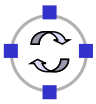
Programmierung und Algorithmen

Kapitel 3
Java Einführung



Überblick

Java – ein Überblick

Historischer Überblick

Arbeiten mit Java

Datentypen in Java

Umsetzung von Algorithmen in Java

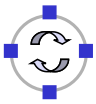


Java - was ist das ?

- Objektorientierte Programmiersprache
  - Entwickelt von Sun Microsystems
  - “Internet-Programmiersprache”

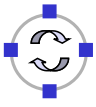


- Plattform
  - Umgebung zur Ausführung von Java-Programmen für PC, Workstation, Mobiltelefonen, Handhelds, Set-Top-Boxen, . . .
  - Bibliothek von nützlichen Klassen/Funktionen, z. B. Datenbankzugriff, 2D/3D-Grafik, verteilte Verarbeitung, . . .



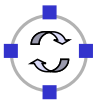
Warum gerade Java?

- Java ist leicht zu lernen (leichter als C, C++, . . .).
- Java ist syntaktisch ähnlich zu C, C++, C#, . . .
- Java läuft überall (fast . . .)
  - auf PC (Windows, Linux, Mac)
  - auf mobilen Geräten: Mobiltelefonen (Android)
  - eingebettet in andere Anwendungen (Datenbanksysteme, Applikationsserver, . . .)
- Es gibt viel Literatur zu Java.
- Es gibt zahlreiche Werkzeuge und Beispiele.



Java – Algorithmen & Programmierung

- Vorlesung
  - Umsetzung von Algorithmen
  - Implementierung von Datenstrukturen
  - Einführung in die Programmierung
- Übungen
  - Lösung der praktischen Aufgaben



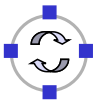
Historischer Überblick: Programmiersprachen

- Anfang der 50er Jahre: erste höhere Programmiersprachen (Autocode)
- 1954: Fortran (FORmula TRANslator) – für wiss.-techn. Anwendungen
- 60er Jahre:
  - Algol (ALGOrithmic Language)
  - Lisp (LISt Processing): funktionale Programmiersprache, künstliche Intelligenz
  - COBOL (COmmon Business Oriented Language): für kommerzielle Anwendungen, auch heute noch sehr weit verbreitete Programmiersprache
  - BASIC (Beginner's All-purpose Symbolic Instruction Code)
- 1967: Simula-67 – objektorientierte Sprache für Simulation



Historischer Überblick: Programmiersprachen

- Anfang der 70er Jahre:
  - Pascal (N. Wirth) – Methode der strukturierten Programmierung, P-Code
  - CPL, BCPL, C – erstes Betriebssystem (UNIX) in einer Hochsprache (C)
- Mitte der 70er Jahre:
  - Modula, Ada – modulare Programmierung
- Ende der 70er Jahre: Smalltalk – reine Objektorientierung
- 1983: C++ (AT&T) – Objektorientierung und C
- Danach: Eiffel (design by contract), Oberon, Skriptsprachen (Tcl, Perl, . . .)
- Heute: Java, C#, Python, . . .



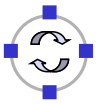
Historischer Überblick: Java

- 1990: Oak – als Programmiersprache für Consumer Electronics (Sun Microsystems; 2010 von Oracle übernommen)
- 1993: Entwickung des World Wide Web
  - Umorientierung auf Web-Anwendungen
  - Umbenennung in Java
- 1995: Freigabe des HotJava-Browsers
  - Aktive Web-Inhalte (Applets), erste größere Java-Anwendung
- 1995: Netscape Navigator 2.0 mit Applet-Unterstützung
- 1997: Freigabe des JDK 1.1
  - Unterstützung durch alle großen Firmen: IBM, Microsoft,…
- Seitdem: Entwicklungen für Desktop, Server, Mobile/Wireless, . . .
- Heute: Java Standard Edition (SE) Version 18 (seit März 2022)



Applet vs. Applikation

- **Applet** (von Oracle seit Java 11 entfernt)
  - Java-Programm, das in andere Applikation eingebettet ist
  - Bsp.: Applets in Web-Dokumenten; werden vom Web-Server in den Browser geladen
  - Sicherheitsrestriktionen: kein Zugriff auf lokalen Computer
  - Anwendung: Frontend zu Internet-Diensten, Präsentation, aktive Dokumente (3D-Welten), . . .
- **Applikation**
  - Java-Programme, die unabhängig von anderen Anwendungen ausgeführt werden können *(standalone)*
  - Keine Sicherheitsrestriktionen
  - Anwendung: Entwicklungswerkzeuge, Office-Anwendungen, Browser, . . .



