



Programmierung 2

Vorlesung 1: Grundlagen, Übergang von C
April 2022

alexander.gepperth@cs.hs-fulda.de



Von C nach Java: quick & dirty



C

- C ist eine imperative Sprache
 - Fokus auf konkreter Umsetzung: Funktionen, Schleifen, Variablen, Bedingungen,
 - maximale Freiheit für Programmierer*innen
 - kaum Einschränkungen (Projektstruktur, Programmstruktur, ...)
- C wird direkt in Maschinensprache kompiliert



Java

- Java ist eine objektorientierte Sprache
 - enthält alle Sprachelemente von C
 - alle wichtigen Konstrukte 1:1 übernommen
 - zusätzliche Konzepte: Klassen, Packages, Projekte
 - Freiheit der Programmierer*innen eingeschränkt zugunsten guter Projekt/Programmstruktur
- Java wird in **Bytecode** für die **JVM** kompiliert



Vom Programm zur Klasse

- Code und Daten müssen in Java stets Teile von **Klassen** sein
 - Code: **Methoden** (statt Funktionen)
 - Daten: **Attribute** (statt Variablen)
- Code außerhalb einer Klasse ist nicht möglich!
- **Beispiel:** Minimale Java- und C-Programme



Beispiel: Migration von C- Programm in ein Java-Projekt

- c_prog.c, JavaClass.java im E-Learning!



Beispiel: wir programmieren in Java wie in C

- large_c_prog.c, LargeJavaClass.java im E-Learning!



Zwischenfazit

- In Java kann man wie in C programmieren

C	Java
Programmdatei mit Funktion <code>main()</code>	Klasse mit Methode <code>main()</code>
Globale Variablen	<code>static-</code> Klassenvariablen
(Globale) Funktionen	<code>static-</code> Methoden
Lokale Variablen	lokale Variablen
Pointer	(Referenzen)
<code>#include</code>	<code>import</code>



Aber: Vorsicht!

- Java ähnelt C sehr, quasi alle Konstrukte haben identische Syntax
- Trotzdem große Unterschiede „unter der Oberfläche“!
- Objektorientierte Programmierung mit Klassen und Instanzen erfordert Umdenken!



Cut: Q&A



Klassen und Instanzen



Kap. 5.1, 5.4

Was ist eine Klasse?

- Klassen definieren neue Datentypen (wie ein C-struct)
- Klassen enthalten Code (**Methoden**, **Konstruktoren**) und **Daten** (Attribute), die thematisch zusammengehören
- Von jeder Klasse können beliebig viele **Instanzen** erzeugt werden





Kap. 5.1, 5.4

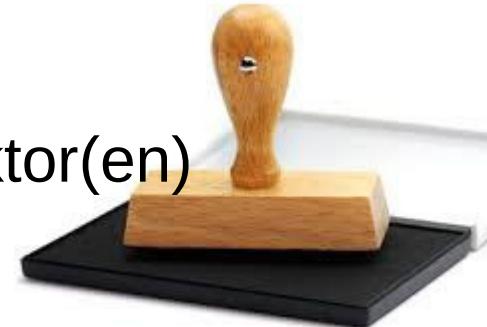
Was enthält eine Klasse?

```
public class V1 {  
  
    public int attribut ;  
  
    public V1() {  
        this.attribut = 0 ;  
    }  
  
    public V1(int k) {  
        this.attribut = k ;  
    }  
  
    public int getAttribut() {  
        return this.attribut ;  
    }  
}
```

Attribut(e)

Konstruktor(en)

Methode(n)





Kap. 5.1, 5.4

Was enthält eine Klasse?

- Definiert u.a.
 - einen oder mehrere **Konstruktoren**: werden aufgerufen bei Instanzierung
 - **Methoden** (`main`-Methode wird bei Programmstart aufgerufen)
 - **Attribute**, potentiell mit Startwerten
 - Zugriffsrechte für Klassen, Methoden und Attribute





Kap. 5.1, 5.4

OOP: Klasseninstanzierung

- Instanzierung mit `new`: erzeugt unabhängige **Instanz** der Klasse

```
v1 ref1 = new V1();  
v1 ref2 = new V1(5);
```



- Bei Instanzierung wird der Konstruktor aufgerufen, ggf. mit Argumenten



Zugriff auf Methoden und Attribute

- Es wird stets der .-Operator benutzt
- Zugriffsrechte werden geprüft!

```
V1 ref1 = new V1() ;  
V1 ref2 = new V1(5) ;
```

```
int ref1Attr = ref1.getAttribut() ;  
int ref2Attr = ref2.attribut ;  
System.out.println(ref1Attr+" "+ref2Attr) ;
```





Kap. 5.2

Zugriffsrechte

- Zugriffsrechte (für Attribute/Methoden/Klassen):
 - public: jeder kann benutzen/aufrufen
 - private: nur Methoden der Klasse können benutzen/aufrufen
 - package-public: nur Klassen im selben Paket können benutzen/aufrufen
 - protected: nur Methoden dieser und abgeleiter Klassen können benutzen/aufrufen





Beispiel für Deklaration, Instanzierung & Benutzung einer Klasse

- → V1.java (im E-Learning)



Kap. 5.1, 5.4

Instanzen

- Analogie: Klasse ist Stempel, Instanz ist Abdruck
 - gibt nur einen Stempel aber viele Abdrücke
 - Abdrücke können verschieden sein
(Attribute sind unterschiedlich)
 - Abdrücke sind unabhängig voneinander (da jeder eigenen Speicherbereich hat)





Kap. 5.1, 5.4

Instanzen

- Etwas formaler:
 - Von einer Klasse können beliebig viele Instanzen existieren
 - Änderungen in einer Instanz haben keinen Einfluss auf die anderen Instanzen
 - die Methoden einer Instanz können die Referenz **this** nutzen
 - Attribute/Methoden unterliegen **Zugriffsrechten**
 - spezielle Methoden: **Konstruktoren**





CUT: Q&A



ARBEIT

- Bitte bearbeiten Sie das „Vorlesungs-Quiz“ im E-Learning!





