Studio Professional 2013, you can deliver quality applications on Windows, Windows Phone, Office, the web, or the cloud.

Ein Projekt in Visual Studio anlegen

- Startseite
- **Neues Projekt** öffnen
- **Konsolenanwendung** auswählen
- Direkt **sinnvollen Namen** für ein Projekt angeben!



Agenda

- Organisatorisches
- Einführung in die Welt der (Programmier-)Sprachen
- .NET-Umgebung und Visual Studio
- **Das erste Programm**
 - „Hello World“
 - Ein- und Ausgabe
 - Kommentieren von Programmcode
 - Nützliche Tipps

Die Main-Methode

- Dient immer als **Einstiegspunkt** in das Programm
 - jedes Programm besitzt einen Einstiegspunkt
 - in einer Solution mit unterschiedlichen startfähigen Projekten kann man angeben, welches Projekt gestartet werden soll
 - über **args** können Parameter direkt beim Programmstart an das Programm weitergegeben werden (für uns noch nicht relevant)
 - die Methode kann eine ganze Zahl zurückgeben, der Wert 0 indiziert eine fehlerfreie Ausführung
- Details später im Kapitel „Methoden“

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```


Konsolen-Ausgabe



- Ausgabe auf der Konsole über:

```
Console.Write(<expression>);
```

```
Console.WriteLine(<expression>);
```

```
The Yip Yips
>> Chicken.
>> Chicken.
>> Yip. Yip.
>> Yipyipyipyipyip.
>> Baaaaaaawk Baaawk Bawk Bawk Bawk Bawk Bawk Bawk Bawk-up!
>> Baaaaaaawk Baaawk Bawk Bawk Bawk Bawk Bawk Bawk Bawk-up!
>> Uh-huh. uh-huh.
>> Nope. Nopenopenopenope.
>> Book. Book.
>> Bookbookbookbook.
```

- Gültige Werte für <expression>

- leere Zeile:

```
Console.WriteLine();
```
- Zeichenfolgen, sog. Strings:

```
Console.Write("This is a text.");
```
- Gleichungen:

```
Console.WriteLine(1+1);
```
- Platzhalter möglich:

```
Console.WriteLine("1+2 is {0}{1}", 1+2, "!");
```

- Achtung!

- Strings müssen in Anführungsstrichen geschrieben werden. Ansonsten werden sie als Anweisungen interpretiert, was zu einem Compilerfehler führen kann.
- Welche Ausgaben entstehen bei den folgenden Anweisungen?
 - ```
Console.WriteLine(1+1);
```
  - ```
Console.WriteLine("1+2");
```
 - ```
Console.WriteLine(1 plus 1);
```
  - ```
Console.WriteLine(1 + "1");
```

Konsolen-Eingabe

- Eingabe auf der Konsole über:

```
Console.Read();  
Console.ReadLine();
```

- Keine Parameter-Übergabe!

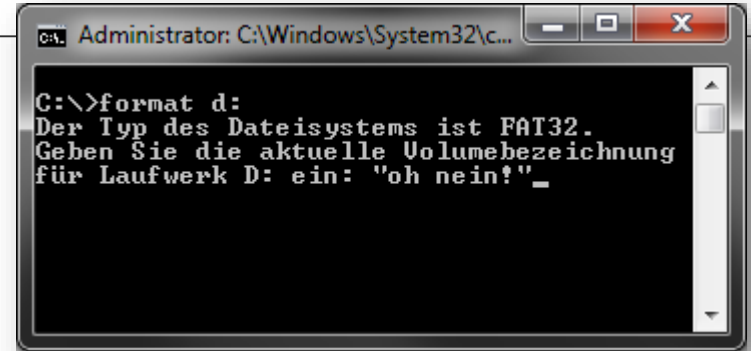
- Blockiert die Programmausführung bis zur Benutzereingabe
 - Programm rechnet im Hintergrund nicht weiter, sondern wartet auf die Eingabe des Benutzers. Die **Blockierung** wird erst aufgehoben, wenn der Benutzer die Eingabe mit „Enter“ abschließt.