Anwendungen: Spiele

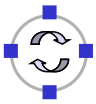


(100% Java) (~99% C++)



Eigenschaften von Java

- Einfach (relativ betrachtet…:o)
  - Automatisierte Speicherverwaltung
  - Verzicht auf Zeiger und `goto`
- Objektorientiert
  - Klassen als Abstraktionskonzept
- Robust und sicher
  - Starke Typisierung
  - Laufzeitüberprüfung von Zugriffen
- Interpretiert und dynamisch
  - Virtuelle Java-Maschine
  - Einfache, schnelle Programmentwicklung
  - „Kleine“ Programme
- Architekturneutral und portabel
  - Plattformunabhängiger Zwischencode (Bytecode)
  - Programme sind ohne Änderungen ablauffähig unter Windows, Unix, MacOS, . . .



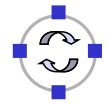
Plattformunabhängigkeit von Java

- Mögliche Klassifikation von Programmiersprachen
  1. **Maschinennahe Programmierung:** direkte Codierung von Prozessorinstruktionen, z. B. Assembler
     → schnell aber plattformspezifisch und umständlich
  2. **Compilierte Programmiersprachen:** Übersetzung des Quelltextes in Maschinencode, z. B. C, C++, Pascal, etc.
     → relativ schnell aber Quelltext abhängig von Compiler und resultierendes Programm plattformspezifisch
  3. **Interpretierte Programmiersprachen:** Übersetzung des Programmtextes zur Laufzeit, z. B. PHP, Python, etc.
     → plattformunabhängig und flexibel, aber relativ langsam und anfällig für Laufzeitfehler
- Java geht Sonderweg zwischen 2 und 3

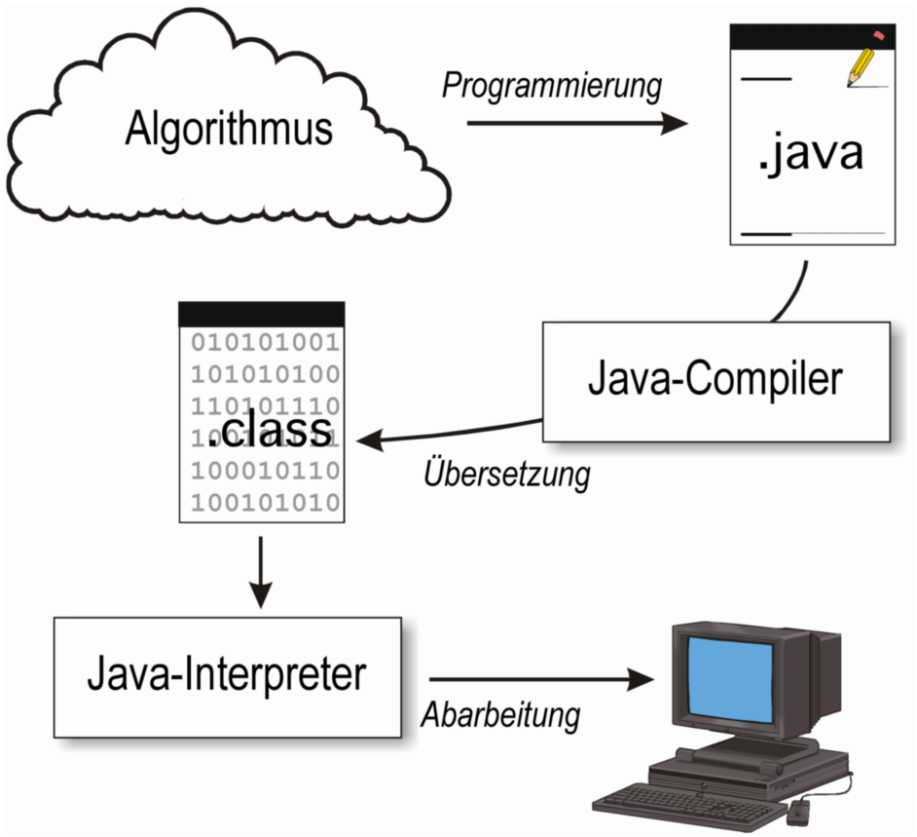


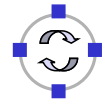
Plattformunabhängigkeit von Java /2

- Java-Compiler `javac`
  - Überprüft Quelltext auf Fehler
  - Übersetzt Quelltext in plattformneutralen Zwischencode (Bytecode)
- Java-Interpreter `java`
  - Interpretiert Bytecode
  - Implementiert virtuelle Java-Maschine

