6. Operatoren

Tabellarische Übersicht aller relevanten Operatoren

Operatoren: Tabelle

Beschreibung	Operator	Stelligkeit	Präzedenz	Assoziativität
Objekt-Member Zugriff	`.`	2	16	links
Array Zugriff	`[ ]`	2	16	links
Methodenaufruf	`( )`	2	16	links
Postfix Inkrement/Dekrement	`++ --`	1	15	links
Präfix Inkrement/Dekrement	`++ --`	1	14	rechts
Plus, Minus, Logisches Nicht	`+ - !`	1	14	rechts
Typcast	`( )`	1	13	rechts
Objekterstellung	`new`	1	13	rechts
Multiplikativ	`* / %`	2	12	links
Additiv	`+ -`	2	11	links
Stringkonkatination	`+`	2	11	links
Vergleiche	`< <= > >=`	2	9	links
Typvergleich	`instanceof`	2	9	links
(Nicht-)Gleichheit	`== !=`	2	8	links
Logisches Und	`&&`	2	4	links
Logisches Oder	`||`	2	3	links
Konditional	`? :`	3	2	rechts
Zuweisungen	`= += -= *= /= %=`	2	1	rechts

Operatoren: Tabelle - Erklärungen

- Die Stelligkeit gibt die Anzahl der Operanden an
- Eine höhere Präzedenz bedeutet stärkere Bindung
- Bei gleicher Präzedenz wird gemäss der Assoziativität ausgewertet

7. Sicheres Programmieren: Assertions

Assertions

Fehlerquellen

- Fehler, die der Compiler findet:
  syntaktische und manche semantische Fehler

Fehlerquellen

- Fehler, die der Compiler findet:
  syntaktische und manche semantische Fehler
- Fehler, die der Compiler nicht findet:
  Laufzeitfehler (immer semantisch)

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

≫ It's not a bug, it's a feature !!≪

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

2. Überprüfe an vielen kritischen Stellen, ob das Programm auf dem richtigen Weg ist!

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

2. Überprüfe an vielen kritischen Stellen, ob das Programm auf dem richtigen Weg ist!
3. Hinterfrage auch das (scheinbar) Offensichtliche, es könnte sich ein simpler Tippfehler eingeschlichen haben!

Gegen Laufzeitfehler: *Assertions*

```
assert expr;
```

- wirft einen Fehler und hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist

Gegen Laufzeitfehler: *Assertions*

```
assert expr;
```

- wirft einen Fehler und hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist
- kann beim Starten der Java-VM ein- oder ausgeschaltet werden

Gegen Laufzeitfehler: *Assertions*

```
assert expr;
```

- wirft einen Fehler und hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist
- kann beim Starten der Java-VM ein- oder ausgeschaltet werden

Alternativ: `assert expr : expr2;`

- Wenn die Assertion nicht hält wird zusätzlich der Wert von `expr2` in der Konsole angezeigt

Div-Mod Identität

`a/b * b + a%b == a`

Überprüfe, ob das Programm auf dem richtigen Weg ist...

```
Out.println("Dividend a =? ");
int a = In.readInt();

Out.println("Divisor b =? ");
int b = In.readInt();

// check input
assert b != 0 : "User error: b must not be zero";
```

Eingabe der Argumente für die Berechnung

Div-Mod Identität

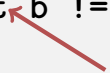
`a/b * b + a%b == a`

Überprüfe, ob das Programm auf dem richtigen Weg ist...

```
Out.println("Dividend a =? ");
int a = In.readInt();

Out.println("Divisor b =? ");
int b = In.readInt();

// check input
assert b != 0 : "User error: b must not be zero";
```



Vorbedingung für die weitere Berechnung

Div-Mod Identität

`a/b * b + a%b == a`

. . . und hinterfrage das Offensichtliche!

```
// check input
assert b != 0 : "User error: b must not be zero";

// compute result
int div = a / b;
int mod = a % b;

// check result
assert div * b + mod == a;
...
```

⟵ Div-Mod Identität

8. Kontrollanweisungen

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke, Sichtbarkeit, Lokale Variablen, While-Anweisung, Do-Anweisung, Sprunganweisungen

Anweisungen (Statements)

Eine Anweisung ist . . .

- vergleichbar mit einem Satz in der natürlichen Sprache
- eine komplette Ausführungseinheit
- immer mit einem *Semikolon* abgeschlossen

Beispiel

```
f = 9f * celsius / 5 + 32 ;
```

Anweisungenarten

Gültige Anweisungen sind:

- Deklarationsanweisung
- Wertzuweisungen
- Inkrement / Dekrement Ausdrücke
- Methodenaufrufe
- Objekterzeugungs-Ausdrücke
- Nullanweisung

Anweisungenarten

Beispiele

```
float aValue;
aValue = 8933.234;
aValue++;
Out.println(aValue);
new Student();
;
```

Blöcke

Ein Block ist . . .

