Freie Universität Berlin

Department Wirtschaftsinformatik
Fachbereich Wirtschaftswissenschaft

Programmieren für Wirtschaftswissenschaftler
SS 2015

Lucian Ionescu

Blockveranstaltung 16.03–27.3.2015

3. Verzweigungen und Schleifen

Agenda

- **Boolesche Ausdrücke**
  - Motivation
  - Logische Operatoren
  - Short Circuiting
- Verzweigungen
- Schleifen

Boolesche Ausdrücke

Beispiel

- Wenn es morgen regnet, nehme ich einen Schirm mit.
  - Logische Formel (Implikation): Regen → Schirm

- Benannt nach George Boole
  - 1815-1864
  - Mathematiker, Logiker und Philosoph
  - Entwarf ein Logikkalkül, das Wahrheitswerte von Gleichungen erfassbar machte
  - wahr = 1, falsch = 0
  - Beispiel:

„x oder y“

→ „nicht (nicht x und nicht y)“

→ $1 - (1 - x) \cdot (1 - y)$

→ $x + y - x \cdot y$

x	y	x oder y
1	1	
0	1	
1	0	
0	0	

Boolesche Ausdrücke – Der Datentyp bool

- Beispiel: ist ein Kunde kreditwürdig?
  - `bool credit = true / false;`

- Kriterien für Beispiel:
  - Kunde hat keinen negativen Eintrag im Schufa-Verzeichnis

    ```
    bool credit = schufa == false; //kürzerer Ausdruck möglich?
    ```

  - Kunde eröffnet Konto mit mindestens 5.000 €

    ```
    credit = capital >= 5000;
    ```

  - Kunde hat ein festes Einkommen

    ```
    credit = hasIncome;
    ```

Vergleichs-Operator	Beschreibung
==	Gleichheit
!=	Ungleichheit
<	Kleiner
<=	Kleiner gleich
>	Größer
>=	Größer gleich

Logischer Vergleich

- **Syntax:**
  - `<expression_1> == <expression_2>`
  - liefert `true` zurück, wenn die Werte von *<expression_1>* und *<expression_2>* übereinstimmen, sonst `false`
  - *<expression_1>* und <expression_2> müssen vom selben (einfachen) Datentyp sein
  - nicht zu verwechseln mit einfachem = für Zuweisungen

- **Analogie zur natürlichen Sprache:**
  - „wenn *<expression_1>* gleich *<expression_2>* ist“

<expression_1>	<expression_2>	<expression_1> == <expression_2>
42	42	
2	1	
`false`	`false`	
`"5"`	5	

Logisches UND

- **Syntax:**
  - *<expression_1>* & <expression_2>
  - liefert `true` zurück, wenn sowohl *<expression_1>* als auch *<expression_2>* wahr ist
  - sonst `false`

- **Analogie zur natürlichen Sprache:**
  - „wenn *<expression_1>* und *<expression_2>* stimmen“
  - „wenn sowohl *<expression_1>* als auch *<expression_2>* wahr ist“
  - „*<expression_1>* und *<expression_2>* sind beide richtig“

<expression_1>	<expression_2>	<expression_1> & <expression_2>
`true`	`true`	
`true`	`false`	
`false`	`true`	
`false`	`false`	

Logisches ODER

- **Syntax:**
  - `<expression_1> | <expression_2>`
  - liefert `true` zurück, wenn *<expression_1>* oder *<expression_2>* wahr ist (oder beides!)
  - sonst `false`

- **Analogie zur natürlichen Sprache:**
  - „wenn *<expression_1>* oder *<expression_2>* stimmen“
  - nicht verwechseln mit „entweder oder“ (exklusives oder)!

<expression_1>	<expression_1>	<expression_1> | <expression_2>
true	true	
true	false	
false	true	
false	false	

Logisches NICHT / Negations-Operator

- **Syntax:**
  - `!<expression>`
  - liefert `true` zurück, wenn `<expression>` falsch ist
  - sonst `false`

- **Analogie zur natürlichen Sprache:**
  - „wenn `<expression>` nicht stimmt“
  - „das Gegenteil von `<expression>`“

<expression>	!<expression>
true	false
false	true

- **Übung:**
  - Ein Kunde besitzt Kreditwürdigkeit, wenn er
    - keinen Schufa Eintrag hat oder ein Eigenkapital von mindestens 5.000 € besitzt
    - einen Schufa Eintrag hat und ein Eigenkapital von mindestens 5.000 € besitzt

Erweitert – Short Circuiting

- Bisher: `<expression_1> & <expression_2>`
  - es werden immer beide Ausdrücke geprüft/ausgeführt