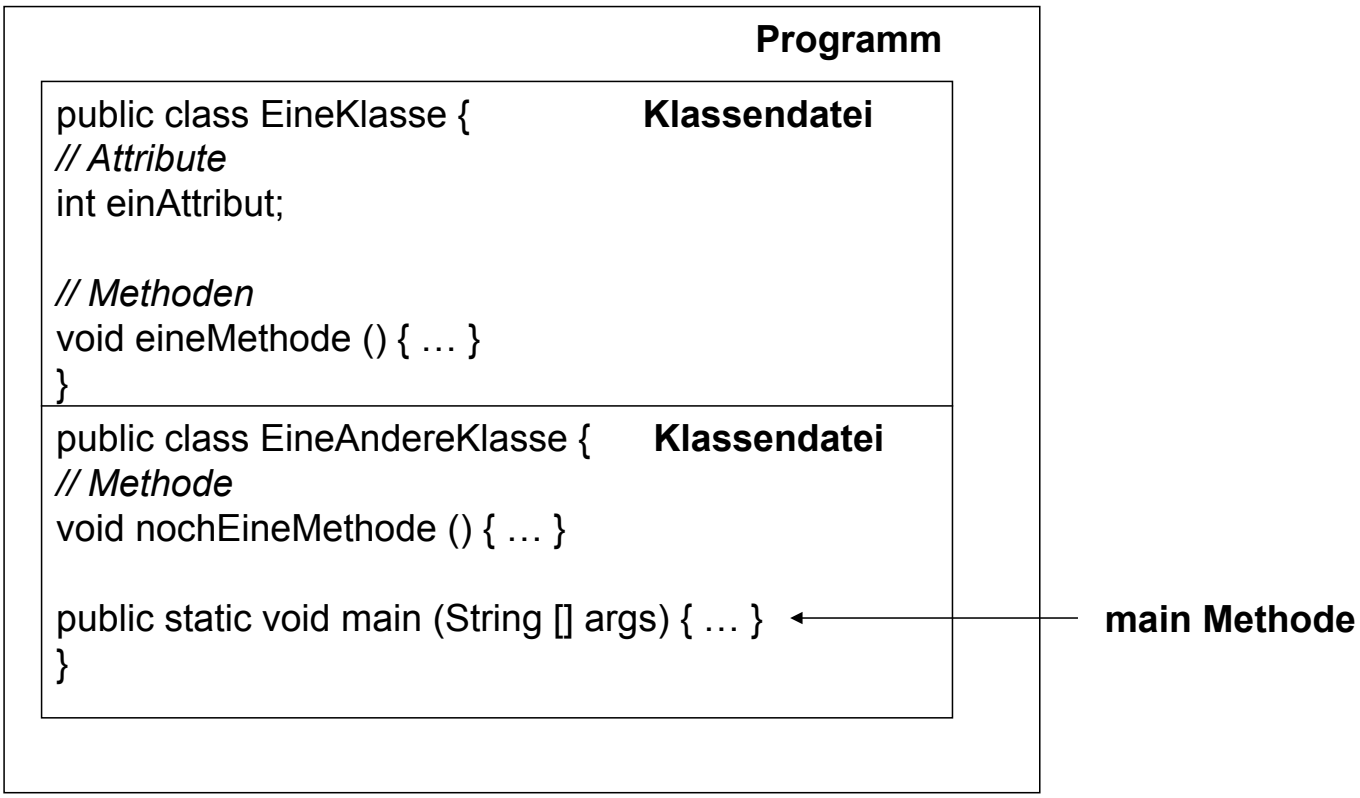


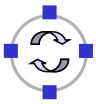
Java-Werkzeuge





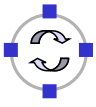
Struktur eines Java-Programms





Ein erstes Java-Programm

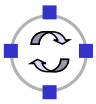
```
/* Hello.java - Das erste Java Programm. */

// Jedes Java-Programm besteht aus
// mind. einer Klasse.
public class Hello {
    // Eine Standalone-Anwendung muss
    // eine main-Methode besitzen
  public static void main(String[] args) {
    // Zeichenkette ausgeben
    System.out.println("Hello Java !");
  }
}
```



Übersetzung und Ausfürung

1. Quelltext in Datei `Hello.java` speichern
   - Dateiname entspricht Klassennamen!
   - Klasse muss eine Methode `main` als Startpunkt der Ausführung besitzen
2. Quelltext kompilieren:
   - Quelltext in Bytecode übersetzen
   - Liefert Datei `Hello.class`

```
> javac Hello.java
```