- eine Gruppe von Anweisungen
- überall erlaubt wo Anweisungen erlaubt sind
- durch geschweiften Klammern markiert

```
{
  statement1
  statement2
  ⋮
}
```

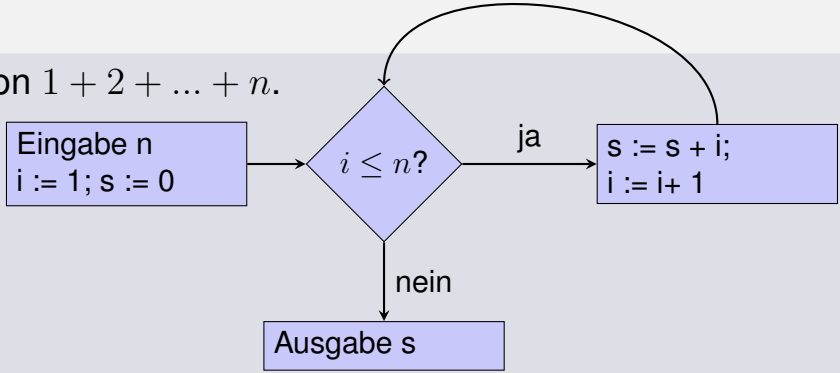
Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen” und „Sprünge”.

Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen” und „Sprünge”.

Berechung von $1 + 2 + ... + n$.



Auswahlanweisungen

realisieren Verzweigungen

- `if` Anweisung
- `if-else` Anweisung

`if`-Anweisung

`if` (*condition*)
    statement

`if`-Anweisung

`if` (*condition*)
    statement

```
int a = In.readInt();
if (a % 2 == 0) {
    Out.println("even");
}
```

`if`-Anweisung

`if` (*condition*)
    statement

Ist *condition* wahr, dann wird *statement* ausgeführt.

```
int a = In.readInt();
if (a % 2 == 0) {
    Out.println("even");
}
```

`if`-Anweisung

`if` (*condition*)
    statement

```
int a = In.readInt();
if (a % 2 == 0) {
    Out.println("even");
}
```

- *statement*: beliebige Anweisung (*Rumpf* der `if`-Anweisung)
- *condition*: Ausdruck vom Typ `boolean`

`if-else`-Anweisung

```
if ( condition )
    statement1
else
    statement2
```

if-else-Anweisung

```
if ( condition )
    statement1
else
    statement2
```

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

`if-else`-Anweisung

`if` (*condition*)
    statement1
else
    statement2

Ist *condition* wahr, so wird *statement1* ausgeführt, andernfalls wird *statement2* ausgeführt.

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

`if-else`-Anweisung

`if` (*condition*)
    statement1
else
    statement2

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

- *condition*: Ausdruck vom Typ `boolean`
- *statement1*: *Rumpf* des `if`-Zweiges
- *statement2*: *Rumpf* des `else`-Zweiges

Layout!

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

Layout!

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

Iterationsanweisungen

realisieren „Schleifen”:

- `for`-Anweisung
- `while`-Anweisung
- `do`-Anweisung

Berechne $1 + 2 + \ldots + n$

```
// input
Out.print("Compute the sum 1+...+n for n=?");
int n = In.readInt();

// computation of sum_{i=1}^n i
int s = 0;
for (int i = 1; i <= n; ++i){
    s += i;
}

// output
Out.println("1+...+" + n + " = " + s);
```

Berechne $1 + 2 + \ldots + n$

```
// input
Out.print("Compute the sum 1+...+n for n=?");
int n = In.readInt();

// computation of sum_{i=1}^n i
int s = 0;
for (int i = 1; i <= n; ++i){
    s += i;
}

// output
Out.println("1+...+" + n + " = " + s);
```

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`	`s`

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`	`s`
`i==1`	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	`i <= 2?`	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`		

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	`i <= 2`?	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`		

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`	`i <= 2`?	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`	falsch	

`for`-Anweisung am Beispiel

```
for ( int i=1;  i <= n ; ++i ){
    s += i;
}
```

Annahmen: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`	falsch	
		`s == 3`

`for`-Anweisung: Syntax

`for` (*init statement* *condition* `;` *expression*)
statement

`for`-Anweisung: Syntax

`for` (*init statement* *condition* `;` *expression*)
statement

Deklarationsanweisung

auch möglich: Ausdrucksanweisung, Nullanweisung

```
for ( int i=1;  i <= n ; ++i ){
    s += i; // Rumpf
}
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* `;` *expression*)
statement

Ausdruck vom Typ `boolean`

