Einführung in die Programmierung

1) Programmiersprachen

- primitivste Sprachen sind die Assemblersprachen
- Maschinencode ist effizient ausführbar
- Programme können interpretiert oder kompiliert werden (übersetzt)

Assemblercode - wird übersetzt in Maschinencode

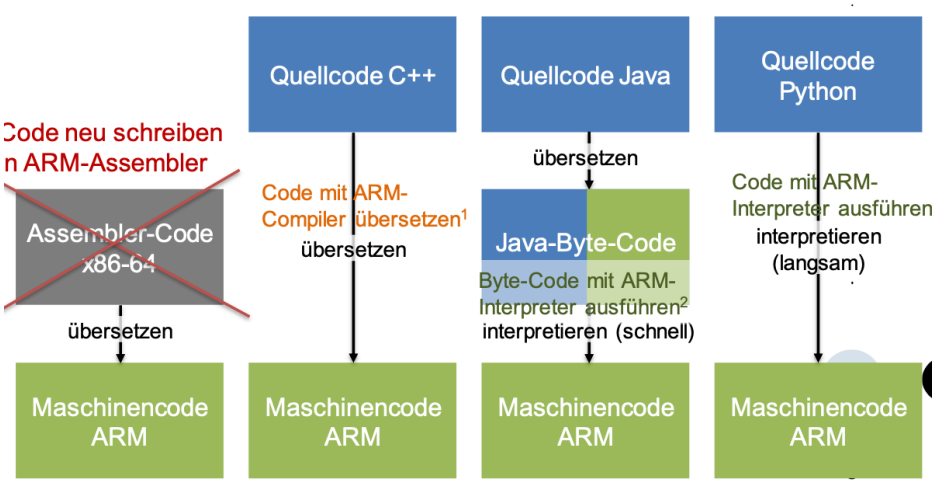
- Compiler übersetzt Quellprogramm in Maschinencode
- Maschinencode wird gespeichert und kann vom Rechner direkt ausgeführt werden
- Effizient, Compiler und Maschinencode sind aber Rechnerabhängig

Quellcode C++ - kompiliert (übersetzt) - in Maschinencode (plattformunabhängig)

Interpretieren:
- Quellcode wird Zeile für Zeile übersetzt und direkt ausgeführt
- Maschinencode wird nicht gespeichert
- Teile, die mehrfach durchlaufen werden, werden auch mehrfach übersetzt
- Vorteil: kann direkt auf Zielmaschine portiert werden, die einen Interpreter besitzt (Python)

Quellcode Python - interpretiert (langsam) - in Maschinencode (plattformunabhängig)

Mischform:
- Compiler übersetzt in maschinennahes Zwischenformat (Bytecode)
- Byte-Code-Interpreter=Java Virtual Machine(Ausführungsumgebung für Java-Bytecode)
- Portable Code, der trotzdem schnell ausgeführt wird



[1]Nur das eine, neu-übersetzte Programm funktioniert jetzt auf ARM
[2]Alle Programme, die es gibt, funktionieren jetzt auf ARM

2) Typumwandlung

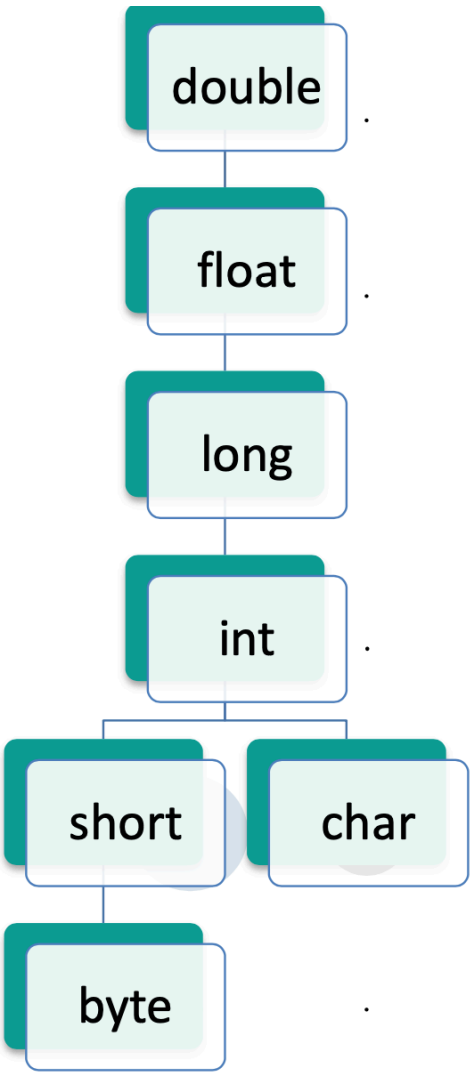
- Ganzzahltypen < Gleitkommazahltypen
- Ganzzahltype: byte < short < int long
- Gleitkommazahl: float y double
- Char < int

