JAVA

Version 8.6
29.11.2024

NOTBUCH

Lizenz: CC BY-NC-ND 4.0

Prof. Dr. Michael Kipp
Technische Hochschule Augsburg
www.michaelkipp.de

Vorwort

Dieses Dokument soll ein Hilfsmittel sein, sowohl beim Erlernen von Java als auch beim alltäglichen Programmieren. Die Priorität liegt auf Klarheit und Kürze, daher ist einiges unvollständig dargestellt. Es sollten jedoch keine formalen Fehler enthalten sein. Wenn Sie Fehler finden, wäre ich für einen Hinweis dankbar: kipp@tha.de

Michael Kipp
Technische Hochschule Augsburg

INHALTE

Basics 1

Erste Hilfe

Style Guide

Datentypen

Arrays

Variablen

Basics 3

Verzweigung (If, Switch)

Schleifen

Boolesche Ausdrücke

Basics 2

Klassen

Methoden

Objekte

Advanced

Wichtige Klassen

Iteratoren

Aufzählungs-typen

Wichtige Hilfsklassen

Wrapperklassen

ERSTE HILFE

Informationen ausgeben

... mal eben kurz programminterne Infos ausgeben
... praktisch bei der Fehlersuche

```
System.out.println("Hallo");
```

Informationen ausgeben

... mal eben kurz programminterne Infos ausgeben
... praktisch bei der Fehlersuche

muss vom Typ **String** sein

```
System.out.println("Hallo");
```

Beispiel mit zusammengesetztem String:

```
int zahl = 3;
System.out.println("Alle guten Dinge sind " + zahl + ".");
```

STYLE GUIDE

Programmieren mit Stil

... unterscheidet den Profi vom Geek

... erhöht Lesbarkeit => weniger Fehler + wiederverwendbar

Konventionen

der Groß-/Kleinschreibung beachten

Reihenfolge

der Code-Teile einhalten

```
/**
 * Repräsentiert Erstklassiges
 * @author Gunter Grossmeier
 */
public class MyClass extends OtherClass {

  public static final int MEANING_OF_LIFE = 42;
  private int counter;
  private boolean hasStarted;

  /**
   * Initialisiert die Eigenschaft foo und baa.
   */
  public MyClass() {
    ...
  }

  /**
   * Trennt Spreu von Weizen.
   * @param amount Menge der Weizen
   */
  public void firstMethod (int amount) {
    ...
  }
}
```

Kommentare

vor Klassen, Konstruktoren und Methoden, im Javadoc-Format (d.h. mit /** beginnen)

Programmieren mit Stil

... unterscheidet den Profi vom Nerd

... erhöht Lesbarkeit => weniger Fehler + wiederverwendbar

Reihenfolge

der Code-Teile einhalten

```
/**
 * Repräsentiert Erstklassiges
 * @author Gunter Grossmeier
 */
public class MyClass extends OtherClass {

  public static final int MEANING_OF_LIFE = 42;
  private int counter;
  private boolean hasStarted;

  /**
   * Initialisiert die Eigenschaften foo und baa.
   */
  public MyClass() {
    ...
  }

  /**
   * Trennt Spreu von Weizen.
   * @param amount Menge der Weizen
   */
  public void firstMethod (int amount) {
    ...
  }
}
```

(1) Instanzvariablen (Konstanten vorneweg)

(2) Konstruktor(en)

(3) Methode(n)

Konventionen

der Groß-/Kleinschreibung beachten

Klassen groß schreiben

Konstanten nur Großbuchstaben

Variablen klein schreiben

Methoden klein schreiben

Kommentare

vor Klassen, Konstruktoren und Methoden, im Javadoc-Format (d.h. mit /** beginnen)

DATENTYPEN

Datentypen

primitive Typen

Datentypname	Beispiel
`byte`	`byte i = 3;`
`int`	`int i = 3;`
`long`	`long i = 3;`
`char`	`char c = 'A';`
`boolean`	`boolean done = false;`
`float`	`float speed = 3.5;`
`double`	`double speed = 3.5;`
`String`	`String name = "Tom";`
`Crab`	`Crab crab = new Crab();`

ganze Zahlen

Buchstaben

Komma-zahlen

Klassen sind auch Typen

ARRAYS

Arrays

... speichern mehrere Werte/Objekte, alle vom gleichen Typ
... haben eine feste Länge

```
int[] numbers;

numbers = new int[3];

numbers[0] = 5;
numbers[1] = -3;
numbers[2] = 23;


int[] numbers = { 5, -3, 23};

String[] friends = { "Peter", "Mary", "Paul"};

Person[] members = { new Person("Joe"),
                     new Person("Jane") };
```

Arrays

... speichern mehrere Werte/Objekte, alle vom gleichen Typ
... haben eine feste Länge

Typ
Genauso ein Typ
wie int, Crab, String...

```
int[] numbers;
```

Deklaration
Variable wird erschaffen

```
numbers = new int[3];
```

Erzeugung
(leeres) Array der Länge 3
wird angelegt

```
numbers[0] = 5;
numbers[1] = -3;
numbers[2] = 23;
```

Zuweisungen
Alle Werte werden gesetzt

```
int[] numbers = { 5, -3, 23};
```

Kurzform: Deklaration,
Erzeugung und Zuweisung in
einem

```
String[] friends = { "Peter", "Mary", "Paul"};

Person[] members = { new Person("Joe"),
                     new Person("Jane") };
```

Jeder Typ (auch Klassen) kann
als Array verwendet werden

VARIABLEN

Instanzvariablen

... enthalten die wesentlichen Eigenschaften der Klasse
... definieren später den Zustand eines Objekts

```
public class MyClass
{

    private int x;
    private int y;

}
```

Instanzvariablen

... enthalten die wesentlichen Eigenschaften der Klasse
... definieren später den Zustand eines Objekts

```
public class MyClass
{
```

