

DEPARTMENT WIRTSCHAFTSINFORMATIK

FACHBEREICH WIRTSCHAFTSWISSENSCHAFT

# Programmieren für Wirtschaftswissenschaftler

## SS 2015

Lucian Ionescu

Blockveranstaltung 16.03–27.3.2015

4. Methoden

# Agenda

---

- **Methoden**
  - Motivation
  - Erstellung eigener Methoden
  - Parameter
  - Rückgabewerte
  - Aufruf einer Methode
- Rekursion

# Was sind Methoden?

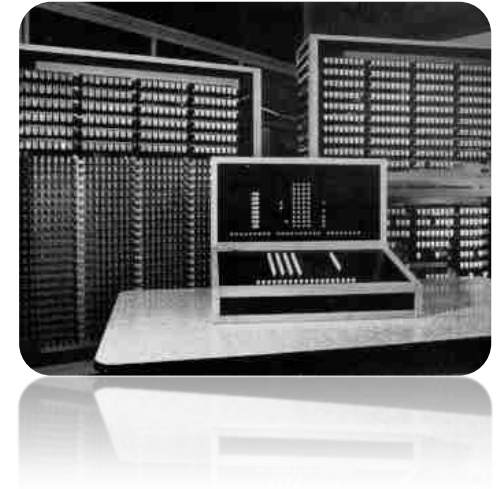
---

- Ausgangsfrage
  - ein längeres Programm kann **mehrere tausend Zeilen Code** besitzen
  - verzahnte Verzweigungen und Schleifen führen leicht zu **Unübersichtlichkeit**
  - Wie sollen mehrere Personen mit **unterschiedlichen Aufgabenbereichen** an einer riesigen Menge von Programmcode arbeiten?
- Lösung
  - **Auslagern** von Code-Abschnitten in **Methoden (Funktionen)**, die einen bestimmten Zweck erfüllen
  - Zerlegung von komplexen Abläufen in **Teilaufgaben** → Definition von Schnittstellen
    - spezielle inhaltliche Kenntnisse müssen nicht von jedem Programmierer beherrscht werden
    - „ich gebe dir etwas (Parameter) und erwarte dafür ein Ergebnis (Rückgabewert)“
  - **Wiederverwendbarkeit** von Code-Abschnitten ist wichtig, um **Redundanzen zu vermeiden**
    - weniger Programmieraufwand, Teillösungen können wiederverwendet werden
    - bessere Wartung

# Methoden

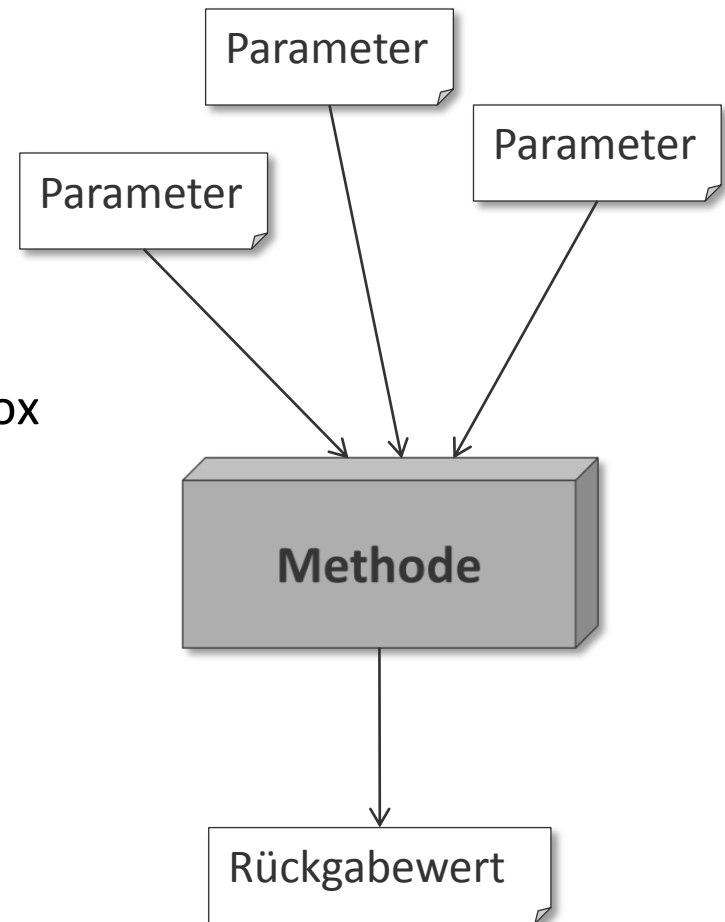
---

- Ihr habt alle schon Methoden benutzt:
  - Main-Methode
  - `Console.WriteLine()`
  - `Console.ReadLine()`
  - Konstruktoren
- Analog zu mathematischen Funktionen, Beispiel  $f(x) = x^2$ 
  - Welche Parameter bekommt die Funktion?
  - Welchen Wert gibt die Funktion zurück?
- Reales Beispiel:
  - Kaffeeautomat
  - Welche Parameter bekommt der Automat?
  - Was gibt der Automat zurück?