Upcast enger Typ zu weiterem Typ (höher)
Implizit

Downcast (weiter Typ zu engerem Typ (niedriger)
-> kann zu Präzisionsverlust führen
Explizit - muss erzwungen werden

Long anotherNumber = 42L;
aNumber = (int) anotherNumber;

Boolean kann nicht konvertiert werden, und kein anderer Typ kann in boolean konvertiert werden



Untertitel

3) Nützliche Formatierungen

%.2f (gerundet)	zwei Nachkommastellen
%10s	Auffüllen auf Breite 10
%10.2f	Kombination von beiden
%+d	Vorzeichen einer Wahl erzwingen
%,d	Tausendertrenner

4) Bedingungsoperator (ternärer Operator)

Syntax: c ? a : b

- Wert von a, falls Wert von c == true
- Wert von b, falls Wert von c == false

Primitive Datentypen in Java

Typ	Länge	Kleinste darstellbare Zahl	Größte darstellbare Zahl
byte	8 Bit	-128	127
short	16 Bit	-32.768	32.767
int	32 Bit	-2.147.483.648	2.147.483.647
long	64 Bit	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
float	32 Bit	ca. $(-2) \times 2^{127}$	ca. 2×2^{127}
double	64 Bit	ca. -10^{308}	ca. 10^{308}
char	16 Bit	Unicode-Zeichen	
boolean	32 Bit	false	true

Untertitel

5) Zuweisungsoperator

X += y x = x + y
X -= y x = x - y …

6) Kontrollstrukturen und Anweisungen

If - Anweisung

```
If (<Bedingung>) {
        <Anweisung(en)>                    //Falls Bedingung true
} else {
        <Anweisung(en)>                    //Falls Bedingung false
}
```

```
If (<Bedingung1>) {
        <Anweisung(en)>                    //Falls Bedingung1 true
} else if (<Bedingung2>){
        <Anweisung(en)>                    //Falls Bedingung1 false
                                           //Bedingung2 true
}
} else if (<Bedingung3>){
        <Anweisung(en)>                    //Falls Bedingung false
                                           //Bedingung2 false
                                           //Bedingung3 true
}
} else
        <Anweisung(en)> }                  //Falls alle Bedingung false
```

Switch - Case

```
Switch (<Ausdruck>) {
        case <Konstante>:
        <Anweisungsfolge>
        break;
…
        default:                                //darf auch wegfallen
        <Anweisungsfolge>
```

- Konstante, die den Wert des Ausdrucks entspricht, wird fortgeführt bis zum break
- Nach break; geht es am Ende des switch - Blocks weiter

7) Schleifen

- Folge von Anweisungen wird mehrfach durchlaufen
- For - Schleife
- While - Schleife
- Do while - Schleife
- For each - Schleife
- Alle schleifen haben eine Abbruchbedingung

While - Schleife

```
Int x = 1;

While (<Bedingung>) {                   1. Bedingung auswerten
        <Anweisung(en)>                 Bedingung false - Schleife Abbrechen
Z.B if Anweisung
}                                       Bedingung true - Anweisung ausführen und bei 1 weiter
x++; (Update)
```

- Schleifenvariable muss außerhalb deklariert werden und im Block aktualisiert werden

For - Schleife

```
for(int i = 1 ;  i < 20; i++){          Bedingung false - Schleife Abbrechen
        <Anweisung(en)>                 Bedingung true - Anweisung ausführen und
                                        bei 2 weiter
}
```

„Führe Anweisungen für alle Werte i von 1 (inkluse) bis 20 (exklusiv) aus“

```
Int i = 1;
While (i < 20){
        <Anweisung(en)>
}
i++;
```

- *Schleifen können verschachtelt werden - sind schneller wie if Anweisungen*
- *Erste Schleife zu erst dann die Inneren*
- *Jede for Schleife kann durch eine while Schleife ersetzt werden*

Do-while-Schleife

- Bedingung wird **nach** erster Ausführung des Anweisungsblocks geprüft (anders al bei while!!)

```
Do {
        <Anweisung(en)>
} while (<Bedingung>)
```