Gültigkeit:
innerhalb der Klasse

```
private int x;
private int y;
```

Deklarationen

Sichtbarkeit
private = nur in Klasse
protected = auch in Unterklassen
public = überall

```
x = 5;
y = 3 * x + 15;
```

Zuweisungen

innerhalb einer beliebigen Methode oder Konstruktor dieser Klasse

Verwendung von x

```
}
```

Methodenparameter

... sind auch Variablen

... leben nur für die Dauer der Methode

```
public void manipulate( Crab crab, int size )
{
    ...

    crab.setSize(size);

    ...
}
```

Methodenparameter

... sind auch Variablen
... leben nur für die Dauer der Methode

Gültigkeit:
innerhalb der Methode

Typ

Parameterliste

```
public void manipulate( Crab crab, int size )
{
    ...

    crab.setSize(size);

    ...
}
```

Methodenparameter

Zugriff

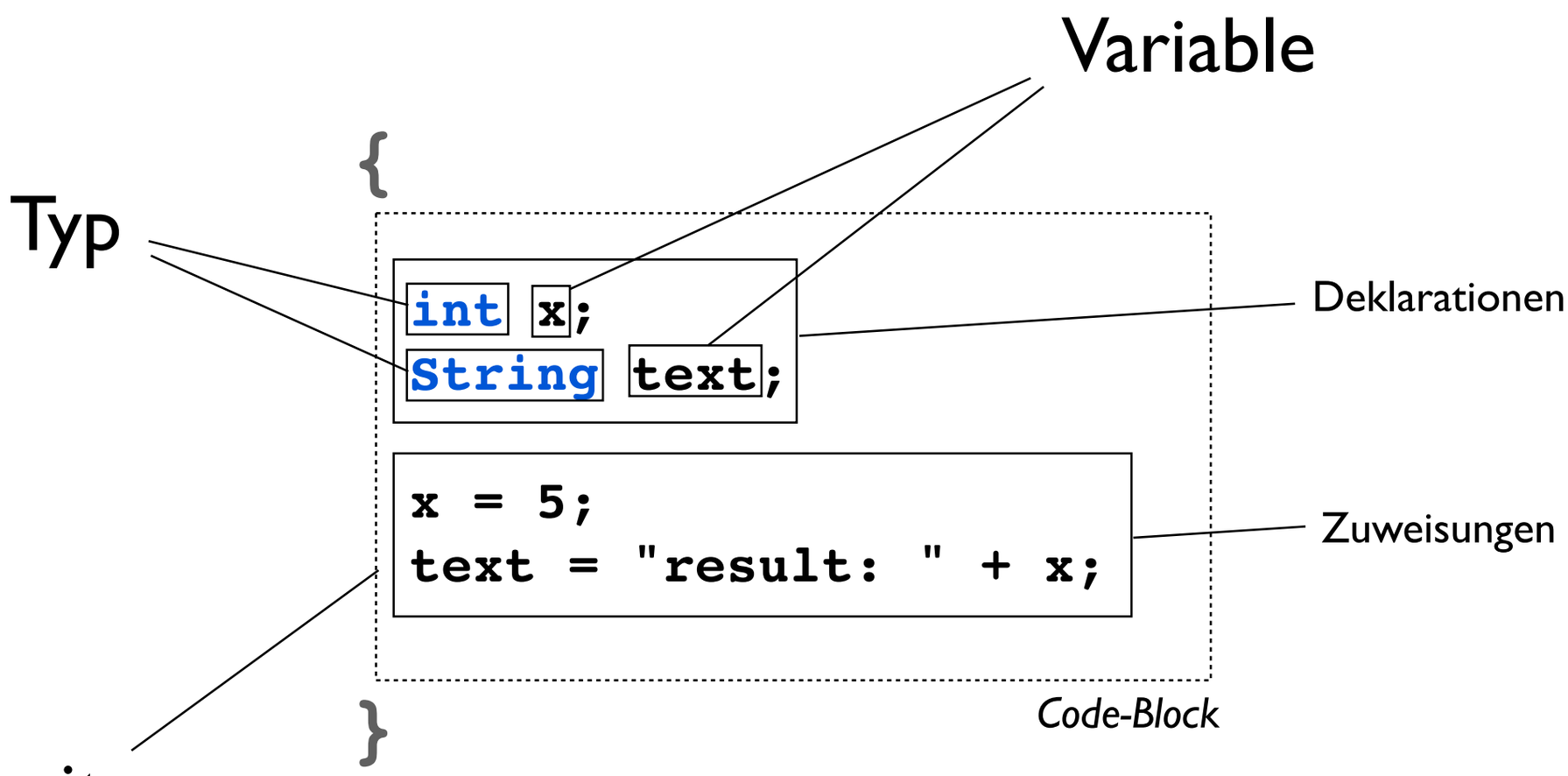
Lokale Variablen

... speichern Zwischenergebnisse (für kurze Zeit)
... sind ständige Begleiter beim Programmieren

```
{

  int x;
  String text;

  x = 5;
  text = "result: " + x;

}
```

Lokale Variablen

... speichern Zwischenergebnisse (für kurze Zeit)
... sind ständige Begleiter beim Programmieren



Gültigkeit:

innerhalb eines Code-Blocks (zwischen zwei zusammengehörigen geschweiften Klammern) in einer Methode oder Konstruktor

Konstanten

... speichern feste Infos, die mehrfach im Code vorkommen
... die Variante hier wird sehr häufig genutzt

```
public class MyClass
{
    public static final double GRAVITATION_ERDE = 9.81 ;

}
```

Konstanten

... speichern feste Infos, die mehrfach im Code vorkommen
... die Variante hier wird sehr häufig genutzt

```
public class MyClass
{
    public static final double GRAVITATION_ERDE = 9.81 ;
}
```

Name der Konstanten
(Konvention: alles groß schreiben)

Wertzuweisung
bei Definition

Sichtbarkeit

`public` heißt, dass andere Klassen mit MyClass.GRAVITATION_ERDE zugreifen können. Hier könnte auch private stehen, wenn die Konstante nur innerhalb der Klasse genutzt wird.