```
for ( int i=1; i <= n ; ++i ){
    s += i; // Rumpf
}
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* `;` *expression*)
statement

Ausdruck vom Typ `int`

```
for ( int i=1;  i <= n ; ++i ){
    s += i; // Rumpf
}
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* `;` *expression*)
statement

Ausdrucksanweisung

```
for ( int i=1;  i <= n ; ++i ){
    s += i; // Rumpf
}
```

Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- Diese Summe kann vorwärts oder rückwärts berechnet werden, was mathematisch gesehen natürlich äquivalent ist.

Harmonische Zahlen (Fliesskomma Regel 2)

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

Harmonische Zahlen (Fliesskomma Regel 2)

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

Eingabe: **10’000’000**

Vorwärts: **15.4037**

Rückwärts: **16.686**

Harmonische Zahlen (Fliesskomma Regel 2)

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

Eingabe: **100’000’000**

Vorwärts: **15.4037**

Rückwärts: **18.8079**

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.

`for`-Anweisung: Terminierung

```
for (int i = 1; i <= n; ++i){
    s += i;
}
```

Hier und meistens:

- *expression* ändert einen Wert, der in *condition* vorkommt.

`for`-Anweisung: Terminierung

```
for (int i = 1; i <= n; ++i){
    s += i;
}
```

Hier und meistens:

- Nach endlich vielen Iterationen wird *condition* falsch: *Terminierung*.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

  - Die *leere condition* ist wahr.
  - Die *leere expression* hat keinen Effekt.
  - Die *Nullanweisung* hat keinen Effekt.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

  - Die *leere condition* ist wahr.
  - Die *leere expression* hat keinen Effekt.
  - Die *Nullanweisung* hat keinen Effekt.

- ... aber nicht automatisch zu erkennen.

```
for ( e; v; e) r;
```

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein Java Programm, das für jedes Java- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

[5] Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein Java Programm, das für jedes Java- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

Das heisst, die Korrektheit von Programmen kann *nicht* automatisch überprüft werden.[5]

[5] Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
int d;
for (d=2; n%d != 0; ++d);
```

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
int d;
for (d=2; n%d != 0; ++d);
```

(Rumpf ist die Null-Anweisung)

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
int d;
for (d=2; n%d != 0; ++d);
```

- Beobachtung 1: Nach der `for`-Anweisung gilt $d \leq n$.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
int d;
for (d=2; n%d != 0; ++d);
```

- Beobachtung 2: n ist Primzahl genau wenn am Ende $d = n$.

Rekapitulation: Blöcke

- Beispiel: Rumpf der main Funktion

```
public static void main(String[] args) {
    ...
}
```

Rekapitulation: Blöcke

- Beispiel: Schleifenrumpf

```
for (int i = 1; i <= n; ++i) {
    s += i;
    Out.println("partial sum is " + s);
}
```

Rekapitulation: Blöcke

- Beispiel: if / else

```
if (d < n) { // d is a divisor of n in {2,...,n−1}
    Out.println(n + " = " + d + " * " + n / d);
} else {
    assert (d == n);
    Out.println(n + " is prime");
}
```

Sichtbarkeit

Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar”.

```
public static void main(String[] args)
{
    {
        int i = 2;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

Sichtbarkeit

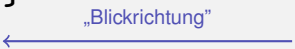
Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar”.

```
public static void main(String[] args)
{
    {
        int i = 2;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

main block

block

„Blickrichtung”



Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

```
public static void main(String[] args) {
{
    for (int i = 0; i < 10; ++i){
        s += i;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

```
public static void main(String[] args) {
{
    for (int i = 0; i < 10; ++i){
        s += i;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

block

Gültigkeitsbereich

Im Block

```
{
    int i = 2;
    ...
}
```

Im Funktionsrumpf

```
void main(String[] args) {
    int i = 2;
    ...
}
```

In Kontrollanweisung

```
for ( int i = 0; i < 10; ++i) {s += i; ... }
```

Gültigkeitsbereich

Im Block

```
{
    int i = 2;
    ...
}
```

scope

Im Funktionsrumpf

```
void main(String[] args) {
    int i = 2;
    ...
}
```

scope

In Kontrollanweisung

```
for ( int i = 0; i < 10; ++i) {s += i; ... }
```

scope

Lokale Variablen

```
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs
        int k = 2;
        Out.println(--k); // outputs
    }
}
```

Lokale Variablen

```
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs 6, 7, 8, 9, 10
        int k = 2;
        Out.println(--k); // outputs 1, 1, 1, 1, 1
    }
}
```

Lokale Variablen