- Anweisungen lassen sich kombinieren:

```
Console.WriteLine(Console.ReadLine());
```

- Was wird bei den folgenden Anweisungen ausgegeben? Warum?

```
Console.WriteLine("Console.ReadLine()");  
Console.WriteLine("Hallo!"); Console.ReadLine();
```



Tipp zum Erstellen von Konsolen-Programmen

- Ein in Visual Studio gestartetes Programm schließt immer automatisch nach Ablauf – und damit auch das Konsolenfenster
- Konsolen-Ausgaben sind meist so schnell, dass man sie nicht mehr lesen kann
- Daher ist es sinnvoll, das Beenden zu kontrollieren

```
Console.WriteLine("Press Enter to close the application");  
Console.ReadLine();           //another possibility: Console.ReadKey()
```

Kommentare im Quellcode

- Der Quellcode kann mit Kommentaren versehen werden
- Wichtig bei längerfristigen Projekten und Teamarbeit
- Kommentare werden vom Compiler ignoriert
- **Quellcodekommentierung ist eine fundamentale Fertigkeit**
- Zeilenkommentar: Inhalt bis zum Zeilenende gilt als Kommentar

```
<expression> //This is a comment.
```
- Blockkommentar: Nur der Inhalt zwischen `/*` und `*/` wird als Kommentar behandelt (Zeilenumbruch möglich)

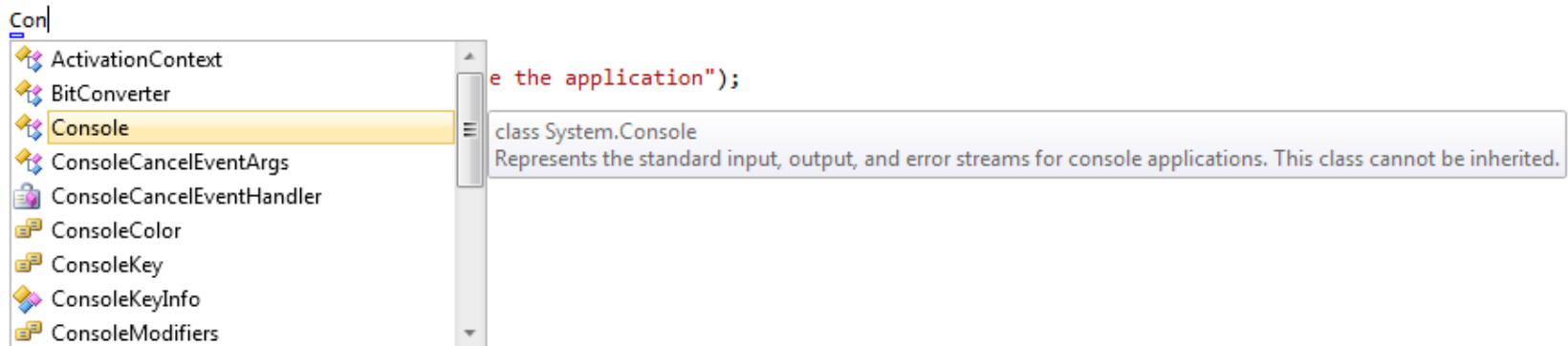
```
<expression1> /* This is another comment. */ <expression2>
```

```
/* And another  
one. */
```
- Auch geeignet, um derzeit nicht genutzte Code-Fragmente zu speichern (sog. Auskommentieren)

Nützliche Tipps

- **Starten** des Programms: F5
- Starten und Durchlaufen in **Einzelschritten**: F10
- **Beenden** des Programms: F6
- Springen zur **Definition** einer Funktion oder Variablen: F12
- **Copy&Paste** nutzen (viele Code-Fragmente nutzt man öfter)
- **IntelliSense** nutzen
 - beim Eintippen einiger Buchstaben bekommt man Vorschläge angezeigt
 - weniger Tipparbeit und geringere **Fehleranfälligkeit**

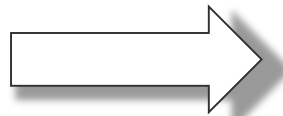
```
//This is a comment.  
/* This is another comment. */
```



Code-Regionen einklappen

- Um einen Code besser lesen zu können, sollte man im Editor logische Einheiten des Codes **einklappen**
 - Namensräume,
 - Klassen,
 - Funktionen
 - ...
- Auch eigene Definition über **Regions** möglich

```
#region my code  
//...  
#endregion my code
```



```
my code
```

- Code wird eingeklappt, aber trotzdem ausgeführt!