Übersetzung und Ausführung /2

1. Java-Programm ausführen
   - Interpretieren des Bytecodes

```
> java Hello
```

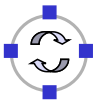
2. Ergebnis

```
Hello Java !
```

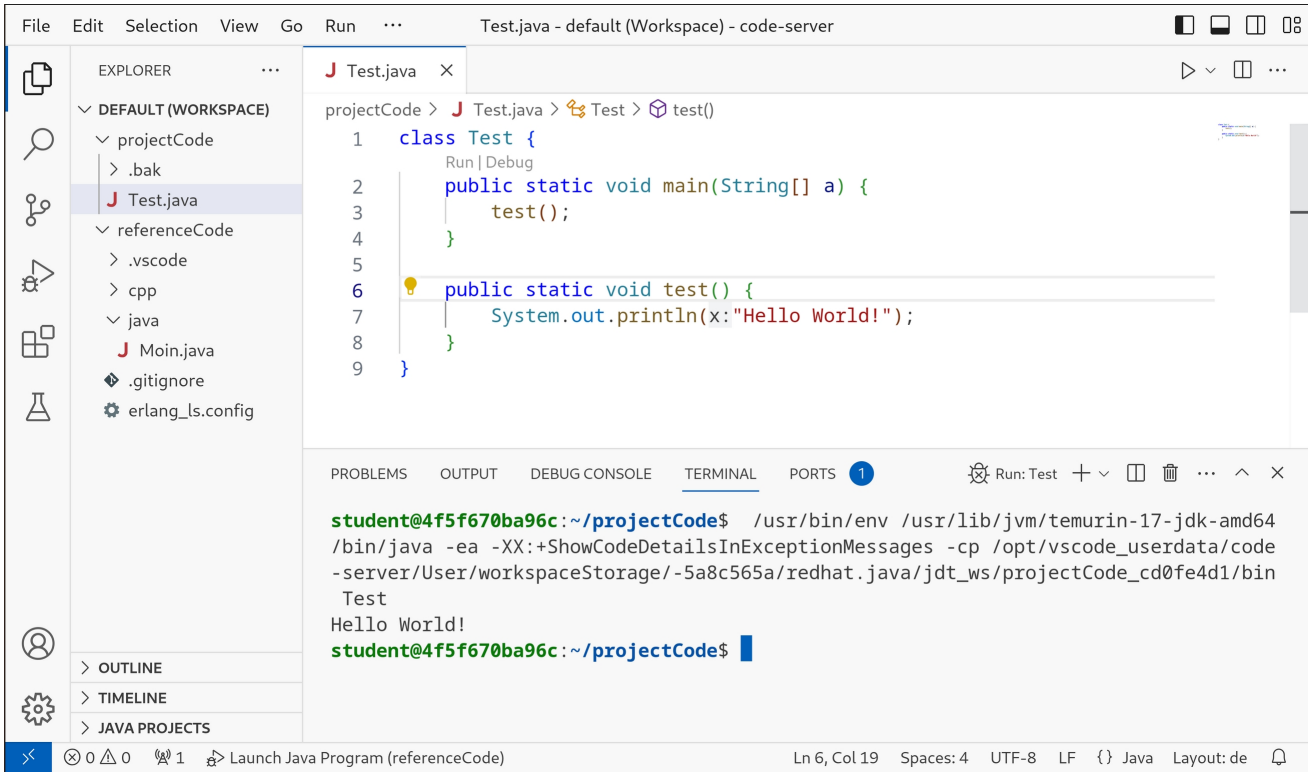


Entwicklungswerkzeuge (1)

- J2SE Software Development Kit (SDK)
  - Laufzeitumgebung, Klassenbibliothek, Basiswerkzeuge
  - `www.oracle.com/technetwork/java/index.html`
  - Für Windows, Linux and MacOS
  - Teilweise schon vorinstalliert (Linux, MacOS X)
- IntelliJ IDEA
  - Komfortable, graphische Entwicklungsumgebung (IDE) `(www.jetbrains.com/idea/)`
  - Sehr gute Unterstützung der Fehlersuche („Debugging“)
- Eclipse
  - Ähnlich zu IntelliJ IDEA `(www.eclipse.org)`
- Visual Studio Code (VSCode)
  - Erweiterbare IDE `(code.visualstudio.com)`



Entwicklungswerkzeuge (2)



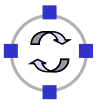


Ein- und Ausgabe mit Java

- Umfangreiche Klassenbibliothek für textuelle und grafische Ein-/Ausgabe sowie Dateiarbeit
- Textuelle Ausgabe über Methode `println` eines vordefinierten Ausgabeobjektes `out`

