Karlsruher Institut für Technologie

2. Tutorium
Datentypen, Operatoren, Referenzen, Strings

Tutorium 14

Péter Bohner | 09.11.2022

Inhaltsverzeichnis

Welche Befehlsformel innerhalb einer Klasse ermöglicht die Ausführung einer Java-Applikation?

Die *main*-Methode ist der Haupteinstiegspunkt in die Anwendung. Hier startet die Programmausführung.

```
class JavaApp {
    public static void main(String[] args) {
        System.out.println("Hello world.");
    }
}
```

Java Dateien werden **exakt** nach der Klasse benannt: Sonst Compilerfehler.

Wiederholung

Mit welchem Befehl kann ein Java-Programm kompiliert werden?

Kompilieren: `javac JavaApp.java`

Ausführen: `java JavaApp`

Wie kann eine Ausgabe auf der Konsole erzeugt werden?

`System.out.println();` oder `System.out.print();`

Sind Klassen Datentypen in Java? Ja...

```
class Car {
    Body body;
    Engine engine;
    ...
}
```

Mit welchem Schlüsselwort kann ein neues Objekt einer Klasse erzeugt werden?

```
new Classname();
```

Wiederholung primitive Datentypen

Typ	Erklärung	Wertebereich	Beispielwerte
boolean	Wahrheitswerte	true oder false	true, false
char	16-Bit-Unicode	0x0000 . . . 0xffff	'A', '\n', '\u05D0'
byte	8-Bit-Integer	$-2^7 \ldots 2^7 - 1$	12
short	16-Bit-Integer	$-2^{15} \ldots 2^{15} - 1$	12
int	32-Bit-Integer	$-2^{31} \ldots 2^{31} - 1$	12
long	64-Bit-Integer	$-2^{63} \ldots 2^{63} - 1$	12L, 14L
float	32-Bit-Gleitk.	1,40239846E-45f. . . 3,40282347E+38f	9.81F, 0.3E-8F, 2f
double	64-Bit-Gleitk.	4,94065645841246544E-324. . . 1,79769131486231570E+308	9.81, 3e1

Wiederholung Variablen

„Platzhalter“ für Werte eines Datentyps

Deklaration

- Name und Datentyp der Variable
- `Datentyp Name;`

Zuweisung

- Wert der Variable
- `Name = Wert;`

Initialisierung

- Kombination aus Deklaration und Zuweisung
- `Datentyp Name = Wert;`

Attribute

Wie greift man auf Attribute von Objekten zu?

- Zugriff mit `Objektname.Variablenname`
- Umgang wie mit „normalen“ Variablen
- Also Initialisierung mit:

  $\Rightarrow$ `Objektname.Variablenname = Wert;`

Arithmetische Operationen

		Präzedenz
unäres −/+		1
+	Addition	3
−	Subtraktion	3
*	Multiplikation	2
/	Division	2
%	Modulo (Rest bei Ganzzahldivision)	2

Präzendenz

Der Wert gibt an wie stark der Operator bindet.
Je **kleiner** der Wert, desto **stärker** bindet der Operator.
Beispiel aus der Mathematik: $5 \cdot 3 + 2 = 17 \neq 25$ da $\cdot$ stärker als + bindet

In-/Dekrement-Operator

++	erhöht Variable um eins
−−	verringert Variable um eins

Notationen:

Präfix	y = ++x	x = x + 1;	y = x;
Postfix	y = x++	y = x;	x = x + 1;

Analog für −−

Vergleichs Operationen

Liefern Wahrheitswert (boolean):

		Präzedenz
<	kleiner	5
<=	kleiner-gleich (entspricht $\leq$)	5
>	größer	5
>=	größer-gleich (entspricht $\geq$)	5
==	Gleichheit	6
!=	Ungleichheit	6