Klassenvariable

`static` heißt, dass GRAVITATION_ERDE nur ein einziges Mal für diese Klasse existiert und nicht (wie üblich) für jedes einzelne Objekt.

konstant

`final` heißt, dass GRAVITATION_ERDE nur ein Mal einen Wert zugewiesen bekommen darf. Entweder bei Definition (wie hier) oder im Konstruktor.

KLASSEN

Klasse (I)

... ist ein Bauplan für konkrete Objekte

```
public class MyClass extends OtherClass {

    ...

}
```

Klasse (I)

... ist ein Bauplan für konkrete Objekte

Sichtbarkeit

Oberklasse

```
public class MyClass extends OtherClass {

    ...

}
```

entfällt, wenn keine
Oberklasse vorhanden

Klasse (2)

... besteht aus im Innern aus drei Teilen

```
public class MyClass extends OtherClass {

  private int counter;
  private boolean hasStarted;

  public MyClass() {
    ...
  }

  public void firstMethod() {
    ...
  }

}
```

Klasse (2)

... besteht aus im Innern aus drei Teilen

```
public class MyClass extends OtherClass {
```

Instanzvariablen

```
private int counter;
private boolean hasStarted;
```

Konstruktor(en)

```
public MyClass() {
   ...
}
```

Wenn kein Konstruktor definiert wird, gibt es automatisch den leeren Konstruktor MyClass()

Methode(n)

```
public void firstMethod() {
   ...
}
```

```
}
```

Konstruktor

... wird bei Erzeugung eines Objekts zu Beginn aufgerufen

```
public class MyClass {

  private String name;
  private int time;

  public MyClass(String n, int time )
  {
      name = n;
      this.time = time;
  }

  ...
}
```

Konstruktor

... wird bei Erzeugung eines Objekts zu Beginn aufgerufen

Parameter

```
public class MyClass {

  private String name;
  private int time;

  public MyClass(String n, int time )
  {
      name = n;
      this.time = time;
  }

  ...
}
```

Sichtbarkeit

notwendig, weil die Instanzvariable `time` genauso heißt wie die Parametervariable `time` (nicht notwending bei `name` und `n`)

METHODEN

Methode

... führt eine Aktion (einen Algorithmus) aus
... bekommt Eingabeinformationen (Parameter)
... kann einen Wert zurückgeben

```
public class MyClass {

    public int computeSomething(int x, int y)
    {
        int result;
        result = x + y;
        return result;
    }

    ...
}
```

Methode

... führt eine Aktion (einen Algorithmus) aus
... bekommt Eingabeinformationen (Parameter)
... kann einen Wert zurückgeben

Rückgabetyp
(`void`, wenn nichts zurückgegeben wird)

Name
der Methode

Parameter
die der Methode als Input gegeben werden

Sichtbarkeit

```
public class MyClass {

  public int computeSomething(int x, int y)
  {
     int result;
     result = x + y;
     return result;
  }

  ...
}
```

mit diesen Variablen wird in der Methode gearbeitet; sie sind nur innerhalb der Methode gültig...

Liefert den Wert von "result" zurück und springt aus der Methode heraus (nachfolgender Code würde nicht ausgeführt werden).

Getter-Methode

... gibt den Wert einer Instanzvariable zurück

```
public class MyClass {

  private String name;

  public String getName()
  {
    return name;
  }

  ...
}
```

Getter-Methode

... gibt den Wert einer Instanzvariable zurück

Getter-Methode
für name

```
public class MyClass {

  private String name;

  public String getName()
  {
    return name;
  }

  ...
}
```

verwenden Sie den exakt gleichen Namen (bei der Methode erster Buchstabe groß)

Setter-Methode

... setzt den Wert einer Instanzvariable

```
public class MyClass {

  private String name;

  public void setName(String name)
  {
    this.name = name;
  }

  ...
}
```

Setter-Methode

... setzt den Wert einer Instanzvariable

Setter-Methode
für name

```
public class MyClass {

  private String name;

  public void setName(String name)
  {
    this.name = name;
  }

  ...
}
```

verwenden Sie den exakt gleichen Namen (bei der Methode erster Buchstabe groß)

notwendig, weil die Instanzvariable "name" genauso heißt wie die Parametervariable "name"

OBJEKTE

Objekte verwenden

```
Crab c;

c = new Crab();

c.move(5);

int x = c.getX();
```

Objekte verwenden

Deklaration
einer neuen lokalen Variable, die Objekte vom Typ Crab speichern kann

Konstruktor-Aufruf
liefert ein nagelneues Objekt der Klasse Crab zurück

```
Crab c;

c = new Crab();

c.move(5);

int x = c.getX();
```

Punktnotation
zum Aufruf von Methoden des Krabben-Objekts

Name des Objekts

VERZWEIGUNG

If-Anweisung

... führt Code-Block aus, wenn eine Bedingung erfüllt ist
... führt einen alternativen Block aus, wenn nicht

```
int x = 10;

if ( x > 0 && x <= 20 )
{
    System.out.println("im grünen Bereich");
} else {
    System.out.println("im roten Bereich");
}
```

If-Anweisung

... führt Code-Block aus, wenn eine Bedingung erfüllt ist
... führt einen alternativen Block aus, wenn nicht

```
int x = 10;

if ( x > 0 && x <= 20 )
{
    System.out.println("im grünen Bereich");
} else {
    System.out.println("im roten Bereich");
}
```