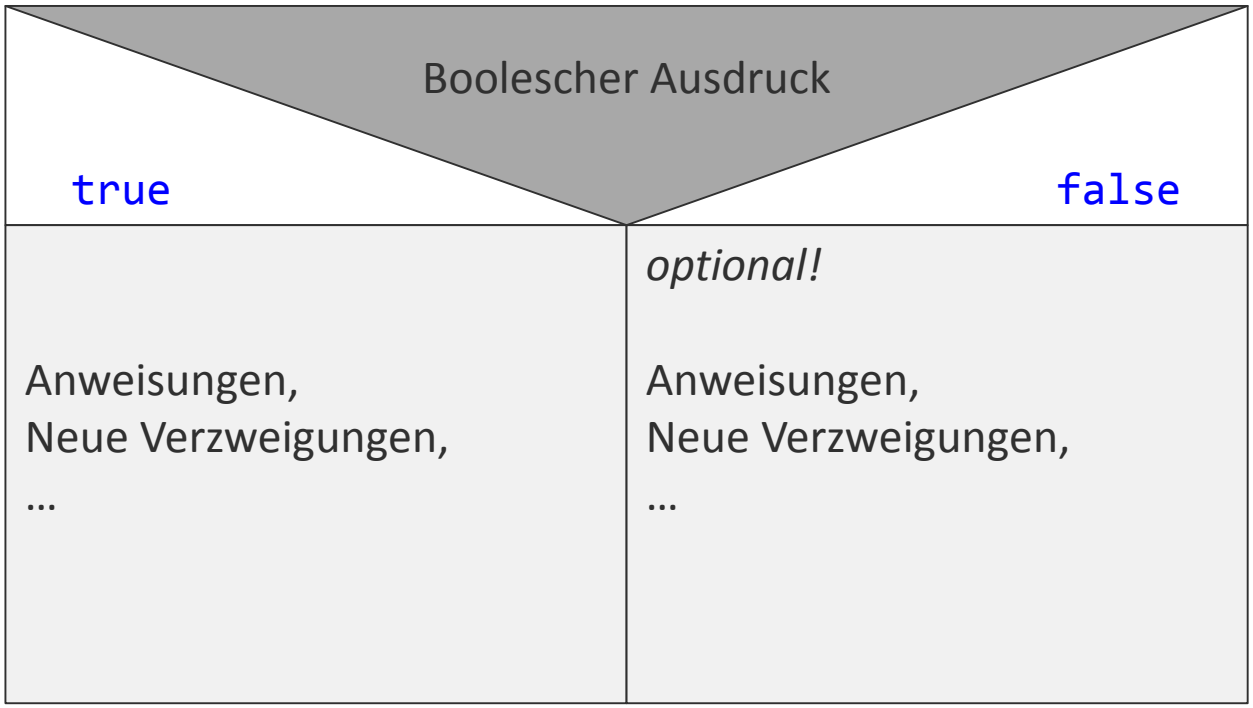
- Alternative: `<expression_1> && <expression_2>`
  - der rechte Ausdruck wird nur überprüft, wenn das Ergebnis nach Prüfung des linken Ausdrucks nicht bereits feststeht
    - `<expression_1>` = `false`: der gesamte Ausdruck kann nicht mehr richtig werden, der zweite Ausdruck wird ignoriert
    - `<expression_1>` = `true`: der zweite Ausdruck wird ausgewertet und erst anschließend das Ergebnis berechnet
- Analog bei `<expression_1> || <expression_2>`

- Der Ausdruck der eher zu einem Abbruch führt, sollte als erstes überprüft werden
- Beschleunigung des Programms bei vielen Vergleichen!

Agenda

- Boolesche Ausdrücke
- **Verzweigungen**
- Schleifen

Verzweigungen mit if-else



Verzweigungen – if then else

- if-else-Anweisung

```
if( <boolean_expression> ) { <expression_i> }
else { <expression_e> }
```

- Beispiele

```
if(day == "Sunday") {
    mood = "Hooray";
}
```

```
if(day == "Monday") {
    mood = "Hooray";
}
else {
    mood = "D'ouh.";
}
```

```
if(day == "Monday")
    mood = "Hooray";
else
    mood = "D'ouh.";
```

  - *<boolean expression>* wird auch *Bedingung* genannt
  - geschweifte Klammern sind nur bei mehreren Anweisungen notwendig!
  - `else` kann auch weggelassen werden
  - Visual Studio rückt Textbausteine (sinnvollerweise) automatisch ein

Verzweigungen – if then else (2)

- Verschachtelung
  - sowohl die if-Anweisungen als auch die else-Anweisungen können selbst weitere Verzweigungen beinhalten

```
if(day == "Monday")
    mood = "Hooray";
else if(day == "Sunday")
    mood = "Hmmm.";
else
    mood = "D'ouh.";
```

Erweitert – Bedingter Operator

- Bei bedingten An- und Zuweisungen ist nicht immer ein gesamter if-else-Block notwendig
- Es gibt eine Abkürzung
- Syntax: *<boolean_expression> ? <expression_i> : <expression_e>*

```
bool isRaining = true; // or false
string mood = isRaining ? "Oh noo." : "Get me out of here!";
Console.WriteLine(mood);
```

- Sowohl *<expression_i>* als auch <expression_e> müssen einen Wert gleichen Typs zurückgeben
- *<expression_e>* muss im Gegensatz zum else-Block immer existieren