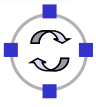
```
System.out.println("Hallo, " +
  "hier kommt eine "+ 42 + "!");
```

- Eingabe flexibel aber umständlich, daher „Vorlesungsklasse“ `aup.IOUtils` mit Methoden
  - zum Lesen von ganzzahligen Werten **`(int)`**: `readInt`
  - zum Lesen von Gleitkommawerten **`(double)`**: `readDouble`
  - zum Lesen von Zeichenketten **`(String)`**: `readString`



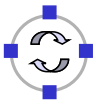
Ein- und Ausgabe mit Java: Beispiel

```
import aup.*; /* Klasse importieren */
public class Rechner {
    public static void main(String[] args) {
    // Variablen für Eingabe und Ergebnis
    int a, b, summe;
    // Werte einlesen
     a = IOUtils.readInt();
     b = IOUtils.readInt();
     // Summe berechnen
     summe = a + b;
     // Summe ausgeben
     System.out.println("Ergebnis = "+ summe);
    }
}
```



Datentypen in Java

- Definieren Struktur, Wertebereich und zulässige Operationen für Datenstrukturen
- Strenge Typisierung:
  - Jede Variable hat einen wohldefinierten Typ
  - Typumwandlung nur unter bestimmten Bedingungen zulässig
- Unterscheidung in
  - Primitive Datentypen
  - Referenzdatentypen



Primitive Datentypen

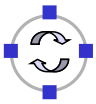
- „Eingebaute“ Datentypen
- Speicherung von Werten
- Statisch, nicht erweiterbar
- Arten
  - Numerisch (Ganzzahl, Gleitkommazahl)
  - Zeichen-Datentypen
  - Boolesche Datentypen



Numerische Datentypen

- Ganzzahlige Werte: **`byte, short, int, long`**
- Gleitkomma-Werte: **`float, double`**

Datentyp	Größe	Wertebereich
`byte`	8 Bit	- 128…127
`short`	16 Bit	- 32768…32767
`int`	32 Bit	$-2^{31}…2^{31}-1$
`long`	64 Bit	$-2^{63}…2^{63}-1$
`float`	32 Bit	$10^{-46}…10^{38}$ (6 sign. Stellen)
`double`	64 Bit	$10^{-324}…10^{308}$ (15 sign. Stellen)



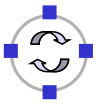
Zeichen-Datentypen

- **`char`**: vorzeichenloser 16-Bit-Integer-Typ
- Repräsentation von Zeichen im Unicode-Zeichensatz
- Wertebereich: ´\u0000´ bis ´\uffff´
- Repräsentation von Zeichenketten (Strings) durch eine Klasse `java.lang.String !!!`



Boolescher Datentyp

- **`boolean`**: 1 Bit-Werte
- Repräsentation von Wahrheitswerten
- Mögliche Werte: **`false`** und **`true`**
- Ergebnistyp für logische Vergleiche
- Konvertierung zwischen Integer-Werten und **`boolean`**-Werten ist nicht zulässig!



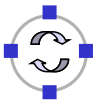
Bezeichner und Literale

- Bezeichner
  - Namen für Variablen, Klassen, Attribute, Methoden usw.
  - Beliebige Länge
  - Schlüsselworte von Java dürfen nicht als Bezeichner verwendet werden
  - Alle ASCII-Zeichen, _, `$`
- Literale (Konstanten)
  - Numerische Werte: `42, 345.4, 7.899E+34`
  - Zeichen: `´a´, ´\u0012´`
  - Zeichenketten: `"Ein String"`



Variablen

- Eine Variable ist ein:
  - benannter Speicherbereich (primitiver Datentyp) bzw.
  - benannter Verweis (Referenz-Datentyp) auf einen Speicherbereich, in dem Wert abgelegt werden kann
- Besitzt Typ: Wert und Operationen müssen Typ entsprechen
- Muss vor Verwendung deklariert werden:
  - Festlegung des Typs, Initialisierung
  - Bestimmt Sichtbarkeit



Variablendeklaration

- Notation

```
typ bezeichner [ = init_wert ];
```

- Vereinbarung
  - Überall in Methoden oder Anweisungsblöcken
  - Vorzugsweise zu Beginn eines Blocks
- Beispiele

```
int eine Variable = 23;
float a, b;
char c1 = 'A', c2 = 'B';
boolean aBoolValue = true;
```

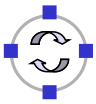


Sichtbarkeit von Variablen

- Alle Variablen sind in bestimmten Teilen des Programms sichtbar, in anderen eventuell aber nicht zugreifbar
- Der Bereich, in dem eine Variable sichtbar ist, wird häufig als „Scope“ oder „Gültigkeitsbereich“ bezeichnet