Bedingung
muss ein boolescher Ausdruck sein, der zu `true` oder `false` wird

If-Teil
wird ausgeführt bei `true`

Else-Teil
wird ausgeführt bei `false`

kann man auch weglassen

Switch-Anweisung

... ist das elegantere **If**, wenn es um eine Fallunterscheidung geht
... funktioniert nur mit int und enums (ab Java 7 auch mit Strings)

```
int level = 3;
String title = "";

switch ( level ) {
  case 1: title = "Anfänger";
    break;
  case 2: title = "Fortgeschrittener";
    break;
  case 3: title = "Experte";
    break;
  default: title = "Unbekannt";
    break;
}
```

Switch-Anweisung

... ist das elegantere **If**, wenn es um eine Fallunterscheidung geht
... funktioniert nur mit int und enums (ab Java 7 auch mit Strings)

```
int level = 3;
String title = "";

switch ( level ) {
  case 1: title = "Anfänger";
  break;
  case 2: title = "Fortgeschrittener";
  break;
  case 3: title = "Experte";
  break;
  default: title = "Unbekannt";
  break;
}
```

Variable

deren Wert die Grundlage der Fallunterscheidung bildet. Die Variable muss vom Typ int oder char sein (oder ein enum-Typ).

Fall

Anweisung(en) für einen bestimmten Fall. Der Fall wird durch break beendet (wenn nicht, wird der nachfolgende Fall mitausgeführt).

Wert

Wenn die Variable (hier: level) diesen Wert annimmt, wird dieser Fall ausgeführt.

Default-Fall

Wenn kein anderer Fall greift, wird dieser ausgeführt. Dieser Teil ist optional.

SCHLEIFEN

For-Schleife

... durchläuft den gleichen Code x-mal
... hat einen "eingebauten" Zähler (= Laufvariable)

```
for ( int i = 0; i < 7; i++ )
{
    System.out.println("Runde Nr. " + i);
}
```

Beispiel einer For-Schleife, in der die Laufvariable `i` folgende Zahlen durchläuft:
`10, 12, 14, 16`

```
for ( int i = 10; i <= 16; i=i+2 )
{
    System.out.println("Runde Nr. " + i);
}
```

For-Schleife

... durchläuft den gleichen Code x-mal
... hat einen "eingebauten" Zähler (= Laufvariable)

Schleifenkopf

Laufvariable

namens `i` wird deklariert und auf 0 gesetzt

Bedingung

muss erfüllt sein, damit die Schleife weiter läuft

Aktion

wird am Ende jeder Runde durchgeführt (hier: Hochzählen)

```
for ( int i = 0; i < 7; i++ )
{
    System.out.println("Runde Nr. " + i);
}
```

Schleifenkörper

Dieser Code wird x-mal durchlaufen (hier: 7 Mal)

Verwendung der Laufvariable
Hier läuft die Variable durch die Zahlen 0, 1, 2, 3, 4, 5, 6

Beispiel einer For-Schleife, in der die Laufvariable `i` folgende Zahlen durchläuft:
`10, 12, 14, 16`

```
for ( int i = 10; i <= 16; i=i+2 )
{
        System.out.println("Runde Nr. " + i);
}
```

While-Schleife

... wiederholt den immergleichen Code... unter **einer** Bedingung

```
Crab c = new Crab();

while ( ! c.atWorldEdge() )

{

    c.move();

}
```

Beispiel einer While-Schleife, die das gleiche tut wie die vorherige For-Schleife:

```
int x = 0;
while ( x < 7 )
{
      System.out.println("Runde Nr. " + x);
      x++;
}
```

While-Schleife

... wiederholt den immergleichen Code... unter **einer** Bedingung

Vorab deklarieren und füllen wir eine lokale Variable mit einem Crab-Objekt

Schleifenkopf

Bedingung

muss erfüllt sein, damit die Schleife weiter läuft

```
Crab c = new Crab();

while ( ! c.atWorldEdge() )
{
    c.move();
}
```

Schleifenkörper

Dieser Code wird solange wiederholt, bis die Bedingung im Kopf "false" wird

Beispiel einer While-Schleife, die das gleiche tut wie die vorherige For-Schleife:

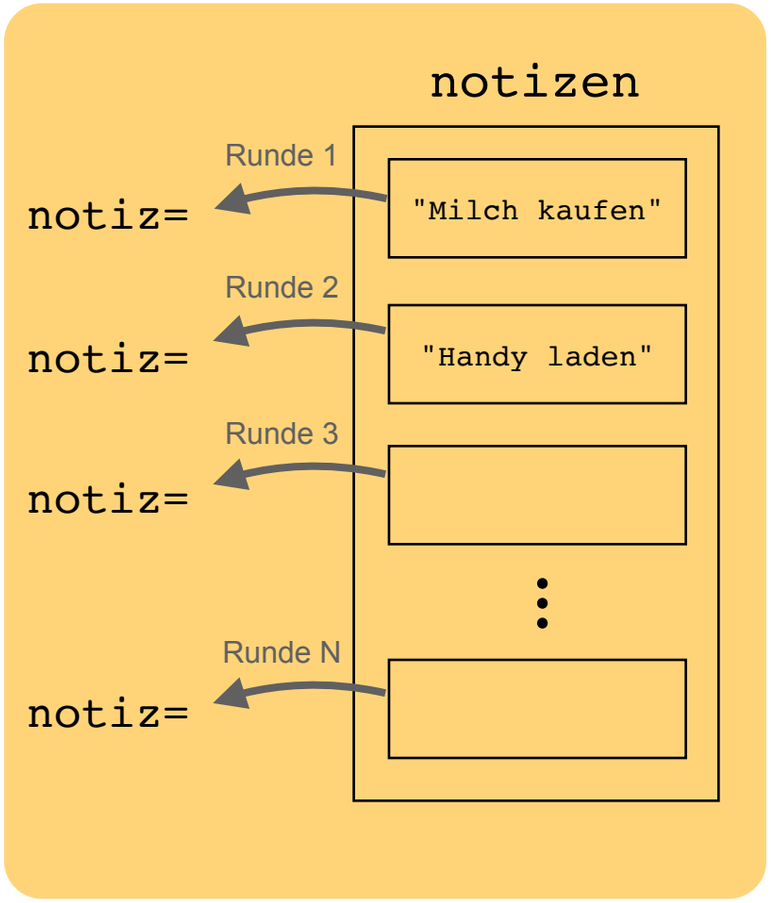
```
int x = 0;
while ( x < 7 )
{
        System.out.println("Runde Nr. " + x);
        x++;
}
```