Operationen auf boolean-Werten

		Präzedenz
!	Negation	1
&&	logisches Und	10
||	logisches Oder	11

Ungenauigkeit double

```
double a = 2.0;
double b = 1.9;
double c = a - b;
```

Wert von c: 0.10000000000000009
Zahlen sind nur **endlich** genau ⇒ *Rundungsfehler*

Achtung bei Vergleichen

Statt Vergleich mit `actual == expected`, geeignetes Delta verwenden:
`Math.abs(expected - actual) < delta`

Hier: Statt `c == 0.1`, verwende `Math.abs(c - 0.1) < 0.0001`

Overflow

Was ist die Ausgabe des folgenden Codes?

```
byte a = 127;
Sytem.out.println(a++ + ", " + a);
```

```
127, -128
```

Warum ist das so?

Der Wertebereich von `byte` geht von -2^7 `(-128)` bis $2^7 - 1$ `(127)`.
Inkrementieren vom größten Wert
$\Rightarrow$ kleinster Wert
Dekrementieren vom kleinsten Wert
$\Rightarrow$ größter Wert

Wie ist bei Gleitkommazahlen?

Karlsruher Institut für Technologie

Overflow:

```
Sytem.out.println(Double.MAX_VALUE);
System.out.println(Double.MAX_VALUE + 1.0);
System.out.println(Double.MAX_VALUE * 2);
```

```
1.7976931348623157E308
1.7976931348623157E308
INFINITY
```

Underflow:

```
Sytem.out.println(Math.pow(2, -1074));
System.out.println(Math.pow(2, -1075));
```

```
4.9E-324
0.0
```

Benennungs-Koventionen

Variablen, Attribute und Methoden	lowerCamelCase	userCounter
Klassen, Enums und Interfaces	UpperCamelCase	StringBuilder
Klassenkonstanten, Einträge in Enums	GROSS_GESCHRIEBEN	GRAVITATION_EARTH
Pakete	kleinbuchstaben (umgedrehte Domain)	com.java.util

Speicherung

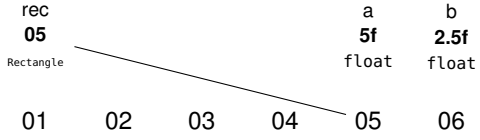
Abstrakter Speicheraufbau:

x	y	z	a	b	c
0	**2**	**-2**	**1.6f**	**-2.5f**	**50**
`int`	`int`	`int`	`float`	`float`	`int`
01	02	03	04	05	06

- Variablen werden an aufeinanderfolgenden Adressen im Speicher gespeichert
- Pro Adresse werden immer nur 8 Bits gespeichert
- `int` braucht eigentlich 4 Adressen (hier vereinfacht)

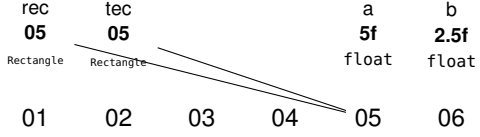
Speicherung von Objekten

- Für ein Objekt wird immer nur eine Referenz auf die Speicheradressen der Objektattribute gespeichert
- Die Objektvariable zeigt auf die Objektidentität
- Beispiel: `Rectangle rec` mit `rec.a = 5f` und `rec.b = 2.5f`



Zuweisung von Objekten

An der Situation von eben (`Rectangle rec` mit `rec.a = 5f` und `rec.b = 2.5f`) wird folgendes geändert:
`Rectangle tec = rec;`
Wie sieht der Speicher anschließend aus?



Spezialfall

null

- Wichtiges Element!
- `Rectangle rec = null;`
- Referenz auf nichts
- Es wird auf „kein Objekt“ referenziert
- Zugriff auf das Objekt führt zur Exception

```
rec.a = 10;
```

⇒ `java.lang.NullPointerException`

Aufgabe - Teil A

Modellierung

Modelliert eine Klasse **Date**, die alle für ein Datum wichtigen Attribute enthält (Tag, Monat, Jahr). Wählt geeignete Datentypen.