# Methoden

- Wodurch ist eine Methode charakterisiert?
  - **Name** der Methode
  - Übergebende **Parameter**(-liste)
  - **Rückgabotyp**
- Nach außen fungiert eine Methode als Black-Box
- Welche Methoden sind schon bekannt?
  - `double y = Math.Exp(x);`
  - `Console.WriteLine("Pampelmuse");`
  - `string eingabe = Console.ReadLine();`



# Es gibt zwei Arten von Methoden

---

- Objektmethoden
  - legen das Verhalten eines **konkreten Objekts** fest
  - arbeiten oft mit Eigenschaften des Objekts (Zugriff auf und Manipulation von Daten)
  - Beispiele:
    - `kontoA.Abheben(10); // Abheben von 10 EUR von kontoA`
    - `kontoA.Überweisen(10, kontoB); // Überweisen von 10 EUR von kontoA auf kontoB`
- Klassenmethoden
  - haben **keine Bindung** an ein konkretes Objekt
  - bilden meist **allgemeine Funktionen** ab
  - werden durch das Schlüsselwort `static` indiziert
  - Beispiele:
    - `Console.ReadLine();`
    - `Console.WriteLine("Der Kaplan klebt klappbare Pappplakate an.");`
    - `Math.Exp(x);`

# Eigene Methoden erstellen

---

- Allgemeine Syntax

```
[static] <datatype> <name> (<datatype1> <parameter1>, <datatype2> <parameter2>, ...)  
{  
    <expression>  
    return <result>;  
}
```

- Beispiel

```
static int Square (int x)  
{  
    int y = x*x;  
    return y;  
}
```

- Anweisungen nach `return` werden ignoriert
- Parameter und in der Methode erstellte Variablen sind nur in der Methode gültig!

# Parameter einer Methode

---

- Die Anzahl an Parametern ist beliebig → „abzählbar unendlich“
- Funktioniert analog zur normalen **Deklaration** von Variablen
- Parameter (und andere in der Methode erstellte Variablen) sind nur innerhalb des Methodenrumpfes sichtbar
- Wenn es außerhalb der Methode eine Variable desselben Namens und Typs gibt, so hat die Methoden-Variable Vorrang

```
static int Square (int x)
{
    int y = x*x;
    return y;
}
```



# Rückgabewert eine Methode

---

- **Ergebnis** der Methode
- Die Rückgabe eines Wertes geschieht durch `return`
- Variable oder Wert hinter `return` muss mit dem **Rückgabe-Typen** übereinstimmen!
- Funktionen erleichtern das Programmieren erheblich:
  - nur das **Ergebnis** (vgl. Rückgabewert) ist für den Aufrufer sichtbar
  - lediglich die nötigen **Eingaben/Einstellung** (vgl. Parameter) wie z.B. die zu quadrierende Zahl sind notwendig für die Rechnung
- Spezialfall: `void`
  - wenn eine Funktion **keinen Rückgabewert** hat, wird `void` als Rückgabetyt genutzt
  - sinnvoll, wenn eine Methode beispielsweise nur Konsolenausgaben ausführt
  - Funktion läuft entweder komplett durch oder kann durch `return`; beendet werden

# Beispiel für eine objektorientierte Methode

```
class Program
{
    static void Main(string[] args)
    {
        Account konto = new Account("Schlaudia Ciffer", "Berlin", 0);
        konto.balance += 10;
        Console.WriteLine(konto.GetInfo(true));
    }
}

class Account
{
    public string name;
    public string address;
    public float balance; // a variable for the current account balance

    public Account(string name, string address, int startBalance) // constructor
    {
        this.balance = startBalance;
        this.name = name;
        this.address = address;
        Console.WriteLine("A new account has been created.");
    }

    public string GetInfo(bool outputBalance)
    {
        if(outputBalance)
            return name + " (" + address + ") has " + balance + "Euro";
        else
            return name + " (" + address + ")";
    }
}
```

# Rückgabewert einer Methode (2)

---

- Was passiert wenn eine Funktion einen Rückgabewert spezifiziert hat (also nicht mit `void` definiert wurde) aber keinen Wert zurückgibt?
- Was passiert wenn Anweisungen nach der `return`-Anweisung stehen?
- Was passiert wenn nicht jeder Pfad bei der Ausführung bei einem `return` endet (etwa bei Verschachtelungen mit `if-else`)?