Foreach-Schleife

... durchläuft alle Elemente einer Collection
(z.B. einer Liste)

```
ArrayList<String> notizen = new ArrayList<String>();
notizen.add("Milch kaufen");
notizen.add("Handy laden");
...
```

```
for ( String notiz: notizen )
{
    System.out.println( notiz );
}
```



Foreach-Schleife

... durchläuft alle Elemente einer Collection
(z.B. einer Liste)

Schleifenkopf

Vorab deklarieren und befüllen wir eine Liste als lokale Variable mit dem Namen `notizen`

```
ArrayList<String> notizen = new ArrayList<String>();
notizen.add("Milch kaufen");
notizen.add("Handy laden");
...
```

Collection

deren Elemente durchlaufen werden sollen

```
for ( String notiz: notizen )
{
    System.out.println( notiz );
}
```

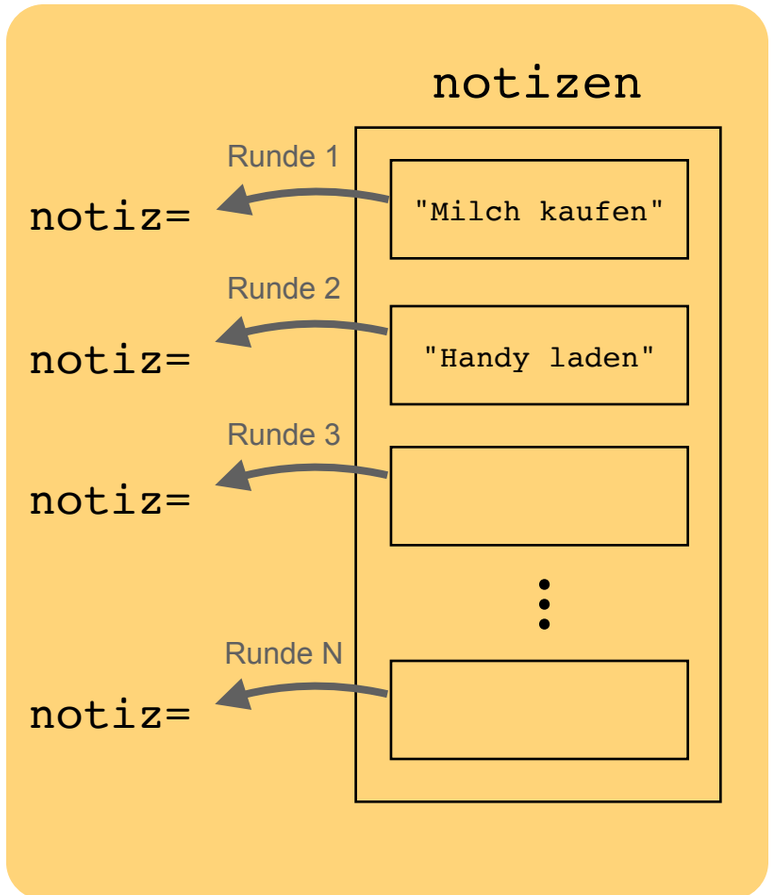
Laufvariable

namens `notiz` enthält immer das aktuelle Element der Liste

Schleifenkörper

Dieser Code wird so oft durchlaufen wie die Liste lang ist. Nur innerhalb dieses Code-Blocks ist die Laufvariable gültig.

Verwendung der Laufvariable



BOOLESCHE AUSDRÜCKE

Boolesche Ausdrücke (1)