1. Anweisung ausführen
2. Bedingung ausführen

A. Bedingung false - Schleife abbrechen

B. Bedingung true - weiter mit 1

- Falls Schleifen **immer true** ist, **terminiert** die Schleife **nich**t - Abbruch mit **break** erzwingen!
- **continue** bricht Ausführung des Schleifenrumpfs ab und springt zum Schleifenkopf

8) Arrays

- Zusammenfassung mehrer Variablen gleichen Typs
- Unterscheidung durch Index
- Array Variablen heißen Elemente - appointments[0] …
- Erstes Element mit **Index 0** - nur **positive Werte** für Index - Index bestimmt **Position** innerhalb des Arrays
- Array ~ eindimensionale Tabelle
- Es gibt auch mehrdimensionale Arrays

```
int[] numbers;
String [] names; …
Numbers = new numbers[42]                    //erzeugt int Array mit Komponente 0…41
Int numbers[] = {…};
```

```
Typ[] Variablennamen;
Typ[] Variablennamen = new Typ [Arraygröße];
```

```
Typ[][] Variablenname;
Variablenname = new String[][];
```

- Wertzuweisung variablennamen[…] = „….“;
- Verkürzung mit Arrays mittel Schleifen möglich
- { } Arrayinitialisierer
- Maximaler Index eines Arrays ist array.length - 1
- System.arraycopy() oder mittels Schleifen kopieren oder .clone()

Primitive Datentypen

- enthalten nur einen Wert (Skala)
- boolean, char, byte, short, int, long, float, double

Referenzdatentypen

- Strings
- Arrays
- Klassen

9) Schleifen für Arrayelemente

Schleife mit aufzählen

```
int[ ] = numbers = {…}
for (int i = 0; i < numbers.length; I++){
        System.out.println(numbers[I]);
}
```

For each:

```
For (int number : numbers){
        System.out.println(number);
}
```

```
int[ ] = numbers = {…}                                          //Zugriff auf Index Array-Eintrag
for (int i = 0; i < numbers.length; I++){
        System.out.println(„Die „ + (I+1)
                                + „-te Zahl) + numbers[I]);
}
```

Mehrdimensionale

String [ ] [ ] teams = {…}

```
For (String team : teams){
        System.out.println(„Team: „);
        for(String member: team){
                System.out.println( „-„ + member);

  }
}
```

10) Enum

```
Public enum Color {
BLUE, GREEN, BLACK, ….
}
```

```
for(Color colors: available){
        Syso ( colors + „ „);
}
```

```
Color color = Color.BLACK;
Color[ ] available = { available.BLUE, available.GREEN, …};
```

11) Zufallszahlen

- Math.random( ) * 10 [0…10]
- (Int) (Math.random( ) * 10) Ganzzahlen 0 … 9
- ((Int) (Math.random( ) * 21) -10) Ganzzahlen -10…10

12) Klassen und Objekte

- Zusammenfassung verschiedener Variablen
- Zusammenfassung von Daten und Operationen auf Daten
- Instanz einer Klasse heißt Objekt

```
Public class <Klassenname> {

Public <Datentyp> <Attributnname>;
Public <Datentyp> <Methodenname>; …
}
```

Objekterzeugung

```
<Klassenname> <Instanzname> = new <Klassenname>( );
```

13) Konstruktor

- Default construcotr public <Klassenname>( )
- Dieser hat keine Argumente
- Man kann ihn definieren dort ggf. Anweisungen durchführen

Konstruktor mit Parametern

```
Public <Klassenname>( String …, String …, int …){
        this.<…> = …;
}                                       //this. Enthält Referenz auf die Instanz, zu der die
                                        ausgeführte Methode/Konstruktor gehört
```

14) Methoden

```
Public <Rückgabewert> <Methodenname> (<Typ> <Name>, ….){
…}

Public String <name> (String …, int…,….)
Return this.(…) + ….
}

Public void set… (Parameter…){
This.(…) = ….;
…
}
```

- Zugriff auf Attribute und Methoden mit Punktnotation
- <Instanzname>.<Attributname>
- <Instanzname>.<Methodenname>

Statische Methoden

Public static <Datentyp> <Methodenname> (<FormaleParameter>){
…}

```
Public class myMath {
Public static int max(int a, int b) {

        if(a > b) {
                Return a;
        } else {
                Return b;
    }
  }
}
```

