Praktische Informatik 1

Imperative Programmierung und Objektorientierung

Blagoy Genov, Karsten Hölscher und Jan Peleska

Wintersemester 2011/2012

Universität Bremen

bedingte Anweisung

Schlüsselwort

zu testende
boolesche Bedingung

```
if (einen Test ausführen)
{
   Aktion(en) wenn der Test true ergab
}
else
{
   Aktion(en) wenn der Test false ergab
}
```

- der else-Zweig ist nicht zwingend!

if-else-Anweisung

- Allgemeine Form:

```
if ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
} else {
   Anweisung n+1;
   ...
   Anweisung n+m;
}
```

if-else-Anweisung

- *if* ohne *else*:

```
if ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
}
```

if-else-Anweisung

- Mehrere *if-else*-Anweisungen:

```
if ( boolescher Ausdruck 1 ) {
   Anweisung 1;
   ...
   Anweisung n;
} else if ( boolescher Ausdruck 2 ) {
   Anweisung n+1;
   ...
   Anweisung n+m;
} else {
   Anweisung n+m+1;
   ...
   Anweisung n+m+k;
}
```

boolescher Ausdruck

- Boolesche Ausdrücke haben nur zwei Werte:
  - wahr (`true`)
  - falsch (false)
- Triviale Beispiele für boolesche Ausdrücke sind **true** und **false**.

```
if ( true ) {
   ...
} else {
   ...
}
```

```
if ( false ) {
   ...
} else {
   ...
}
```

boolescher Ausdruck

- Boolesche Ausdrücke haben nur zwei Werte:
  - wahr (`true`)
  - falsch (false)
- Triviale Beispiele für boolesche Ausdrücke sind **true** und **false**.

```
if ( true ) {
   ...
} else {
  // nicht erreichbar
}
```

```
if ( false ) {
  // nicht erreichbar
} else {
   ...
}
```

boolescher Ausdruck

- Vergleichsoperatoren haben den Typ `boolean`

```
>  : int x int → boolean
<  : int x int → boolean
== : int x int → boolean
!= : int x int → boolean
<= : int x int → boolean
>= : int x int → boolean
```

if-else-Anweisung

```
public int doSomething(int a, int b) {
   if( a > b ) {
      a = a – b;
      if( a < b ) {
         b = a – b;
      } else {
         b = b – a;
      }
   }
   return b;
}
```

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
=> if( a > b ) {
       a = a - b;
       if( a < b ) {
          b = a - b;
       } else {
          b = b - a;
       }
   }
   return b;
```

7 > 3 : **true**

a = 7
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
if( a > b ) {
=> a = a – b;
   if( a < b ) {
      b = a – b;
   } else {
      b = b – a;
   }
}
return b;
```

a = 7 – 3 = 4

a = 7
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
if( a > b ) {
   a = a - b;
=> if( a < b ) {
      b = a - b;
   } else {
      b = b - a;
   }
}
return b;
```

4 < 3 : **false**

a = 4
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
if( a > b ) {
   a = a – b;
   if( a < b ) {
      b = a – b;
   } else {
   => b = b – a;
   }
}
return b;
```

b = 3 – 4 = -1

a = 4
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
   if( a > b ) {
      a = a - b;
      if( a < b ) {
         b = a - b;
      } else {
         b = b - a;
      }
   }
=> return b;
```

return -1

a = 4
b = -1

Logische Operatoren

- Logisches UND (&&)
  - `&& : boolean x boolean → boolean`
  - Gibt **true** zurück, wenn beide Argumente **true** sind.
A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false
  - Wenn A und B boolesche Ausdrücke sind, dann ist A && B auch ein boolescher Ausdruck.

Logische Operatoren

- Logisches ODER (||)
  - `|| : boolean x boolean → boolean`
  - Gibt **true** zurück, wenn mindestens ein Argument **true** ist.
A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false
  - Wenn A und B boolesche Ausdrücke sind, dann ist A || B auch ein boolescher Ausdruck.

Logische Operatoren

- Logisches NOT (**!**)
  - `! : boolean → boolean`
A	!A
true	false
false	true
  - Wenn A ein boolescher Ausdrücke ist, dann ist **!**A auch ein boolescher Ausdruck.

Logische Operatoren

- Verschiedene Ausdrücke können die gleiche Aussage darstellen:
  - a == b <=> !(a != b)
  - a == b <=> !((a > b) || (a < b))
  - a == b <=> (a >= b) && (a <= b)

if-else-Anweisung

```
public int doSomething(int a, int b) {
   if( a > b && a < 2*b ) {
      b = a – 2*b;
   } else if ( a >= 2*b ) {
      b = 2*b – a;
   }

   return b;
}
```

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
=> if( a > b && a < 2*b ) {
      b = a - 2*b;
   } else if ( a >= 2*b ) {
      b = 2*b - a;
   }

   return b;
```