Tipp

Benutzt für das Attribut Monat ein enum.

Lösung - Teil A

```
class Date {
    enum Month {
        JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
        AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
    }
    int day;
    Month month;
    int year;
}
```

Aufgabe - Teil B

Werte setzen

Schreibt ein Programm, welches ein Datum in den **Argumenten** übergeben bekommt:

```
java ProgramName day month year
```

Erstellt ein **neues Objekt** vom Typ `Date` und weist diesem die übergebenen Werte zu.

Alle Argumente werden als gültige und sinnvolle Integer-Werte übergeben. (keine Fehlerbehandlung nötig)

Wichtige Info

Um aus einem String ein Integer zu „parsen“:

```java
int a = Integer.parseInt("5");
```

Lösung - Teil B

```
class DateApp {
    public static void main(String[] args) {
        int day = Integer.parseInt(args[0]);
        Month month = null;
        int year = Integer.parseInt(args[2]);

        switch(Integer.parseInt(args[1])) {
            case 1:
                month = Month.JANUARY;
                break;
            ...
            case 12:
                month = Month.DECEMBER;
                break;
        }
        Date date = new Date();
        date.day = day;
        date.month = month;
        date.year = year;
    }
}
```

Scanner

Für Eingaben während das Programm läuft, verwende die Klasse `Scanner`:

Importieren mit `import java.util.Scanner;` vor der Klassendefinition!
Neuen Scanner erstellen mit: `new Scanner(System.in)`

Einlesen mit:

`nextLine()`	für Strings
`nextInt()`	für Ganzzahlen
`nextDouble()`	für Gleitkommazahlen
...	...

Scanner - Beispiel

```java
import java.util.Scanner;
class ReadTerminal {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Gib eine Zahl ein!");
        int input = scanner.nextInt();
        System.out.println("Deine Zahl war " + input + ".");
    }
}
```

```
$ java ReadTerminal
Gib eine Zahl ein!
> 5
Deine Zahl war 5.
```

Fallunterscheidung

Beispiel aus dem echten Leben: Ist die Ampel rot?

Ja

Dann bleibe ich stehen.

Nein

Weiterfahren!

Nach diesem Schema funktioniert Fallunterscheidung in Java
⇒ Ist etwas wahr? Dann mache dies, sonst mache das oder was ganz anderes.

if-else

Allgemein

```
if (Bedingung is true) {
    // führe das aus
}
else {
    // führe etwas anderes aus
}
```

Beispiel

```
if (light.isRed()) {
    stop();
}
else {
    drive();
}
```

Die Bedingung muss immer ein `boolean` sein!

else if

Was tun wir, wenn wir mehrere alternative Bedingungen überprüfen müssen?

Allgemein

```
if (Bedingung is true) {
    // führe das aus
}
else if (Bedingung2 is true) {
    // führe das aus
}
else {
    // führe etwas anderes aus
}
```

Beispiel

```
if (light.isRed()) {
    stop();
}
else if (light.isGelb()) {
    decide();
}
else {
    drive();
}
```

Kurzschlussauswertung

Und (Ausdruck1 && Ausdruck2)

`false && Ausdruck2`	Ausdruck2 wird nicht mehr ausgewertet
`true && Ausdruck2`	Nun muss auch Ausdruck2 ausgewertet werden
`Ausdruck1 & Ausdruck2`	Es werden immer beide Ausdrücke geprüft

Anwendung

Zugriff auf Objektreferenz die null sein kann:

```
if (rec != null && rec.a > 5)
```

Zugriff auf `rec.a` erfolgt nur falls die Referenz rec auf ein Objekt verweist

Kurzschlussauswertung

Oder (Ausdruck1 || Audruck2)

`true || Ausdruck2`	Ausdruck2 wird nicht mehr ausgewertet
`false || Ausdruck2`	Nun muss auch Ausdruck2 ausgewertet werden
`Ausdruck1 | Ausdruck2`	Es werden immer beide Ausdrücke geprüft

Und jetzt ihr

Aufgabe

Vervollständigt die folgende Methode, sodass das Maximum beider Zahlen zurückgegeben wird.

```
public static void main(String[] args) {
    int a = 5;
    int b = 2;
    // your code: find the maximum of a and b
    // use 'System.out.println("");' for your output
}
```

Für Schnelle

Erweitert die Methode, sodass entweder der Wert -1 (`a < b`), 0 (`a == b`)
oder 1 (`a > b`) ausgeben wird.