```
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs 6, 7, 8, 9, 10
        int k = 2;
        Out.println(--k); // outputs 1, 1, 1, 1, 1
    }
}
```

Lokale Variablen (Deklaration in einem Block) haben *automatische Speicherdauer*.

`while` Anweisung

```
while ( condition )
  statement
```

`while` Anweisung

```
while ( condition )
  statement
```

ist äquivalent zu

```
for ( ; condition ; )
  statement
```

`while`-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife”)

```
for (int i = 1; i <= n; ++i){
    s += i;
}
```

`while`-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife”)

```
for (int i = 1; i <= n; ++i){
    s += i;
}
```

- Falls der Fortschritt nicht so einfach ist, kann `while` besser lesbar sein.

Beispiel: Die Collatz-Folge $(n \in \mathbb{N})$

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1}+1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1}+1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (Repetition bei 1)

Die Collatz-Folge in Java

```
// Input
Out.println("Compute Collatz sequence, n =? ");
int n = In.readInt();

// Iteration
while (n > 1) {       // stop when 1 reached
    if (n % 2 == 0) { // n is even
        n = n / 2;
    } else {              // n is odd
        n = 3 * n + 1;
    }
    Out.print(n + " ");
}
```

Die Collatz-Folge in Java

```
n = 27:
82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233,
700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336,
668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276,
638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429,
7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20,
10, 5, 16, 8, 4, 2, 1
```

`do` Anweisung

```
do
  statement
while ( expression );
```

`do` Anweisung

```
do
  statement
while ( expression );
```

ist äquivalent zu

```
statement
while ( expression )
  statement
```

Beispiel Taschenrechner

Summiere ganze Zahlen (bei 0 ist Schluss):

```
int a;    // next input value
int s = 0; // sum of values so far
do {
    Out.print("next number =? ");
    a = In.readInt();
    s += a;
    Out.println("sum = " + s);
} while (a != 0);
```

Sprunganweisungen

- `break`
- `continue`

Taschenrechner mit `break`

Summiere ganze Zahlen (bei 0 ist Schluss):

```
int a;
int s = 0;
do {
    Out.print("next number =? ");
    a = In.readInt();
    // irrelevant in letzter Iteration:
    s += a;
    Out.println("sum = " + s);
}  while (a != 0);
```

Taschenrechner mit `break`

Unterdrücke irrelevante Addition von 0:

```
int a;
int s = 0;
do {
    Out.print("next number =? ");
    a = In.readInt();
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    Out.println("sum = " + s);
} while (a != 0)
```

Taschenrechner mit `break`

Äquivalent und noch etwas einfacher:

```
int a;
int s = 0;
for (;;) {
    Out.print("next number =? ");
    a = In.readInt();
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    Out.println("sum = " + s);
}
```

Taschenrechner mit `continue`

Ignoriere alle negativen Eingaben:

```
for (;;)
{
    Out.print("next number =? ");
    a = In.readInt();
    if (a < 0) continue; // springe zu }
    if (a == 0) break;
    s += a;
    Out.println("sum = " + s);
}
```

Kontrollfluss

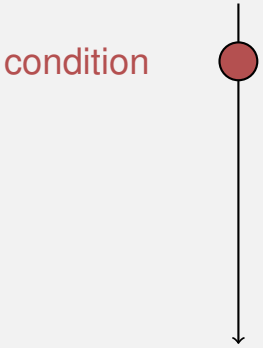
Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...

Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ...ausser in Auswahl- und Kontrollanweisungen

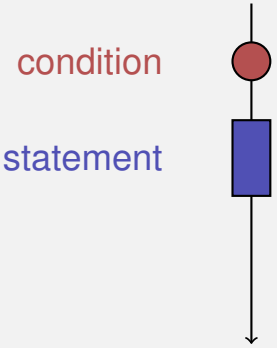


`if` (*condition*)
    statement

Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ...ausser in Auswahl- und Kontrollanweisungen

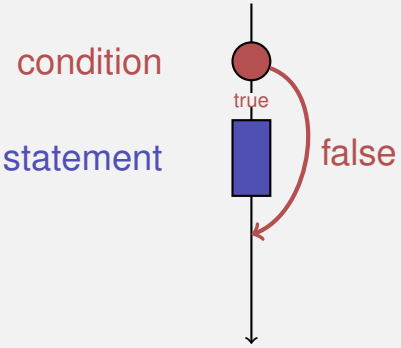


`if` (*condition*)
statement

Kontrollfluss

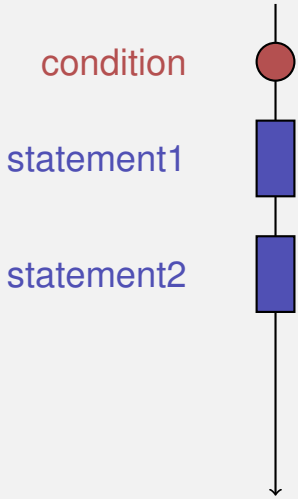
Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ...ausser in Auswahl- und Kontrollanweisungen



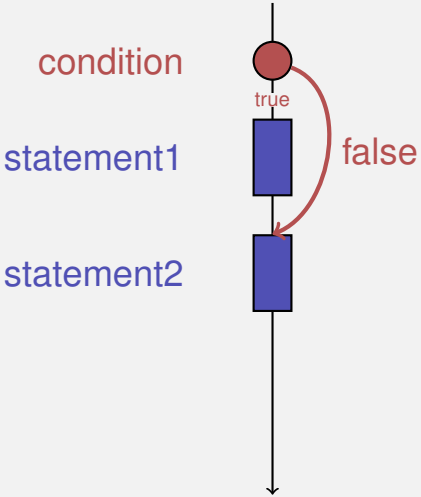
`if` (*condition*)
statement

Kontrollfluss `if else`



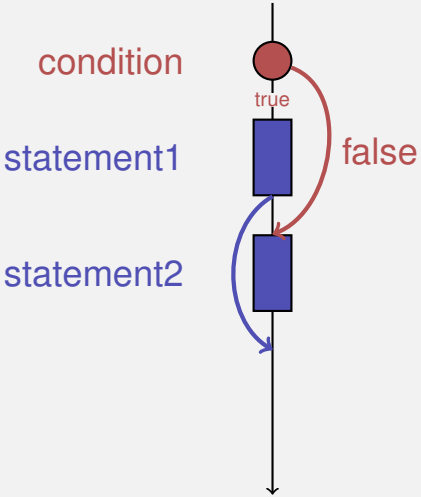

```
if ( condition )
    statement1
else
    statement2
```

Kontrollfluss `if else`




```
if ( condition )
    statement1
else
    statement2
```

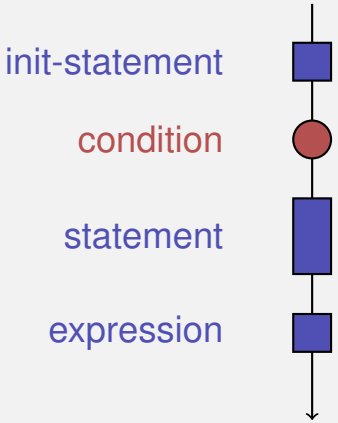
Kontrollfluss `if else`