7 > 3 : **true**

a = 7
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
=> if( a > b && a < 2*b ) {
      b = a - 2*b;
   } else if ( a >= 2*b ) {
      b = 2*b - a;
   }

   return b;
```

7 < 2*3 : **false**

a = 7
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
   if( a > b && a < 2*b ) {
      b = a – 2*b;
=> } else if ( a >= 2*b ) {
      b = 2*b – a;
   }

   return b;
```

7 > 2*3 : **true**

a = 7
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
if( a > b && a < 2*b ) {
   b = a – 2*b;
} else if ( a > 2*b ) {
=> b = 2*b – a;
}

return b;
```

b = 2*3 – 7 = -1

a = 7
b = 3

if-else-Anweisung

- Aufruf: doSomething(7, 3);

```
   if( a > b && a < 2*b ) {
      b = a - 2*b;
   } else if ( a > 2*b ) {
      b = 2*b - a;
   }

=> return b;
```

return -1

a = 7
b = -1

if-else-Anweisung

- Wie kann man den *else*-Zweig durch eine zweite *if*-Anweisung modellieren?

```
int get(int a) {
   int b;
   if( a == 0 ) {
      a = 1;
      b = 10;
   } else {
      a = 2;
      b = 11;
   }
   return b;
}
```

if-else-Anweisung

- Sind beide Methoden äquivalent?

```
int get(int a) {
   int b;
   if( a == 0 ) {
      a = 1;
      b = 10;
   } else {
      a = 2;
      b = 11;
   }
   return b;
}
```

```
int get(int a) {
   int b;
   if( a == 0 ) {
      a = 1;
      b = 10;
   }

   if ( a != 0) {
      a = 2;
      b = 11;
   }
   return b;
}
```

if-else-Anweisung

- Sind sie jetzt äquivalent?

```
int get(int a) {
   int b;
   if( a == 0 ) {
      a = 1;
      b = 10;
   } else {
      a = 2;
      b = 11;
   }
   return b;
}
```

```
int get(int a) {
   int b;
   int c = a;
   if( c == 0 ) {
      a = 1;
      b = 10;
   }

   if ( c != 0) {
      a = 2;
      b = 11;
   }
   return b;
}
```

switch-Anweisung

- Die **switch**-Anweisung ist eine Mehrfachverzweigung.
- Allgemeine Form:

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
  case ...
   ...
  default: Anweisung n;
}
```

switch-Anweisung

- Die **switch**-Anweisung ist eine Mehrfachverzweigung.
- Allgemeine Form:

Ein Ausdruck vom Typ **byte**, **char**, **short** oder **int**

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
  case ...
   ...
  default: Anweisung n;
}
```

switch-Anweisung

- Die **switch**-Anweisung ist eine Mehrfachverzweigung.
- Allgemeine Form:

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
  case ...
   ...
  default: Anweisung n;
}
```

Ein konstanter Wert, z.B. 0, -32 oder 42

switch-Anweisung

- Semantik:

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
    ...
  default: Anweisung n;
}
```

switch-Anweisung

- Semantik:

  (1) Werte den *Ausdruck* aus.

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
    ...
  default: Anweisung n;
}
```

switch-Anweisung

- Semantik:

  (1) Werte den *Ausdruck* aus.

  (2) Wenn ein **case**-Zweig zum Ergebniswert definiert ist, springe ihn an.

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
    ...
  default: Anweisung n;
}
```

switch-Anweisung

- Semantik:
  - (1) Werte den *Ausdruck* aus.
  - (2) Wenn ein **case**-Zweig zum Ergebniswert definiert ist, springe ihn an.
  - (3) Ansonsten führe den **default**-Zweig aus.

```
switch( ganzzahliger Ausdruck ) {
  case Wert 1 : Anweisung 1; break;
  case Wert 2 : Anweisung 2; break;
  case Wert 3 : Anweisung 3; break;
    ...
  default: Anweisung n;
}
```

switch-Anweisung

```
public int doSomething(int a){

   switch( a ) {
      case 0: a = 1; break;
      case 1: a = a+2; break;
      case 2: a = 2*a; break;
      default: a = 0;
   }

   return a;
}
```

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
=> case 0: a = 1; break;
   case 1: a = a+2; break;
   case 2: a = 2*a; break;
   default: a = 0;
}

return a;
```

a == 0 : **false**

a=2

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
   case 0: a = 1; break;
=> case 1: a = a+2; break;
   case 2: a = 2*a; break;
   default: a = 0;
}

return a;
```

a == 1 : **false**

a=2

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
   case 0: a = 1; break;
   case 1: a = a+2; break;
=> case 2: a = 2*a; break;
   default: a = 0;
}

return a;
```