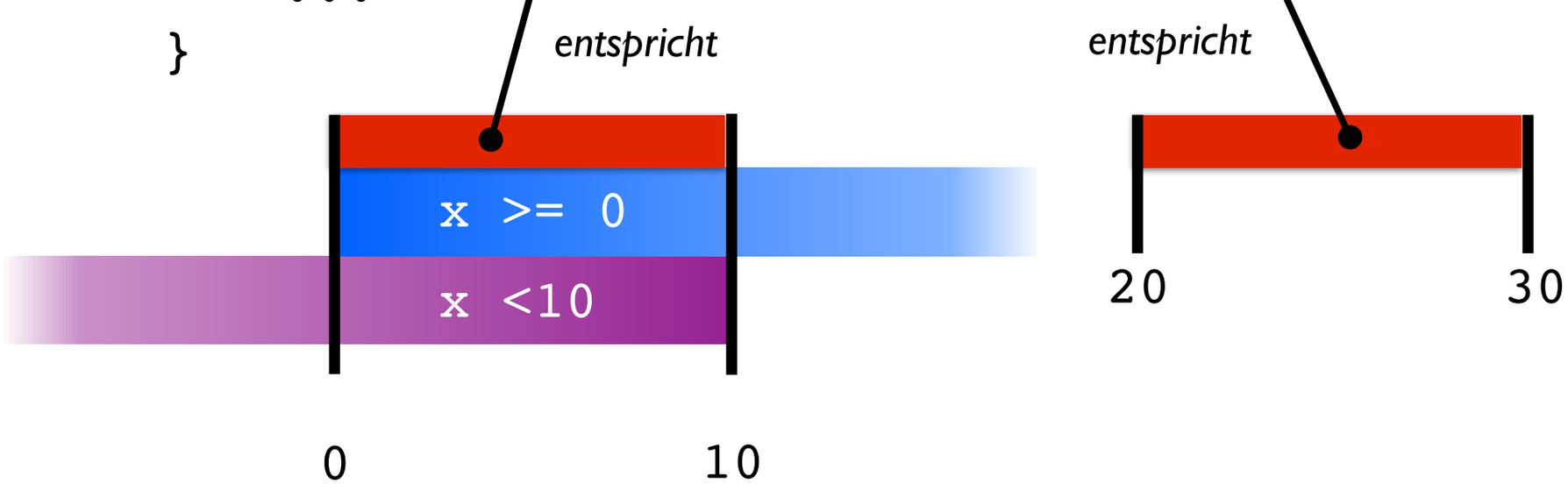
... sind `true` oder `false`

... brauchen Sie bei `if`, `while`, `switch`

logisches UND entspricht einer Schnittmenge (einzelner roter Bereich)

logisches ODER entspricht einer Vereinigungsmenge (beide roten Bereiche)

```
int x = 15;

if ((x >= 0 && x <10) || (x >= 20 && x <30))
{
    ...
}
```



Boolesche Ausdrücke (2)

... sind `true` oder `false`

... brauchen Sie bei `if`, `while`, `switch`

arithmetische Vergleiche

sind boolesche Ausdrücke

```
x  >  5
45 >= 20
y  <  x
y  <= 100
x  == z
```

Logische Operatoren

verknüpfen boolesche Ausdrücke mit UND, ODER und NICHT

Wahrheitstabellen zeigen alle möglichen Kombinationen

UND

x	y	x && y
T	T	T
T	F	F
F	T	F
F	F	F

ODER

x	y	x || y
T	T	T
T	F	T
F	T	T
F	F	F

NICHT

x	!x
T	F
F	T

WICHTIGE KLASSEN

String

... alles, was Text ist

im Java package:
java.lang

```
String street = "Friedberger Str.";
int num = 2;
String navi = "Adresse: " + street + num;
```

String

... alles, was Text ist

im Java package:
java.lang

Deklaration

einer lokalen String-Variable

```
String street = "Friedberger Str.";
int num = 2;
String navi = "Adresse: " + street + num;
```

Alles, was in Anführungszeichen ist, ist automatisch ein String-Objekt

Das Plus-Zeichen verknüpft Strings und vieles mehr... zu einem String

Wichtige Methoden

Details und weitere Methoden finden Sie in der Java-API ...

`int length()`	*Länge*
`boolean equals(String str)`	*gleicher Inhalt wie* `str`*?*
`String substring(int i, int j)`	`i`*-ter bis* `(j-1)`*ter Buchstabe*
`boolean contains(String str)`	*ist* `str` *enthalten?*
`boolean startsWith(String str)`	*deckt sich* `str` *mit Beginn?*
`boolean endsWith(String str)`	*deckt sich* `str` *mit Ende?*
`int indexOf(String str)`	*erstes Vorkommen von* `str`
`String concat(String str)`	*dieser String plus* `str`
`int compareTo(String str)`	*Reihenfolgevergleich*
`String replaceAll(String a, String b)`	*alles was* `a` *mit* `b` *ersetzen*

ArrayList<T>

im Java package:
java.util

... speichert viele Objekte vom Typ T

```
ArrayList<Person> friends = new ArrayList<Person>();
friends.add(new Person("John Smith"));
friends.add(new Person("Jody Foster"));
...
Person mySecondFriend = friends.get(1);
```

ArrayList<T>

im Java package:
java.util

... speichert viele Objekte vom Typ T

Typ
betrachten Sie den ganzen Ausdruck als Typ (genauso wie z.B. `Crab`)

Konstruktoraufruf erzeugt ein neues, leeres Listenobjekt

```
ArrayList<Person> friends = new ArrayList<Person>();
friends.add(new Person("John Smith"));
friends.add(new Person("Jody Foster"));
...
Person mySecondFriend = friends.get(1);
```

Liefert das zweite Element der Liste zurück (hier vom Typ `Person`)

Wichtige Methoden

Details und weitere Methoden finden Sie in der Java-API ...

`int`	`size()`	*Länge*
`boolean`	`add(T element)`	*Hängt* `element` *ans Ende*
`boolean`	`add(int i, T element)`	*Fügt* `element` *in Position* `i` *ein*
`T`	`get(int i)`	`i-tes` *Element*
`boolean`	`remove(T element)`	*Entfernt* `element`
`boolean`	`contains(T element)`	*Ist* `element` *in Liste?*
`boolean`	`addAll(ArrayList<T> list)`	*Hängt alle Element von* `list` *ans Ende*
`boolean`	`isEmpty()`	*Liste leer?*
`void`	`clear()`	*Liste leeren*

HashMap<K,V>

im Java package:
java.util

... speichert Informationen als Paare: Schlüssel (Typ K) und Wert (Typ V)
...Beispiel Telefonbuch: Schlüssel = Name, Wert = Telefonnummer

```
HashMap<String,String> telBuch = new HashMap<String,String>();
telBuch.put("Will Smith", "+1 141 994 385");

...
String number = telBuch.get("Will Smith");
```

Typ
betrachten Sie den ganzen Ausdruck als Typ (genauso wie z.B. `Crab`)

HashMap<K,V>

im Java package: **java.util**

... speichert Informationen als Paare: Schlüssel (Typ K) und Wert (Typ V)
...Beispiel Telefonbuch: Schlüssel = Name, Wert = Telefonnummer

```
HashMap<String,String> telBuch = new HashMap<String,String>();
telBuch.put("Will Smith", "+1 141 994 385");
```

Schlüssel

Wert

```
...
String number = telBuch.get("Will Smith");
```

Liefert den Wert zurück, der zu Schlüssel "Will Smith" gehört

Wichtige Methoden

Details und weitere Methoden finden Sie in der Java-API ...