```
if ( condition )
    statement1
else
    statement2
```

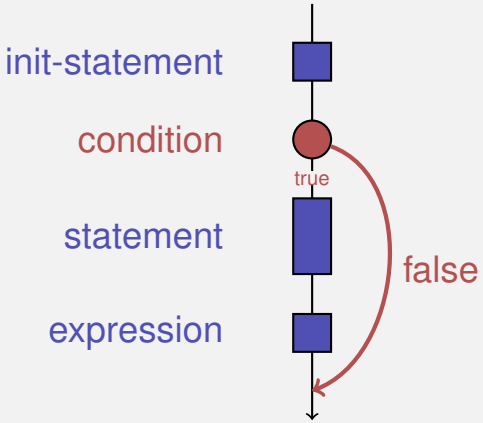
Kontrollfluss `for`

`for` (*init statement* *condition* `;` *expression*)
statement



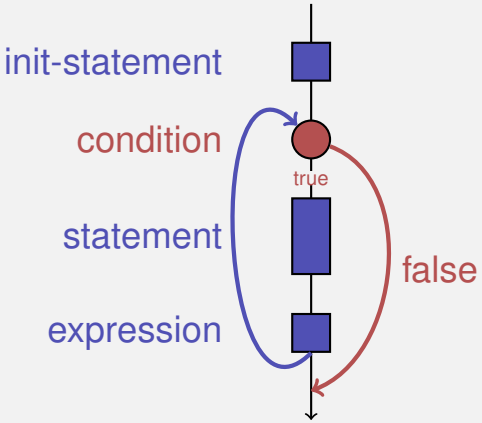
Kontrollfluss `for`

`for` (*init statement* *condition* `;` *expression*)
statement

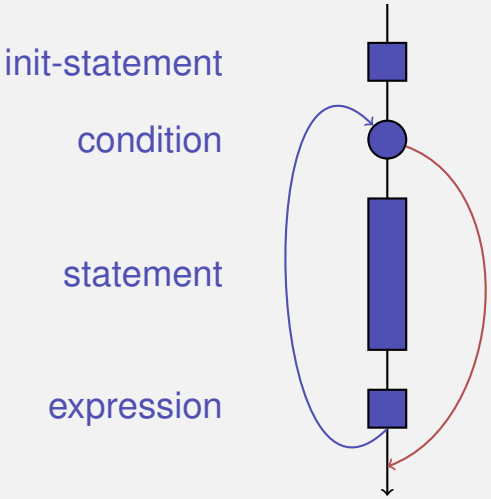


Kontrollfluss `for`

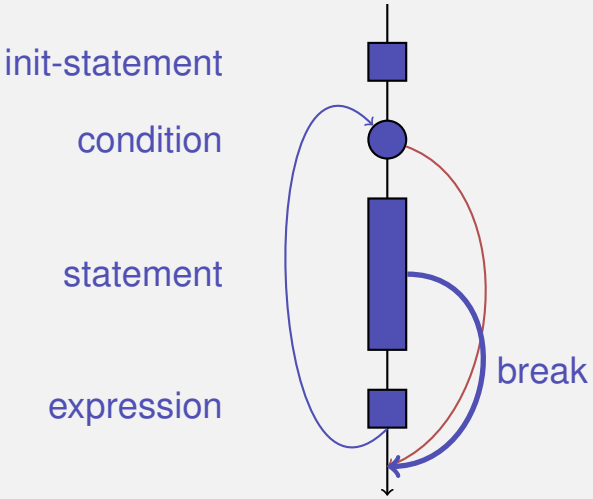
`for` (*init statement* *condition* `;` *expression*)
statement



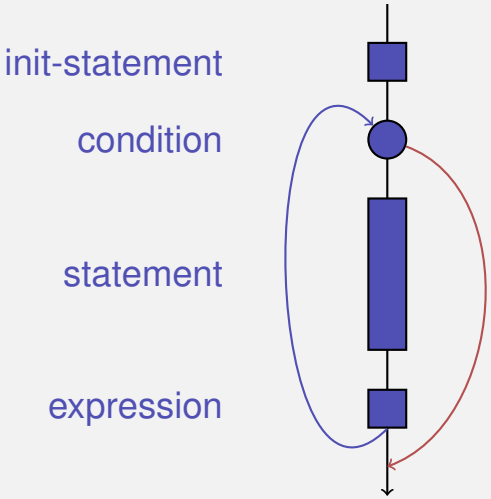
Kontrollfluss `break` und `continue` in for



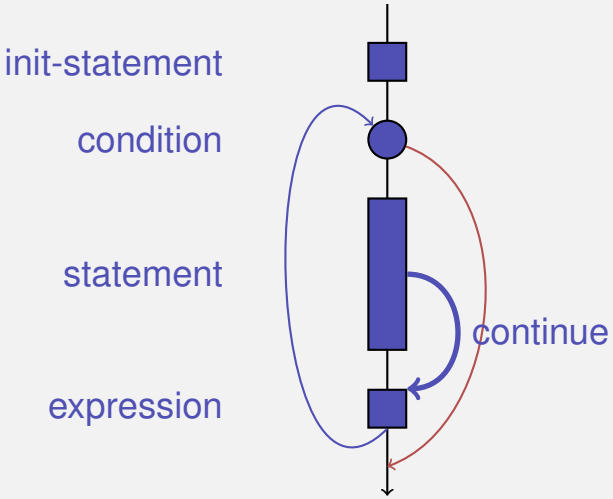
Kontrollfluss `break` und `continue` in for



Kontrollfluss `break` und `continue` in for



Kontrollfluss `break` und `continue` in for



Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Ziele sind oft nicht gleichzeitig erreichbar.

Ungerade Zahlen in $\{0, \ldots, 100\}$

Erster (korrekter) Versuch:

```
for (int i = 0; i < 100; ++i) {
    if (i % 2 == 0){
        continue;
    }
    Out.println(i);
}
```

Ungerade Zahlen in $\{0, \ldots, 100\}$

Weniger Anweisungen, *weniger* Zeilen:

```
for (int i = 0; i < 100; ++i) {
    if (i % 2 != 0){
        Out.println(i);
    }
}
```

Ungerade Zahlen in $\{0, \ldots, 100\}$

Weniger Anweisungen, *einfacherer* Kontrollfluss:

```
for (int i = 1; i < 100; i += 2) {
    Out.println(i);
}
```

Ungerade Zahlen in $\{0, \ldots, 100\}$

Weniger Anweisungen, *einfacherer* Kontrollfluss:

```
for (int i = 1; i < 100; i += 2) {
    Out.println(i);
}
```

Das ist hier die “richtige” Iterationsanweisung!

Die `switch`-Anweisung

`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