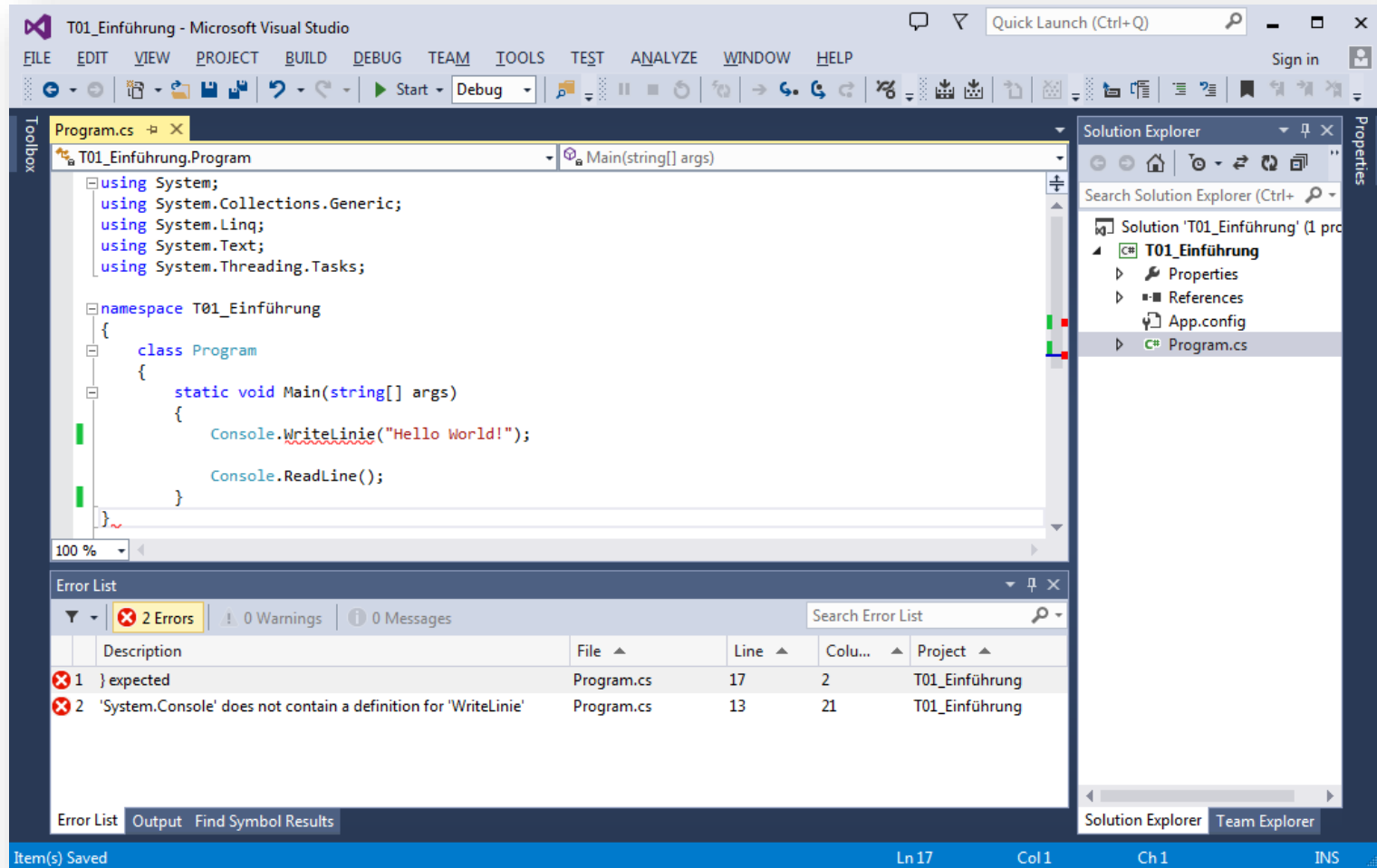
Häufig gemachte Fehler

- C# unterscheidet zwischen Groß- und Kleinschreibung
 - `Console.WriteLine()` vs. `Console.Writeline()`
- Eine Anweisung wird immer mit einem Semikolon beendet
 - `Console.WriteLine();`
 - `int a = 10;`
- Absätze werden wie Leerzeichen behandelt
- Tabulatoren zum Einrücken von Text sind optional, sollten aber genutzt werden
- In den meisten Fällen lässt sich der Quellcode auch in einer einzigen Zeile schreiben. Daran haben später alle Spaß!

• C# unterscheidet zwischen Groß- und Kleinschreibung – `Console.WriteLine()` vs. `Console.Writeline()` • Eine Anweisung wird immer mit einem Semikolon beendet `Console.WriteLine(); int a = 10;` • Absätze werden wie Leerzeichen behandelt • Tabulatoren sind optional, aber bitte nutzen • In den meisten Fällen lässt sich der Quellcode auch in einer einzigen Zeile schreiben. Daran haben später alle Spaß!

Häufig gemachte Fehler

- Wenn Ihr ein fehlerhaftes Programm ausführen möchtet, gibt es **Compilerfehler**
- Man lernt mit der Zeit, was sie zu bedeuten haben



Das Letzte: Hello World objektorientiert

```
class Program
{
    static void Main(string[] args)
    {
        new HelloWriter(); //3.create a new object from the class Hello

        Console.ReadLine();
    }
}

class HelloWriter //1.we create a new class
{
    public HelloWriter() //2.'constructor' method that is called when a new object of Hello is created
    {
        Console.Write("Hello World");
    }
}
```

Erzeugen eines neuen Objekts durch das Schlüsselwort **new**