a == 2 : **true**

a=2

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
   case 0: a = 1; break;
   case 1: a = a+2; break;
=> case 2: a = 2*a; break;
   default: a = 0;
}

return a;
```

a = 2*2 = 4

a=2

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
   case 0: a = 1; break;
   case 1: a = a+2; break;
=> case 2: a = 2*a; break;
   default: a = 0;
}

return a;
```

a=4

switch-Anweisung

- Aufruf: doSomething(2);

```
   switch( a ) {
      case 0: a = 1; break;
      case 1: a = a+2; break;
      case 2: a = 2*a; break;
      default: a = 0;
   }

=> return a;
```

return 4

a=4

switch-Anweisung

- Warum ist break wichtig?

```
public int doSomething(int a){

   switch( a ) {
      case 2: a = 2*a;
      default: a = 0;
   }

   return a;
}
```

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
=> case 2: a = 2*a;
   default: a = 0;
}

return a;
```

a == 2 : **true**

a=2

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
=> case 2: a = 2*a;
   default: a = 0;
}

return a;
```

a = 2*2 = 4

a=2

switch-Anweisung

- Aufruf: doSomething(2);

```
switch( a ) {
   case 2: a = 2*a;
=> default: a = 0;
}

return a;
```

a = 0

a=4

switch-Anweisung

- Aufruf: doSomething(2);

```
   switch( a ) {
      case 2: a = 2*a;
      default: a = 0;
   }

=> return a;
```

return 0

a=0

while-Schleife

- Allgemeine Form:

```
while ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
}
```

while-Schleife

- Semantik:

```
while ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
}
Anweisung n+1;
```

while-Schleife

- Semantik:

  (1) Werte den *Ausdruck* aus.

```
while ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
}
Anweisung n+1;
```

while-Schleife

- Semantik:

  (1) Werte den *Ausdruck* aus.

  (2) Wenn das Ergebnis **true** ist, führe Anweisungen *1* bis *n* aus und kehre zu (1) zurück.

```
while ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
}
Anweisung n+1;
```

while-Schleife

- Semantik:

  (1) Werte den *Ausdruck* aus.

  (2) Wenn das Ergebnis **true** ist, führe Anweisungen *1* bis *n* aus und kehre zu (1) zurück.

  (3) Wenn das Ergebnis **false** ist, führe Anweisung *n+1* aus.

```
while ( boolescher Ausdruck ) {
   Anweisung 1;
   ...
   Anweisung n;
}
Anweisung n+1;
```

while-Schleife

```
public int doSomething(int a, int b) {

   while ( a > b ) {
      a = a – b;
   }
   return a;
}
```

while-Schleife

- Aufruf: doSomething(7, 4);

```
=> while ( a > b ) {
       a = a - b;
   }
   return a;
```

7 > 4 : **true**

a = 7
b = 4

while-Schleife

- Aufruf: doSomething(7, 4);

```
while ( a > b ) {
=> a = a – b;
}
return a;
```

a = 7 – 4 = 3

a = 7
b = 4

while-Schleife

- Aufruf: doSomething(7, 4);

```
=> while ( a > b ) {
       a = a - b;
   }
   return a;
```

3 > 4 : **false**

a = 3
b = 4

while-Schleife

- Aufruf: doSomething(7, 4);

```
   while ( a > b ) {
      a = a - b;
   }
=> return a;
```

return 3

a = 3
b = 4

do-while-Schleife

- Allgemeine Form:

```
do {
   Anweisung 1;
   ...
   Anweisung n;
} while ( boolescher Ausdruck );
```

do-while-Schleife

- Semantik:

```
do {
   Anweisung 1;
   ...
   Anweisung n;
} while ( boolescher Ausdruck );
Anweisung n+1;
```

do-while-Schleife

- Semantik:

  (1) Führe *Anweisungen 1* bis *n* aus.

```
do {
   Anweisung 1;
   ...
   Anweisung n;
} while ( boolescher Ausdruck );
Anweisung n+1;
```

do-while-Schleife

- Semantik:
  - (1) Führe *Anweisungen 1* bis *n* aus.
  - (2) Werte den *Ausdruck* aus.

```
do {
   Anweisung 1;
   ...
   Anweisung n;
} while ( boolescher Ausdruck );
Anweisung n+1;
```

do-while-Schleife

- Semantik:
  (1) Führe *Anweisungen 1* bis *n* aus.
  (2) Werte den *Ausdruck* aus.
  (3) Wenn das Ergebnis **true** ist, kehre zu (1) zurück.

```
do {
   Anweisung 1;
   ...
   Anweisung n;
} while ( boolescher Ausdruck );
Anweisung n+1;
```

do-while-Schleife

- Semantik:
  (1) Führe *Anweisungen 1* bis *n* aus.
  (2) Werte den *Ausdruck* aus.
  (3) Wenn das Ergebnis **true** ist, kehre zu (1) zurück.
  (4) Ansonsten führe *Anweisung n+1* aus.

```
do {
   Anweisung 1;
   ...
   Anweisung n;
} while ( boolescher Ausdruck );
Anweisung n+1;
```

do-while-Schleife

```
public int doSomething(int a, int b) {

   do {
      a = a – b;
   } while ( a > b );

   return a;
}
```

do-while-Schleife

- Aufruf: doSomething(7, 4);

```
do {
=> a = a - b;
} while ( a > b );

return a;
```

a = 7 – 4 = 3

a = 7
b = 4

do-while-Schleife

- Aufruf: doSomething(7, 4);

```
   do {
      a = a - b;
=> } while ( a > b );

   return a;
```