Aufruf über Klassenname: Syso(myMath.max(15, 19));

15) Interfaces

- Klassen dienen der Implementierung
- Interfaces dienen der Modellierung/Spezifikation
- Ein Interface kann von verschiedenen Klassen implementiert werden
- Ziel: Weniger Abhängigkeit von bestimmten Implementierungen

- Nur Signaturvorgabe
- Keine Attribute, kein Konstrukt
- Klassen, die ein Interface implementieren müssen alle Methoden des Interfaces definieren
- <Klassenname> implements <Interface>
- @Override bei Methoden
- Interfaces können andere Interfaces Extenden (… extends …)
- Typprüfung mit instanceOf

If (… instanceOf <Typname>) { … }

16) Vererbung

- **class Unterklasse extends Oberklasse { … }**
- *Subclasses* erben Attribute und Methoden der *superclasses*
- A extends B
- Eine klasse kann höchstens von einer anderen Klasse erben
- Spezialisierung: Klasse A die von Klasse B erbt, ist spezieller, da sie mindestens alles von B übernimmt, aber ggf. mehr anbietet

- Unterklassen können:
    - Neue Attribute einführen
    - Neue Methoden einführen
    - Bestehende Methoden neu implementieren
    - Implementierung der Oberklasse nutzen

- super (…) ruft Konstruktor der Oberklasse aufgerufen
- Muss im Konstrukt der Unterklasse als erster Befehl stehen
- Ein super ( ) kann weggelassen werden, wenn die Oberklasse einen paramterlosen Konstruktor definiert (default Konstruktor)
-

17) Abstrakte Klassen

- können Methoden ohne Rumpf definieren (abstrakte Methoden)
- Vorlage für Implementierung
- Können nicht instanziiert werden
- eine klasse die von einer abstrakten Klasse erbt, muss alle abstrakten Methoden überschreiben und implementieren
- Implementierung von Methoden möglich (anders als bei Interfaces)
- Attribute erlaubt (bei Interfaces nicht)
- Kennzeichen mittels **abstract**

```
public abstract class (…) {

        public abstract double ..( );  //Kein Rumpf
        public String toString ( ) { return … }
```

18) Typumwandlung

- **implizit : upcast** jederzeit möglich
    - Umwandlung in Oberklasse
- **explizit: downcast**
    - Umwandlung in Unterklasse über cast-Operator

-Typüberprüfung mittels instanceOF

Methoden überschreiben mit @Override zur Kennzeichnung für Compiler (versehentliches Überladen verhindern)

19) Kapselung

- Öffentlichen Zugriff minimieren
- Nur absolut notwendige öffentlich
- Attribute verborgen - getter/setter
- Vorteil: Implementierung kann lokal geändert werden
- Variablen verstecken:
    - Private, protected, -default, public

Public:
- Können überall auch in anderen Pakten genutzt werden
- Attribute, methode, Konstruktoren die public sind, könne überall dort zugegriffen werden, wo auf klasse zugergriffen werden kann

Protected:
- zugriff von eigenen Klassen und andere innerhalb desselben Pakets
- Attribute und Methoden werden an Subklassen weitervererbt und sind dort zugänglich

Private:
- können nur in der eigenen Klasse verwendet werden

20) Getter/Setter

- Variablen private
- Zugriff über

getAttribut( ) { return this.attribut; } //Holt Attribut. **Lesezugriff**

setAttribut(Type neuerWert) { this.attribut = neuerWert; } //Bestimmt Attribut **Schreibzugriff**

- nur getter: kein nachträgliches Ändern möglich
- Variablentyp und -nutzung kann innerhalb der Klasse anders implementiert werden

21) Exceptions

- An Fehlerstelle mit throw new <ExceptionTyp> (<Message>)
- IllegalArgumentException
    - Erbt von Exception über mehrere Subklassen
    - In java sek vorhandene exception für **ungültige Methodenparameter**

```
Public int divide(…){

if(b == 0 ) {
        throw new IllegalArgumentException ( „…“);
}
Return a/b;
}
```

- Fehlerursache aufrufbar mittels public String getMessage( )
- Fehlerort abrufbar mittels public void printStackTrace( )

- Exception abfangen mit: … throws IOException { … }
- Exception verschlucken mit **try catch**

- Mehrere Blöcke möglich bei catch (mehrere Exception abfangen)

```
Try {
Scanner ….;
} catch (IOException e) | …) { Soso.(… e.getMessage( ) );
```