Agenda

- Boolesche Ausdrücke
- Verzweigungen
- **Schleifen**
  - While-Schleifen
  - For-Schleifen
  - Verschachtelung
  - *Erweiterung: Do-While-Schleifen*
  - Schleifensteuerung

Schleifen – Motivation

- Es gibt Anweisungen, die in gleicher Form wiederholt ausgeführt werden müssen
- Beispiele:
  - Bank schreibt jedem Kunden am Ende jedes Monats Zinsen gut
  - ein Bewegungsmelder soll jede Sekunde prüfen, ob sich etwas verändert hat

- Es gibt bestimmte (for) und unbestimmte (while, do-while) Schleifen
  - bei bestimmten Schleifen ist vor Schleifeneintritt bekannt, wie viele Iterationen durchlaufen werden
  - bei unbestimmten Schleifen entscheidet sich erst während der Durchläufe, wie viele Iterationen es geben wird (boolesche Ausdrücke!)
  - bei allen Schleifentypen darauf achten, dass sie irgendwann terminieren!

- Ein Schleifendurchgang wird **Iteration** genannt

While-Schleifen

- Syntax:

```
while( <boolean_expression> ) { <expression> }
```

- Beispiel:

```
int a = 0;
while( a <= 10 ) {
   a++;
   Console.WriteLine("a = {0}", a);
}
```

- Ähnlich zu Verzweigungen wird *<expression>* nur ausgeführt, wenn die Iterations-Bedingung erfüllt ist
- Ist die Bedingung nicht erfüllt, wird die Schleife beendet und der nachfolgende Code wird ausgeführt

While-Schleifen – Übung

- Was wird hier ausgegeben?

```
int i = 5;
while( i > 0 )
   Console.WriteLine("i = {0}", i--);
Console.WriteLine("i = {0}", i);
```

```
int a = 1;
while( a <= 5 ) {
         i++;
         Console.WriteLine("a = {0}", a);
}
```

```
int i = 1;
while( i < 6 )
   Console.WriteLine("i = {0}", i);
   i += 2;
```

```
int i = 1;
while( i <= 5 ) {
   Console.WriteLine("i = {0}", i);
   i++;
}
```

For-Schleifen

- Syntax:

  `for ( <initialization>; <iteration_condition>; <iteration_command> )`
  `{ <expression> }`

- Vor der Schleifenausführung:
  - *<initialization>* ausführen

- Je Schleifendurchlauf:
  - **davor:** *<iteration_condition>* prüfen
    - *<iteration_condition>* erfüllt: *<expression>* ausführen
    - *<iteration_condition>* nicht erfüllt: Schleife beendet
  - **danach:** *<iteration_command>* ausführen

```
for (int i = 0; i < 10; i++) {
      Console.WriteLine("i = {0}", i);
}
```

For-Schleifen (2)

- Variablen, die in *<initialization>* deklariert und initialisiert werden, heißen **Schleifenvariablen** und sind nur innerhalb der Schleife gültig

- Wenn die Bedingung *<iteration_condition>* nicht spezifiziert ist, wird die Schleife endlos ausgeführt

For-Schleifen – Übung

- Was wird hier ausgegeben?

```
for (int i = 1; i <= 5; i++)
   Console.WriteLine("i");
```

```
for (int i = 5; i <= 2; i++)
         Console.WriteLine("i = {0}", i);
```

```
for (int i = 5; i >= 0; i--)
   Console.WriteLine("i = {0}", i);
```

```
for (int i = 0; i <= 10; i*=2)
   Console.WriteLine("i = {0}", i);
```

Verschachtelung

- Schleifen können ineinander verschachtelt werden
  - eine Schleife ist eine Anweisung, sie „klammert“ ihren Anweisungsblock

```
for (int i = 0; i <= 9; i++)
   for (int j = 0; j <= 9; j++)
      Console.WriteLine("{0}{1}", i, j);
```

Erweitert: Do-While-Schleifen

- Syntax:

```
do { <expression> }
while( <boolean_expression> );
```

- Beispiel:

```
int i = 1;
do {
   Console.WriteLine("i = {0}", i);
   i*=2;
} while(i <= 128);
```

- Iterationsblock *<expression>* wird **mindestens einmal** durchlaufen
- Erst **anschließend** wird die Iterationsbedingung getestet und es folgen bei positiver Prüfung weitere Iterationen
- Wichtig: Do-While-Schleifen brauchen ein abschließendes **Semikolon** (While-Schleifen nicht) – warum?

Erweitert: Schleifensteuerung

- `break;`
  - Schleifenausführung **komplett** beenden
  - in verschachtelten Schleifen bezieht sich `break` immer nur auf die aktuelle Schleife

- `continue;`
  - bricht die Durchführung der **aktuellen Iteration** ab
  - in verschachtelten Schleifen, bezieht sich `continue` immer nur auf die aktuelle Schleife

- Einsatzgebiet:
  - Schleifensteuerungskonstrukte kommen insbesondere bei absichtlich konstruierten **Endlosschleifen** zum Einsatz