# Aufruf einer Methode

---

- Das Aufrufen einer Methode kennt ihr schon!
- Obiges Beispiel:

```
string eingabe = Console.ReadLine(); // Eingabe einlesen
int x = Convert.ToInt32(eingabe);    // Eingabe in Zahl konvertieren
int square = Square(x);              // Quadrat berechnen und speichern
```

- Frage: Was geschieht, wenn man eine Funktion **mit Rückgabetyt** nicht in Verbindung mit einer **Zuweisung** nutzt?

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

```
Square(5);
```

# Gültigkeitsbereich von Variablen

---

- Sichtbare Variablen in Methoden
  - **Parameter(-Variablen)** einer Methode sind (nur) innerhalb des Methodenrumpfes gültig
  - in der Methode deklarierte **lokale Variablen** sind ebenfalls (nur) innerhalb des Methodenrumpfes gültig
- Übergebene Variablen
  - Variablen, die beim Aufruf **übergeben** werden sind in der Methode selbst nicht nutzbar und werden **nicht verändert**
  - Aber: die Werte sind über die Parameter-Variablen nutzbar!
- Rückgabewerte
  - Rückgabewerte sind nur dort sichtbar, wo die Methode **aufgerufen** wurde
  - Wenn der Rückgabewert weiter genutzt werden soll, muss dieser in einer Variablen **gespeichert werden!**

# Erweitert: Überladen von Methoden

---

- Es können Methoden mit demselben Namen existieren, die aber unterschiedliche Parameter besitzen
- Die Entscheidung, welche Methode beim Aufruf genutzt werden soll, fällt durch die Signatur (also Parameterliste)

```
static int Multiplication(int x, int y)
{
    return x*y;
}
```

```
static int Multiplication(int x, int y, int z)
{
    return x*y*z;
}
```

# Erweitert: Möglichkeiten der Übergabe

---

- Passing by Value
  - für jeden Parameter wird eine **Kopie** erstellt
  - Veränderungen sind außerhalb einer Methode nicht sichtbar
  - wird standardmäßig bei allen **einfachen Datentypen** genutzt
- Passing by Reference mit **ref**
  - ändert die Methode etwas an einem Parameter, so wird die Änderung auch nach Ablauf der Methode **beibehalten**
  - wird standardmäßig bei allen **komplexen Datentypen** genutzt (z.B. Listen)
  - Parameter muss zuvor initialisiert werden (normale Variablen-Initialisierung)
- Passing by Reference mit **out**
  - notwendig, wenn eine Methode **mehr als einen** Rückgabewert haben soll
  - Änderungen werden wie bei **ref** **beibehalten**
  - keine Initialisierung nötig (geschieht implizit wie bei normalen Parametern)

# Agenda

---

- Methoden
- **Rekursion**



# Rekursion (*Divide and Conquer*)

---

- Funktionen können sich auch **selbst aufrufen**
- Beispiel: **Fakultät** berechnen über Rekursion ( $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ )

```
static ulong Factorial(ulong x)
{
    if(x <= 1) return 1;           // Rekursionsende
    return x * Factorial(x-1);     // Rekursionsaufruf
}
```

- Es muss ein **Rekursionsende** geben, sonst führt die Rekursion zu **unendlich** vielen Aufrufen
- Übermäßige Nutzung von Rekursionen führen zum *Stack Overflow*
- **Eleganter Ersatz** für Schleifen
- Um Rekursion intuitiv zu verwenden ist ein wenig Übung notwendig

# Zusammenfassung

---

- Methoden kapseln Code-Abschnitte, die somit **wiederverwendet** werden können
- Eine Methode wird definiert durch
  - Rückgabetyp
  - Parameter(-liste)
  - Name
- **Rückgabe** erfolgt durch das Schlüsselwort `return`
- Parameter und in der Methode erstellte Variablen sind nur **innerhalb der Methode sichtbar!**
- Es können unterschiedliche Methoden mit **demselben Namen**, aber unterschiedlichen Parameterlisten existieren
- Methoden können sich **selbst aufrufen!**

# Anmerkung

---

- Was ist der Unterschied zwischen *Methode* und *Funktion*?
  - jede Methode ist auch eine Funktion
  - es gibt Funktionen, die keine Methoden sind, dazu evtl. später