3 > 4 : **false**

a = 3
b = 4

do-while-Schleife

- Aufruf: doSomething(7, 4);

```
   do {
      a = a - b;
   } while ( a > b );

=> return a;
```

return 3

a = 3
b = 4

for-Schleife

- Allgemeine Form:

```
for( Initialisierungsanweisungen;
     Boolescher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung n;
}
```

for-Schleife

- Semantik:

  (1) Führe die *Initialisierungsanweisungen* aus.

```
for( Initialisierungsanweisungen;
     Boolscher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung N;
}
```

for-Schleife

- Semantik:
  - (1) Führe die *Initialisierungsanweisungen* aus.
  - (2) Werte den *Ausdruck* aus.

```
for( Initialisierungsanweisungen;
     Boolscher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung N;
}
```

for-Schleife

- Semantik:
  (1) Führe die *Initialisierungsanweisungen* aus.
  (2) Werte den *Ausdruck* aus.
  (3) Wenn das Ergebnis **true** ist,
    i. Führe *Anweisungen 1* bis *n* aus.

```
for( Initialisierungsanweisungen;
     Boolscher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung N;
}
```

for-Schleife

- Semantik:
  (1) Führe die *Initialisierungsanweisungen* aus.
  (2) Werte den *Ausdruck* aus.
  (3) Wenn das Ergebnis **true** ist,
    i. Führe *Anweisungen 1* bis *n* aus.
    ii. Führe die *Aktualisierungsanweisungen* aus.

```
for( Initialisierungsanweisungen;
     Boolscher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung N;
}
```

for-Schleife

- Semantik:
  - (1) Führe die *Initialisierungsanweisungen* aus.
  - (2) Werte den *Ausdruck* aus.
  - (3) Wenn das Ergebnis **true** ist,
    - i. Führe *Anweisungen 1* bis *n* aus.
    - ii. Führe die *Aktualisierungsanweisungen* aus.
    - iii. Kehre zu (2) zurück.

```
for( Initialisierungsanweisungen;
     Boolscher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung N;
}
```

for-Schleife

- Semantik:
  - (1) Führe die *Initialisierungsanweisungen* aus.
  - (2) Werte den *Ausdruck* aus.
  - (3) Wenn das Ergebnis **false** ist, führe *Anweisung n+1* aus.

```
for( Initialisierungsanweisungen;
     Boolscher Ausdruck;
     Aktualisierungsanweisungen) {
   Anweisung 1;
   ...
   Anweisung n;
}
Anweisung n+1;
```

for-Schleife

```
public int doSomething(int a, int b) {

   for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
}
```

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

i = 0

a=7
b=3
i=0

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

0 < 3 : **true**

a=7
b=3
i=0

for-Schleife

- Aufruf: doSomething(7, 3);

```
for(int i=0; i<b; i++) {
=> a = a + i*b;
}

return a;
```

a = 7 + 0*3 = 7

a=7
b=3
i=0

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

i = 0 + 1 = 1

a=7
b=3
i=0

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

1 < 3 : **true**

a=7
b=3
i=1

for-Schleife

- Aufruf: doSomething(7, 3);

```
for(int i=0; i<b; i++) {
=> a = a + i*b;
}

return a;
```

a = 7 + 1*3 = 10

a=7
b=3
i=1

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

i = 1 + 1 = 2

a=10
b=3
i=1

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

2 < 3 : **true**

a=10
b=3
i=2

for-Schleife

- Aufruf: doSomething(7, 3);

```
for(int i=0; i<b; i++) {
=> a = a + i*b;
}

return a;
```

a = 10 + 2*3 = 16

a=10
b=3
i=2

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

i = 2 + 1 = 3

a=16
b=3
i=2

for-Schleife

- Aufruf: doSomething(7, 3);

```
=> for(int i=0; i<b; i++) {
      a = a + i*b;
   }

   return a;
```

3 < 3 : **false**

a=16
b=3
i=3

for-Schleife

- Aufruf: doSomething(7, 3);

```
   for(int i=0; i<b; i++) {
      a = a + i*b;
   }

=> return a;
```

return 16

a=16
b=3