Checked vs unchecked

Unchecked:
- RuntimeException müssen nicht behandelt werden (können aber)
  - NullPointerException: Parameter ist null
  - IllegalArgumentException: Parameter ist nicht null, aber ungültiger Wert
  - IndexOutOfBoundsException: Index außerhalb gültigen Bereichs (Array)

…

Checked:
- IOException
- FileNotFoundException
- ClassNotFoundException

…

Zusammenfassung
- Werfen einer Exception beendet den Programmfluss und zeigt Fehler
- Können über Methodensignaturen an Methodenaufrufer weitergeleitet werden
- Behandlung durch try catch (catch oder Multi-catch)

22) Parameterzahl

Summe berechnen mit Arrays

```
Public static int sumUp( int [ ] values) {
Int sum = 0;
For( int a: values) {
        Sum += a;
        }
        Return sum;
}
```

Oder

```
Public static int sumUp(int [ ] values) {
Int sum = 0;
For( int i = 0; values.length; i++) {
        sum += values[I];
}
Return sum;
}
```

Übergabe:

```
Int sum = sumUp(new int[ ]  {….});          //Nachteil Werte müssen in Array verpackt werden
```

Variable Parameterzahl:

```
Public static int sumUp( int… values) { Varargs
Int sum = 0;
For( int a: values) {
        Sum += a;
        }
        Return sum;
}
 Int sum = sumUp(…);
```

Vorteil:

- Beliebig viele Paramterzahlen des gleichen Typs
- Parameter werden automatisch in Array verpackt

```
Public static int sumUp(int a, int b , int … values) { … }       //Mindestens x Parameter
```

Überladen:

```
Public static int sumUp(int a, int b , int c) { … }
```

23) Rekursion

- Eine Methode M heißt **rekursiv**, wenn sie sich direkt (rekursiver Aufruf) oder indirekt (verschränkt rekursiver Aufruf) selbst aufruft
- Ansonsten heißt M **iterativ:** Methode besteht nur aus Anweisungen, Schleifen und/oder Aufrufen nicht rekursiver Methoden; es gibt keine (direkten oder indirekten) Aufrufe

- Eine rekursive Methode braucht **immer eine Abbruchbedingung - Rekursionanker**
- Wenn Abbruchbedingung fehlt: StackOverFlow

```
Int factorial( int n ) {
        if( n == 1) { return 1; }                    //Abbruchbedingung
Return factorial(n-1) * n;
}
```

- Aufruf einer Methode in ihrer eigenen Implementierung

24) Generics

Typsicherheit

- Alle Elemente des Arrays haben denselben Typ
- Im Konstruktor kann man nur Elemente eines konkreten Typs mitgeben
- Typsicherheit beim Zugriff auf Elemente

Möglich mit Generics

- **Platzhalter** für Elementtyp
- Beim instanziieren **Ersetzen durch konkreten Typ**

- **Platzhalter T** (kann auch anders benannt werden)
- <T> im Anschluss an Klassennamen

public class generischesQuartett **<T>** {

Private **T** [ ] karten;
Public generischesQuartett (**T**… Karten) {
    this.karten = karten;
}

Public **T** drawACard ( ) {
    return this.karten
    [(int) (Math.random( ) * this.karten.length)];
}

- Bei Instanziierung mit **T durch konkreten Typ ersetzt**
- < > = Diamant Operator

genereischesQuartett <AutoKarte> autoQuartett = new generischesQuartett **<AutoKarte>**(…)

- Klassen zur Verwaltung von Daten eines oder mehreren Datentypen (zB Paare, Mengen und Listen von Elementen eines Typs)
- Jaa Collection Klassen: ArrayList<T>, new ArrayList<AutoKarte> etc.

- **< T extends Oberklasse>** schränkt Typ ein
- T muss jetzt Oberklasse selbst oder ein von Oberklasse abgeleiteter Typ sein

Public class Sonder < T extends AutoKarte> { … Zugriff auf Attribute/Methoden von AutoKarte}

Sonder <AutoKarte> auto = new Sonder<AutoKarte> ( ) ;

Mehrere Typparameter

Public class KeyValue <K, V> {
Public K key; //besser private mit get Methode
Public V value;

Public JeyValue(K key, V value) {
    this.key = key;
    this.value = value;
  }
}

KeyValue <String, Integer> nameAge = new KeyValue<String, Integer>(„Chris“, 23);
String name = nameAge.key;
Int age = nameAge.value;