```
int Note;
...
switch (Note) {
    case 6:
        Out.print("super!");
        break;
    case 5:
        Out.print("gut!");
        break;
    case 4:
        Out.print("ok!");
        break;
    default:
        Out.print("schade.");
}
```

Die `switch`-Anweisung

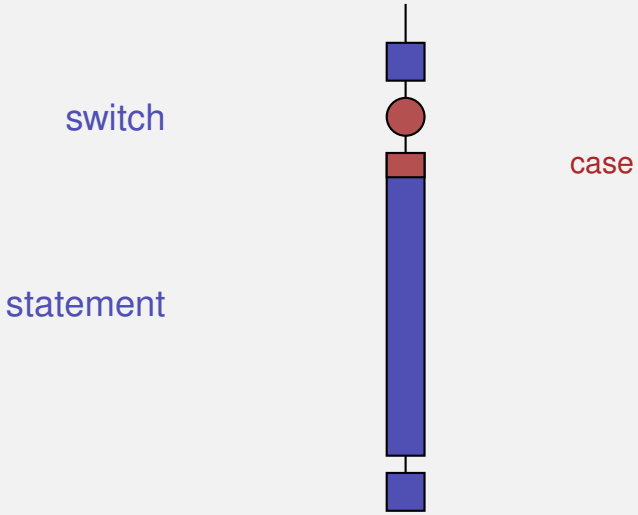
`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

- *condition*: Ausdruck, konvertierbar in einen integralem Typ
- *statement* : beliebige Anweisung, in welcher **`case`** und **`default`**-Marken erlaubt sind, **`break`** hat eine spezielle Bedeutung.

Kontrollfluss `switch`



Kontrollfluss `switch`

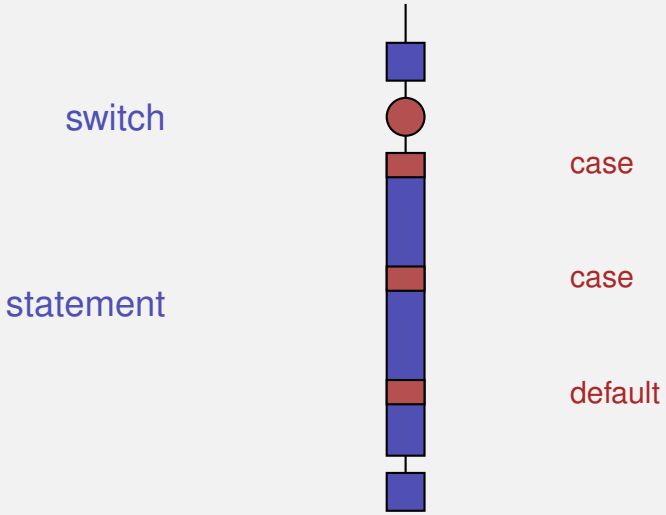
switch

statement

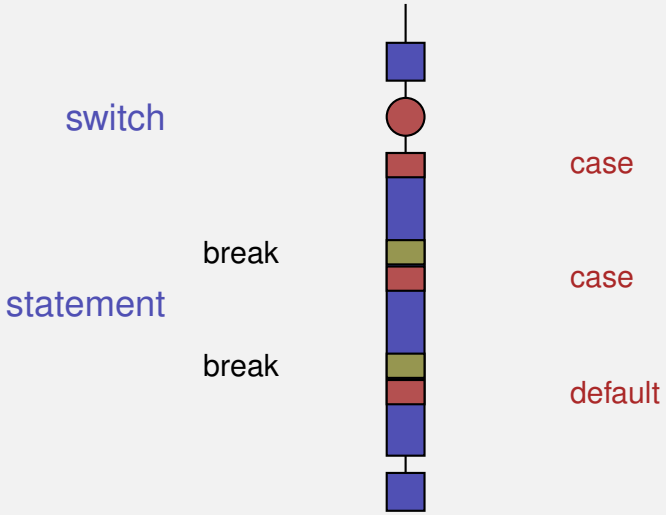
case

case

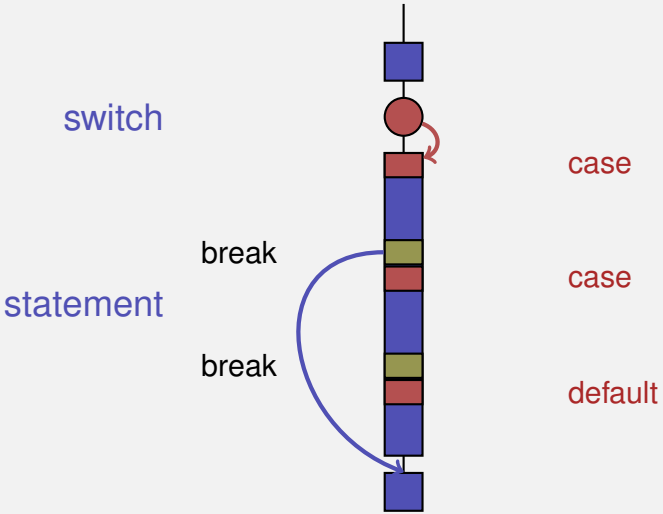