`int`	`size()`	*Länge*
`V`	`put(K key, V value)`	*Fügt Paar* `key/value` *hinzu*
`V`	`get(K key)`	*Wert von Schlüssel* `key`
`V`	`remove(K key)`	*Entferne Paar mit* `key`
`Set<K>`	`keySet()`	*Alle Schlüssel*
`Collection<V>`	`values()`	*Alle Werte*
`boolean`	`containsKey(K key)`	*Schlüssel* `key` *vorhanden?*
`boolean`	`containsValue(V value)`	*Wert* `value` *vorhanden?*

WICHTIGE HILFSKLASSEN

Arrays

im Java package:
java.util

... ist selbst **kein** Array

... stellt praktische Helfer bereit (als statische Methoden)

```
String[] friends = { "Hanna", "Arndt", "Jerry" };
Arrays.sort(friends); // Arndt, Jerry, Hanna
```

Arrays

im Java package:
java.util

... ist selbst **kein** Array
... stellt praktische Helfer bereit (als statische Methoden)

```
String[] friends = { "Hanna", "Arndt", "Jerry" };
Arrays.sort(friends); // Arndt, Jerry, Hanna
```

Bei statischen Methoden muss die Klasse vorangestellt werden

Für alle Methoden gibt es Varianten für alle möglichen Arraytypen (`int[]` ist nur ein Beispiel)

Diese Methoden verändern das Array unwiederruflich!

Wichtige Methoden

Details und weitere Methoden finden Sie in der Java-API ...

`static void`	`sort(int[] a)`	*Sortiert Array*
`static void`	`sort(Object[] a)`	*Sortiert Array*
`static void`	`fill(int[] a, int val)`	*Befüllt ganzes Array mit* `val`
`static int`	`binarySearch(int[] a, int v)`	*Sucht* `v` *im Array (vorher sortieren!)*
`static boolean`	`equals(int[] a, int[] b)`	*Haben Arrays gleiche Werte?*
`static int[]`	`copyOf(int[] a)`	*Neues Array mit gleichen Werten*
`static String`	`toString(int[] a)`	*Array als String zum Ausgeben*

Collections

im Java package:
java.util

... ist selbst **keine** Collection
... stellt praktische Helfer bereit (als statische Methoden)

```
ArrayList<String> names = new ArrayList<String>();
names.add("Hanna"); // Liste befüllen
...
Collections.sort(names); // Liste sortieren
```

Collections

im Java package:
java.util

... ist selbst **keine** Collection
... stellt praktische Helfer bereit (als statische Methoden)

```
ArrayList<String> names = new ArrayList<String>();
names.add("Hanna"); // Liste befüllen
...
Collections.sort(names); // Liste sortieren
```

Bei statischen Methoden muss die Klasse vorangestellt werden

Strings haben eine "natürliche Ordnung" (alphabetisch), aber die meisten anderen Klassen nicht! Verwenden Sie sort() also mit Vorsicht...

Wichtige Methoden

Details und weitere Methoden finden Sie in der Java-API ...

`static void sort(List<T> list)`	*Sortiert Liste ("natürliche" Ordnung)*
`static void swap(List<T> list, int i, int j)`	*i-tes Element mit j-tem Element tauschen*
`static boolean disjoint(List<T> ls1, List<T> ls2)`	*Listen haben keine gemeinsamen Elemente*
`static void fill(List<T> list, T val)`	*Befüllt ganze Liste mit* `val`
`static boolean replaceAll(List<T>, T old, T newV)`	*Elemente ersetzen*
`static void reverse(List<T> list)`	*Reihenfolge umdrehen*
`static void shuffle(List<T> list)`	*Liste zufällig mischen*

ITERATOREN

Iterator

... Klasse zum Durchlaufen von Collections
... einzige Möglichkeit, um Elemente beim Durchlaufen zu löschen

```
ArrayList<Person> friends = new ArrayList<Person>();
friends.add(new Person("Johnny Depp"));
friends.add(new Person("Sigourney Weaver"));
...
```

```
Iterator<Person> it = friends.iterator();
while ( it.hasNext() )

{
    Person p = it.next();
    if (p.getName().startsWith("J")) {
        it.remove();
    }

}
```

Iterator

... Klasse zum Durchlaufen von Collections
... einzige Möglichkeit, um Elemente beim Durchlaufen zu löschen

```
ArrayList<Person> friends = new ArrayList<Person>();
friends.add(new Person("Johnny Depp"));
friends.add(new Person("Sigourney Weaver"));
...
```

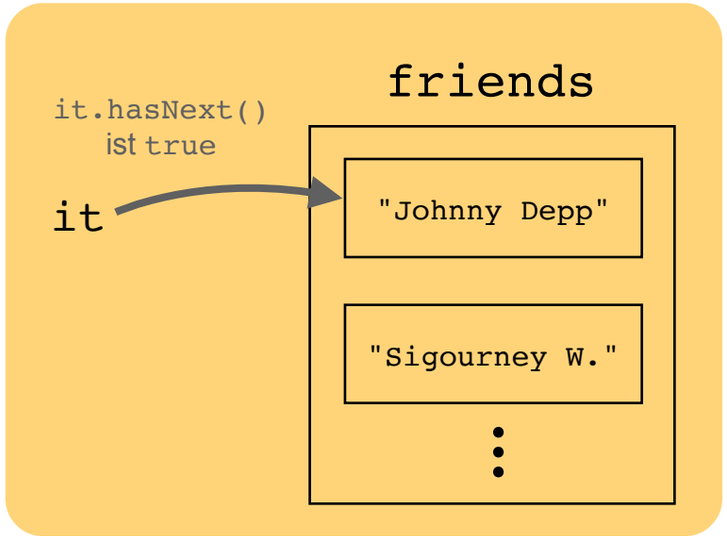
Collection
deren Elemente durchlaufen werden sollen