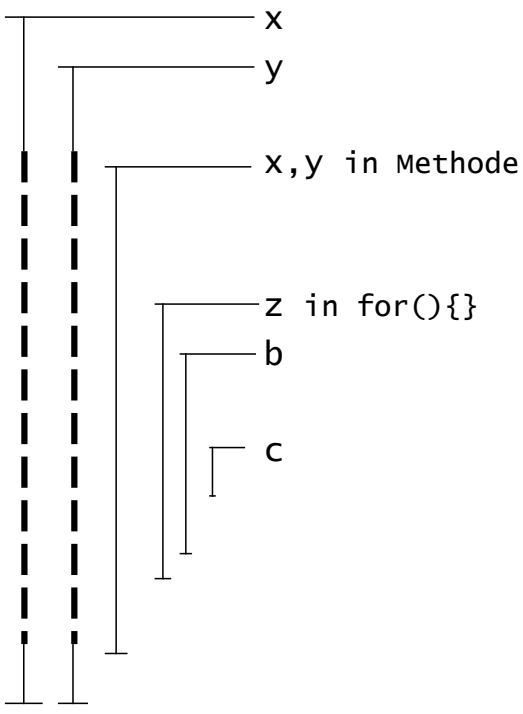
- In Java sind Klassen und Funktionen überall sichtbar (es sind auch Vorwärtsverweise möglich!)

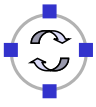
- Variablen haben eine eingeschränkte Sichtbarkeit
  - „globale“ Variablen sind in der gesamten Klasse sichtbar
  - „lokale“ Variablen sind lediglich in der aktuellen Methode oder sogar nur im aktuellen Block sichtbar
  - In Java ist eine Mehrfachverwendung von gleichen Variablennamen in geschachtelten Blöcken ***NICHT*** erlaubt



Beispiel zur Sichtbarkeit

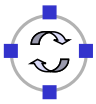
```
public class Sichtbarkeit{
  static int x = 1; // global
  static int y = 2; // global

  public int meineMethode( int x, int y){
     this.x = x;
     this.y = y;
     for (int z = 0; z < x; z++){
       int b = 1;
       {
           int c = 2;
       }
     }
     return y;
  }
}
```





Referenzdatentypen

- Variablen dieser Typen enthalten nicht Daten selbst, sondern Verweis (Referenz) auf Speicherort der Daten
- Standardwert: **`null`**
- Vergleich (==) und Zuweisung (=) erfolgt auf Referenzen!!
- Referenzdatentypen: Arten
  - Feld (Array): Datenstruktur fester Größe, die aus Elementen gleichen Typs (sowohl primitive als auch Referenzdatentypen) aufgebaut ist
  - Objektdatentyp: Repräsentation von Referenzen auf Objekte



Felder

- Felder = Referenzdatentypen (Verweis)

```
int einFeld[];
int[] auchEinFeld;
```

- Erfordert explizite Allokation
  - **`new`**-Operator zur Bereitstellung des Speicherplatzes
  - Direkte Instantiierung mit Literalen
  - Zuweisung einer anderen Feld-Referenz



Felder - Allokierung und Zugriff

- Allokierung
  - **New**-Operator

```
int[] feld = new int [20];
```

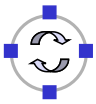
  - Initialisierung mit Literalen

```
int[] feld = { 5, 23, 42 };
```

  - Zuweisung

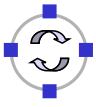
```
int[] nochEinFeld = feld;
```

- Zugriff
  - auf *i* -tes Element mit `feld[i]` mit
    i = 0. . . `feld.length` - 1



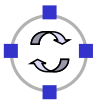
Operatoren

- Unäre Operatoren
  ++ (Inkrement) -- (Dekrement) ! (Komplement)
  - Bsp.: `a++` (entspricht `a = a+1`)
    `--b`
    `!true`
- Arithmetische Operatoren
  `+ - * / %` (Rest bei Division)
  - Bsp.: `45 * c / (d - 34)`
- Vergleichsoperatoren
  `< > <= >= ==` (Gleichheit) `!=` (Ungleichheit)
  - Bsp.: `34 >= c  x == y`
- Logische Vergleichsoperatoren
  `&&` (UND) || (ODER)
  - Bsp.: `a > b && c == true`