Kontrollfluss `switch`



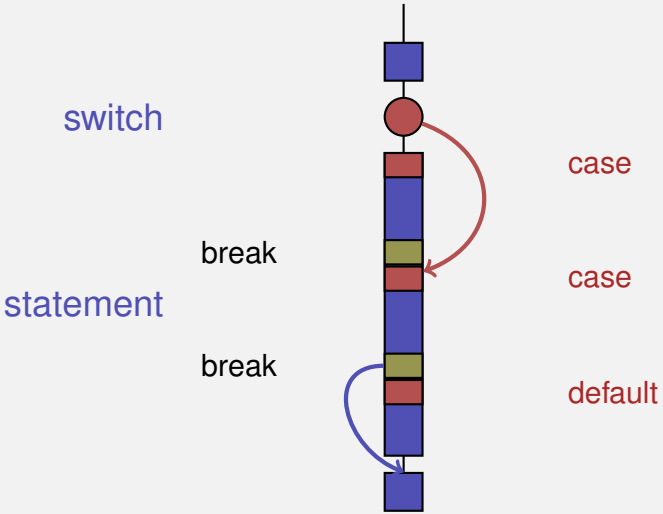
Kontrollfluss `switch`



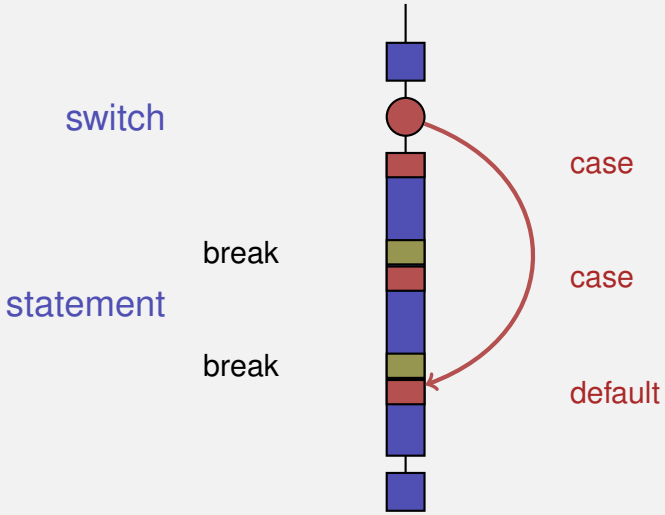
Kontrollfluss `switch`



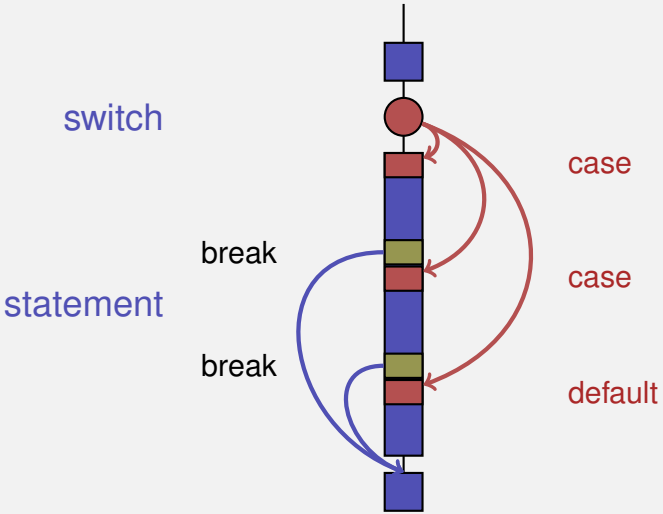
Kontrollfluss `switch`



Kontrollfluss `switch`



Kontrollfluss `switch`



Kontrollfluss `switch` allgemein

Fehlt **`break`**, geht es mit dem nächsten Fall weiter.

7: Keine Note!
6: bestanden!
5: bestanden!
4: bestanden!
3: oops!
2: ooops!
1: oooops!
0: Keine Note!

```
switch (Note) {
    case 6:
    case 5:
    case 4:
          Out.print("bestanden!");
          break;
    case 1:
          Out.print("o");
    case 2:
          Out.print("o");
    case 3:
          Out.print("oops!");
          break;
    default:
          Out.print("Keine Note!");
}
```