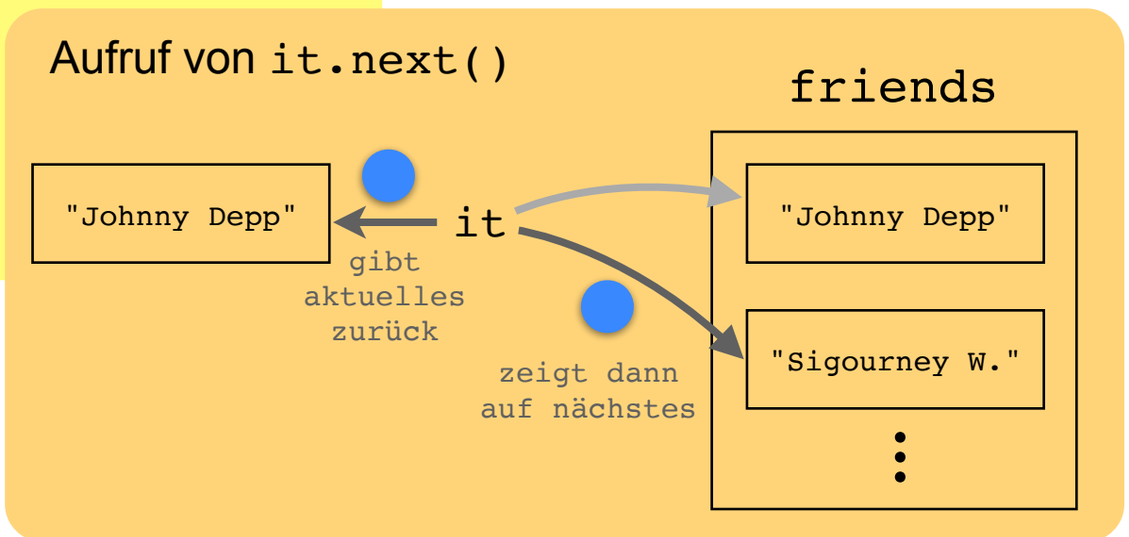
```
Iterator<Person> it = friends.iterator();
while ( it.hasNext() )

{
    Person p = it.next();
    if (p.getName().startsWith("J")) {
        it.remove();
    }

}
```

Gibt aktuelles Element zurück und schaltet auf nächstes Element

Iterator-Objekt
namens `it` ist vom Typ `Iterator<Person>`

Element, welches zuletzt zurückgegeben wurde, wird gelöscht





AUFZÄHLUNGS-TYPEN

Enum(erated Type)

... definiert eine Zahl von Optionen
... ist, wie eine Klasse, ein eigener Datentyp

```
public enum Gender {

    MALE, FEMALE, UNKNOWN;

}
```

Enum(erated Type)

... definiert eine Zahl von Optionen
... ist, wie eine Klasse, ein eigener Datentyp

Kopfzeile ähnlich wie bei "class"

Neuer Datentyp

```
public enum Gender {

    MALE, FEMALE, UNKNOWN;

}
```

Werte mit Großbuchstaben wie bei Konstanten.

Verwendungsbeispiel mit einer Klasse "Person":

```
Person p1 = new Person("Joe", Gender.MALE);
Gender gen = p1.getGender();
switch (gen) {
  case MALE: System.out.println("Lieber " + p1.getName());
    break;
  case FEMALE: System.out.println("Liebe " + p1.getName());
    break;
  default: System.out.println("Hallo " + p1.getName());
    break;
}
```

Enum(erated Type)

... definiert eine Zahl von Optionen
... ist, wie eine Klasse, ein eigener Datentyp

```
public enum Gender {

    MALE, FEMALE, UNKNOWN;

}
```

Jedes Enum bekommt zwei eingebaute Methoden mit:

```
public void printGenderValues() {
  Gender[] all = Gender.values();
  for (Gender val : all) {
    System.out.println(val);
  }
}
```

liefert alle definierten Werte (hier: MALE, FEMALE, UNKNOWN) als Array von Gender

```
String input = "FEMALE";
Gender g = Gender.valueOf(input);
```

wandelt einen String (z.B. "FEMALE") in das entsprechende Gender-Objekt um.

WRAPPER-KLASSEN

Wrapperklassen

... wickeln primitive Typen in "große" Klassen ein
... können in Collections (Listen, Hashtabellen) verwendet werden
... Autoboxing und Autounboxing garantieren sorgenfreien Umgang

```
ArrayList<int> zahlen = new ArrayList<int>(); // Fehler!

ArrayList<Integer> zahlen = new ArrayList<Integer>();
```

```
zahlen.add(new Integer(5));  // OK, aber umständlich
zahlen.add(-11);             // hier: automatisch eingepackt

int firstValueDoubled = zahlen.get(0) * 2; // ergibt 10
```

Wrapperklassen

... wickeln primitive Typen in "große" Klassen ein
... können in Collections (Listen, Hashtabellen) verwendet werden
... Autoboxing und Autounboxing garantieren sorgenfreien Umgang

Problem: primitive Typen hier nicht erlaubt

```
ArrayList<int> zahlen = new ArrayList<int>(); // Fehler!

ArrayList<Integer> zahlen = new ArrayList<Integer>();
```

stattdessen Wrapperklasse einsetzen!

```
zahlen.add(new Integer(5));   // OK, aber umständlich
zahlen.add(-11);              // hier: automatisch eingepackt

int firstValueDoubled = zahlen.get(0) * 2; // ergibt 10
```

wird automatisch in Integer eingepackt (Autoboxing)

wird automatisch von Integer zu int überführt (Autounboxing)

Wichtige Wrapperklassen:

primitiver Typ	Wrapperklasse
int	Integer
boolean	Boolean
float	Float
double	Double