Lösung

Aufgabe

```java
public static void main(String[] args) {
    int a = 5;
    int b = 2;
    if (a > b) {
        System.out.println(a);
    }
    else {
        System.out.println(b);
    }
}
```

Für Schnelle

```java
public static void main(String[] args) {
    int a = 5;
    int b = 2;
    if (a > b) {
        System.out.println("1");
    }
    else if (a == b) {
        System.out.println("0");
    }
    else {
        System.out.println("-1");
    }
}
```

switch-case

```
switch (Ausdruck) {
    case 1: // 1. Anweisung;
            break;
    case 2: // 2. Anweisung;
            break;
    ...
    case n: // n. Anweisung;
            break;
    default: // x. Anweisung;
            break;
}
```

Wichtig

- Wann benutzt man switch-case?
- Anwendung, wenn es sehr viele Fallunterscheidungen gibt
- Mit `Ausdruck` ist ein Datentyp gemeint, der verschiedene Werte annehmen kann (char, byte, short, int, enum, String)
- **break** nicht vergessen und überlegen, ob **default** notwendig ist!

Beispiel

```
switch (lightcolor) {
    case "red":
        System.out.println("Stop!");
        break;
    case "yellow":
        System.out.println("Decide!");
        break;
    case "green":
        System.out.println("Drive!");
        break;
    default:
        System.out.println("I think your traffic light is broken!");
        break;
}
```

Beispiel - Wie viele Tage hat ein Monat?

```
switch (month) {
    case 2:
        System.out.println("28");
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println("30");
        break;
    default:
        System.out.println("31");
        break;
}
```

Zusatz

- „Fallen“ durch die Fälle möglich
- Kein guter Stil, erschwert Fehlerfindung!

Kleines Extra

Der ternäre Operator

- Abkürzung für ein einfaches `if-else`
- `bedingung ? wert1 : wert2`

```
int k;
if (i == 10) {
        k = 12;
} else {
        k = 5;
}
```

```
int k = (i == 10) ? 12 : 5;
```

⇒ Nur sparsam und bei nicht zu komplexen Bedingungen verwenden!

Schleifen - Einführung

Schleifen erlauben die wiederholte und bedingte Ausführung von Anweisungen

Schleifenarten:

- `while`
  - `do-while`
- `for`
  - `for-each` (später bei Arrays)

Die while-Schleife

```
while (bedingung) {
    // Anweisungsblock
}
```

Anweisungsblock wird ausgeführt, solange die Bedingung wahr ist:

1. Überprüfe Bedingung
2. **Bedingung wahr:** führe Anweisungsblock aus
   - **Wiederholung:** springe zu Schritt 1
3. **Sonst:** breche Ausführung der Schleife ab und führe Code nach der Schleife aus

Achtung: Endlos-Schleifen

Endlos Schleifen leicht möglich bei Bedingungen, die immer wahr sind: `while (true) { ... }`

Beispiel while-Schleife

Füllstand

```
Scanner scanner = new Scanner(System.in);
int capacity = 100;
int fillLevel = 0;
while (fillLevel <= capacity) {
    fillLevel += scanner.nextInt();
}
System.out.println("Behälter ist übergelaufen!");
```

Die do-while-Schleife

```
do {
    // Anweisungsblock
} while (bedingung);
```

Anweisungsblock wird ausgeführt, solange die Bedingung wahr ist, aber **mindestens** einmal:

1. führe Anweisungsblock aus
2. Überprüfe Bedingung
3. **Bedingung wahr:** springe zu Schritt 1 (Wiederholung)
4. **Sonst:** breche Ausführung der Schleife ab und führe Code nach der Schleife aus

Achtung: Endlos-Schleifen

Endlos Schleifen leicht möglich bei Bedingungen, die immer wahr sind: `do { ... } while (true);`

Beispiel do-while-Schleife

Errate meine Zahl

```
Scanner scanner = new Scanner(System.in);
int number = 86;
int guess;
do {
    guess = scanner.nextInt();
} while (guess != number);
System.out.println("Glückwunsch, du hast meine Zahl erraten!");
```

Die for-Schleife

Die for-Schleife ist eine sogenannte Zählschleife.

```
for (initialisierung; bedingung; schritt) {
    // Anweisungsblock
}
```

Anweisungsblock wird ausgeführt, solange die Bedingung wahr ist:

0. führe Initialisierungs-Anweisung **einmalig** aus
1. Überprüfe Bedingung
2. **Bedingung wahr:** führe Anweisungsblock aus
   2.1 führe die Schritt-Anweisung aus
   2.2 **Wiederholung:** springe zu Schritt 1
3. **Sonst:** breche Ausführung der Schleife ab und führe Code nach der Schleife aus

Beispiele for-Schleife

Countdown

```java
for (int i = 10; i > 0; i--) {
    System.out.println(i);
}
```

Fakultätsberechnung

```java
int result = 1;
for (int i = 1; i < 5; i++) {
    result *= i;
}
```

while vs. for

for-Schleife

- Anzahl der Iteration ist bekannt
- Kurzform für n-mal die Schleifenanweisung (Schleifenrumpf) hintereinander schreiben
- eigentlich sequentielles Programm ohne Schleife

while-Schleife

- die Anzahl der Wiederholung hängt von den Schleifenanweisungen (Schleifenrumpf) ab
- man weiß nicht unbedingt, wie oft diese ausgeführt wird
- Es gibt keinen Algorithmus, der entscheidet, ob ein Programm mit einer while Schleife terminiert

Die break-Anweisung

- Manchmal ist es nötig eine Schleife vorzeitig zu verlassen
- `break;` veranlasst das sofortige Verlassen der innersten Schleife
- nur sparsam und gezielt einsetzen, so dass der Programmcode übersichtlich und verständlich bleibt

Beispiel Wurzel ziehen

```
Scanner scanner = new Scanner("4 9 -4 16");
while (scanner.hasNextInt()) {
    int input = scanner.nextInt();
    if (input < 0)
        break;
    System.out.println(Math.sqrt(input));
}
```

Die continue-Anweisung

- `continue` bricht die aktuelle Scheifeniteration ab und springt direkt zur nächsten Iteration
  - Die Schleifenbedingung wird dabei geprüft
  - Bei for-Schleifen wird trotzdem die Schritt-Anweisung ausgeführt

Beispiel

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.print(i + " ");
}
```

Ausgabe: `1 2 3 4 6 7 8 9 10`

Übung

kleines Einmaleins

Schreibt ein Programm, welches das kleine Einmaleins berechnet.

Ausgabe:

1er Reihe:
1 2 3 4 5 6 7 8 9 10
2er Reihe:
2 4 6 8 10 12 14 16 18 20
...
10er Reihe:
10 20 30 40 50 60 70 80 90 100

Tipps

1. Überlegt euch zuerst, welche Schleife am geeignesten ist. (for-Schleife)
2. Ihr braucht zwei ineinander verschachtelte Schleifen.

Lösung

```java
class Main {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i + "er Reihe:");
            for (int j = 1; j <= 10; j++) {
                System.out.print(i * j + " ");
            }
            System.out.println();
        }
    }
}
```

Bis zum nächsten Tutorium am 16.11.2022!