Operatoren /2

- Zuweisungsoperatoren
  = (Zuweisung) `+=` `-=` `*=` `/=` . . .
  - Bsp.: `a = 34`
    `b += 1` (entspricht `b = b + 1`)
- Verschiebeoperatoren (Bitweises Verschieben)
  `<<`
  `>>`
- Bitoperatoren
  `&` (bitweises UND)
  `|` (ODER)



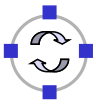
Umsetzung von Algorithmen in Java

- Anweisungen als elementare Arbeitsschritte für Algorithmen
- Umsetzung in Programmiersprachen
  - Operationen auf Datentypen + Wertzuweisungen
  - Kontrollstrukturen
- Strukturierungsmittel (Prozeduren, Methoden)



Ausdrücke

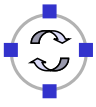
- Wertzuweisungen, Berechnungen
- Primäre Ausdrücke
  - Namen, Konstanten:
    `x  eineVariable  42`
  - Feldzugriffe:
    `feld[23]`
  - Methodenaufrufe:
    `obj.methode()`
  - Allokationen (Erzeugen von Feldern und Objekten)
    `new int[5]  new Vector()`
- Zusammengesetzte Ausdrücke
  - Durch Verwendung von Operatoren
  - Bedingter Ausdruck: `<Bedingung> ? <Dann> : <Sonst>`
    Bsp.: `int absOfx = x<0 ? -x : x;`



Anweisungen

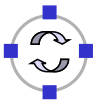
- Arbeitsschritt eines Programms
- Durch „;“ abgeschlossen
- Arten
  - Leere Anweisung
  - Ausdrücke
  - Bedingungen, Schleifen, Sprünge
  - . . .
- Zusammenfassung mehrerer Anweisungen zu einem Block durch Klammerung

```
{ anw1; anw2; ...anw3; }
```



Kontrollstrukturen

- Bedingungs- und Auswahlanweisungen
  - **`if-else`**
  - **`switch`**
- Schleifenanweisungen
  - **`while, do-while`**
  - **`for`**
- Sprunganweisungen
  - **`break, continue, return`**



if-else-Anweisung

- Syntax

```
if ( bedingung ) anw1; [ else anw2; ]
```

- Anweisung bzw. Anweisungsblock *anw1* wird nur ausgeführt,
  - wenn Bedingungsausdruck *bedingung* erfüllt ist (**`true`** liefert),
  - andernfalls *anw2* (sofern else-zweig vorhanden)
- Beispiele

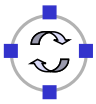
```
if (x == 0) y = 3; else y = -3;
if (x > y) {
     m = x; n = 2 * y;
}
```



switch-Anweisung

- Syntax

```
switch ( ausdruck ) {
 case auswahlwert1:
   // Anweisungen für Fall #1
   [ break; ]
 case auswahlwert2:
   // Anweisungen für Fall #2
   [ break; ]
 default:
   // Anweisungen für alle anderen Fälle
   [ break; ]
}
```



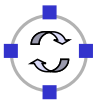
switch-Anweisung /2

- Verallgemeinerte Form der Fallunterscheidung
- Für jeden Wert des Ausdrucks ausdruck eigene Anweisungsfolge
- Ausdruck muss ganzzahligen Wert liefern: **`char`**, **`byte`**, **`short`**, **`int`**
  - Seit Java SDK7: **`String`** ebenfalls als Ausdruck erlaubt
- Auswahlwerte: Literale (also konstant während Programmablauf)
- Ohne **`break`**: Kontrollfluss zu nächster Anweisung
- **`default`**-Zweig ist optional



switch-Anweisung: Beispiel

```
int i = ...;
switch (i) {
 case 0:
         System.out.println("Null");
         break;
 case 1:
 case 2:
         System.out.println("eins oder zwei");
         break;
 default:
         System.out.println("größer als zwei");
         break;
}
```



while-Schleife

- Bedingte Schleife mit Bedingungsprüfung am Anfang
- Syntax:

```
while ( bedingung ) anweisung;
```

- Solange der Ausdruck *bedingung* erfüllt ist (**true**), wird die Anweisung bzw. der Anweisungsblock *anweisung* ausgeführt
- Überprüfung der Bedingung vor jedem Durchlauf
  - d. h. wenn die Bedingung nicht erfüllt ist, wird Anweisung nicht ausgeführt



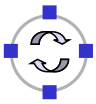
while-Schleife: Beispiele

- Zählschleife: 5 ... 1

```
int i = 5;
while (i > 0) {
  System.out.println(i);
  i--;
}
```

- Endlosschleife

```
while (true)
  System.out.println("Endlos ...");
```



do-while-Schleife

- Bedingte Schleife mit Bedingungsprüfung am Ende des Durchlaufs
- Syntax:

```
do anweisung; while ( bedingung );
```

- Anweisung bzw. Anweisungsblock *anweisung* wird ausgeführt, solange der Ausdruck bedingung erfüllt ist (**true**)
- Überprüfung der Bedingung nach jedem Durchlauf
  - d. h. Schleifenrumpf wird mindestens einmal ausgeführt



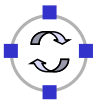
do-while-Schleife: Beispiele

- Zählschleife: 1...5

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

- Endlosschleife

```
do System.out.println("Endlos");
while (true);
```



for-Schleife

- Schleife mit Zählvariablen
- Syntax

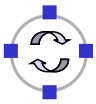
```
for (init_ausdr ; bed_ausdr ; mod_ausdr )
     anweisung;
```

- Initialisierungsausdruck *init_ausdr*
  - Zählvariable auf Startwert setzen (und evtl. deklarieren)
- Bedingungsausdruck *bed_ausdr*
  - Bedingung für Fortsetzung (Ausführung des Schleifenrumpfes)
- Modifikationsausdruck *mod_ausdr*
  - Verändern der Zählvariablen



for-Schleife: Semantik

1. Zählvariable deklarieren und initialisieren
2. Solange Bedingungsausdruck erfüllt ist (**`true`** liefert)
3. Führe Anweisung bzw. Anweisungsblock *`anweisung`* aus
4. Berechne Modifikationsausdruck (Verändern der Zählvariablen)



for-Schleife: Beispiele

- Zählschleife

```
for (int i = 0; i < 5; i++)
      System.out.println(i);
```

- Endlosschleife

```
for (;;) System.out.println("Endlos");
```

- Weitere Möglichkeiten

```
int i = 0, n = 0;
for (; i < 5 && n < 10; i++, n += 3) {
      // ...
}
```



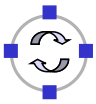
return-Anweisung

- Beenden der Methodenausführung
- Rücksprung zur Aufrufstelle
- Rückgabe eines Wertes (Ergebnis)
- Syntax

```
return [ ausdruck ];
```

- Beispiel

```
int plus(int a, int b) {
  return a + b;
}
```



break-Anweisung

- Unterbrechen des Kontrollflusses des aktuellen Blocks
- Fortsetzung nach Anweisungsblock
- Beispiel:

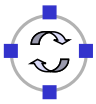
```
int i = 0;
while (true) {
  if (i == 10)
    break;
  // ...
}
```



continue-Anweisung

- Unterbrechen des Kontrollflusses des aktuellen Blocks
- Fortsetzung der Schleife mit der nächsten Iteration
- Beispiel:

```
for (int i = 0; i < 10; i++) {
  if (i % 2 == 0)
     continue;
  System.out.println(i);
}
```



Methoden

- Logische Einheit von einer oder mehreren Anweisungen
- Immer im Kontext einer Klasse definiert!
- Definiert Verhalten von Objekten
- Entspricht Funktion oder Prozedur
- Kann mit Parametern aufgerufen werden
- Kann Ergebnis (z. B. einer Berechnung) liefern
- Spezielle Methode `main` ⇝ Hauptprogramm

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



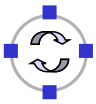
Methodendefinition

- Notation

```
sichtbarkeit [static]datentyp name(parameterliste) {
  anweisungen
}
```

- Sichtbarkeit: **`public`** (öffentlich), **`protected`** (sichtbar in Objekten gleicher und abgeleiteter Klassen), **`private`** (nicht-öffentlich)
- **`static`** Klassenmethode (nicht auf Objekt!)
- Datentyp des Rückgabewertes oder **`void`** (keine Rückgabe)
- Parameterliste:

```
typ1 name1, ..., typN nameN
```



Java-Beispiel: Fakultät rekursiv

```
import aup.IOUtils;
public class FacRecursive {
  public static int factorial(int x) {
    if (x <= 1) return 1;
    else return x * factorial(x - 1);
  }
  public static void main(String[] args) {
    int z;
    System.out.print("Zahl: ");
    z = IOUtils.readInt();
    System.out.println("Fakultät(" + z + ") = " +
      factorial(z));
  }
}
```



Zusammenfassung

- Objektorientierte Programmiersprache Java
- Entwickelt auf der Grundlage von C++
- Übersetzung in plattformunabhängigen Bytecode (`javac`)
- Ausführung in virtueller Maschine (`java`)
- Zahlreiche Werkzeuge
- Programmieren in Java
  - Datentypen in Java
  - Umsetzung von Algorithmen