Wichtige Java-Elemente

- Konsoleneingabe/ausgabe
- Java-Arrays
- Java-Strings
- boolean statt bool
- Beispieldatei im E-Learning:
JavaArraysStrings.java



Konsolenein/ausgabe

- Eingabe: über Instanz der Klasse
`java.util.Scanner`
- Sehr leistungsfähig, kann von der Konsole
oder aus Daten/Puffern lesen
- Wir benutzen nur die Methode `nextLine()`



Konsolenein/ausgabe

- Ausgabe: u. A. über die Methoden
`System.out.print()`
`System.out.println()`
- Beide Methoden erwarten ein String-Argument



Kap. 5.5

Die Klasse String

- Instanzierung: `String s = "Hallo" ;`
- Wichtige Methoden: `charAt()`, `length()`, `find()`
- Verkettung mit „+“-Operator:

```
int i = 5 ;
```

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



Kap. 4

Java-Arrays

- Syntax ist etwas anders als in C:
 - 1D Arrays:

```
int [] intArray = new int [3] ;  
intArray[2] = 0 ;
```
 - 2D Arrays: Arrays von Arrays

```
int [] [] intArray2D = new int [3] [4] ;  
intArray2D[2] [1] = 9 ;
```
- Alle Arrays sind Instanzen und haben ein Attribut `length`



Kap. 2.3.8

Java-booleans

- In C existiert der Typ `bool` (eigentlich `unsigned char`) mit Werten 0 und 1
- In Java existiert der spezielle Typ `boolean` mit Werten `false` und `true`
- Alle Java-Bedingungen und Schleifen arbeiten mit `boolean`-Werten



Beispiele

- JavaArraysStringsBooleans.java im E-Learning



CUT: Q&A



Eclipse



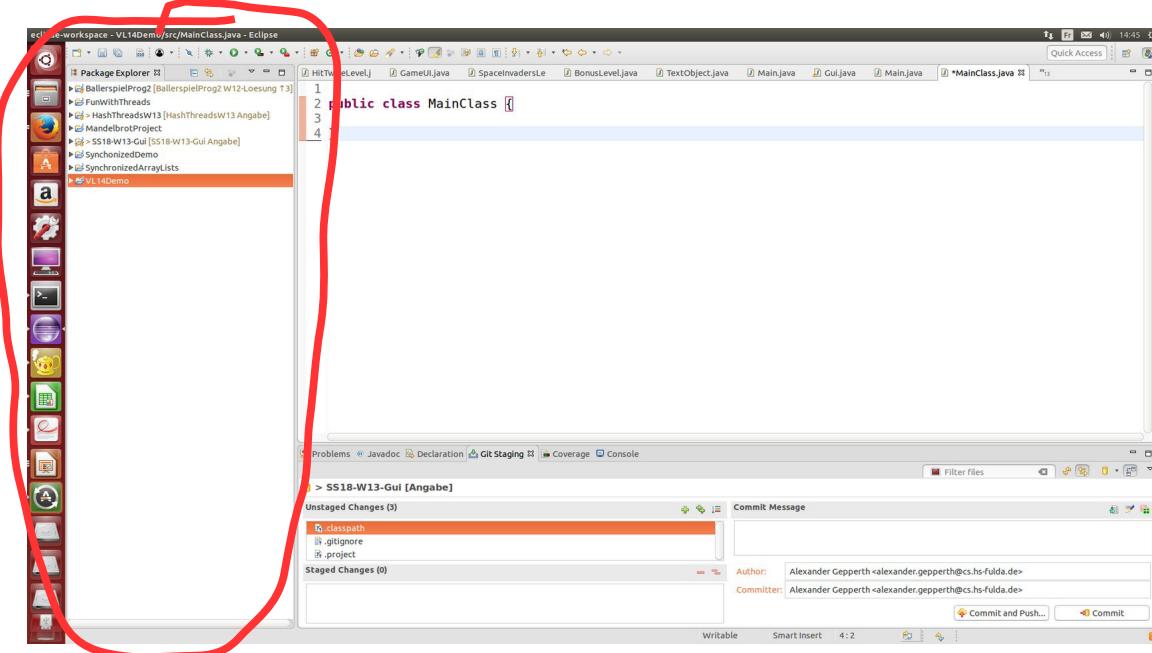
Eclipse

- Stellvertretend für alle IDEs
- Wird in dieser Vorlesung zur Demonstration und für Übungsaufgaben genutzt
- andere sind genauso gut: Visual Studio Code, NetBeans IDE, ...



Der Eclipse-Workspace

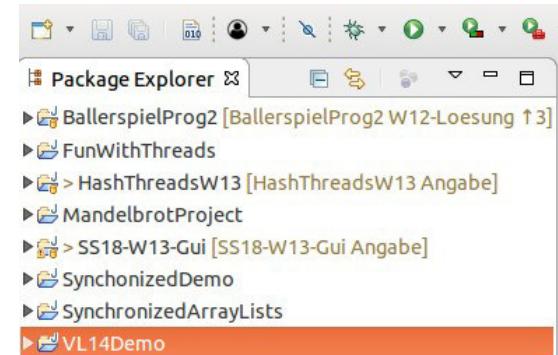
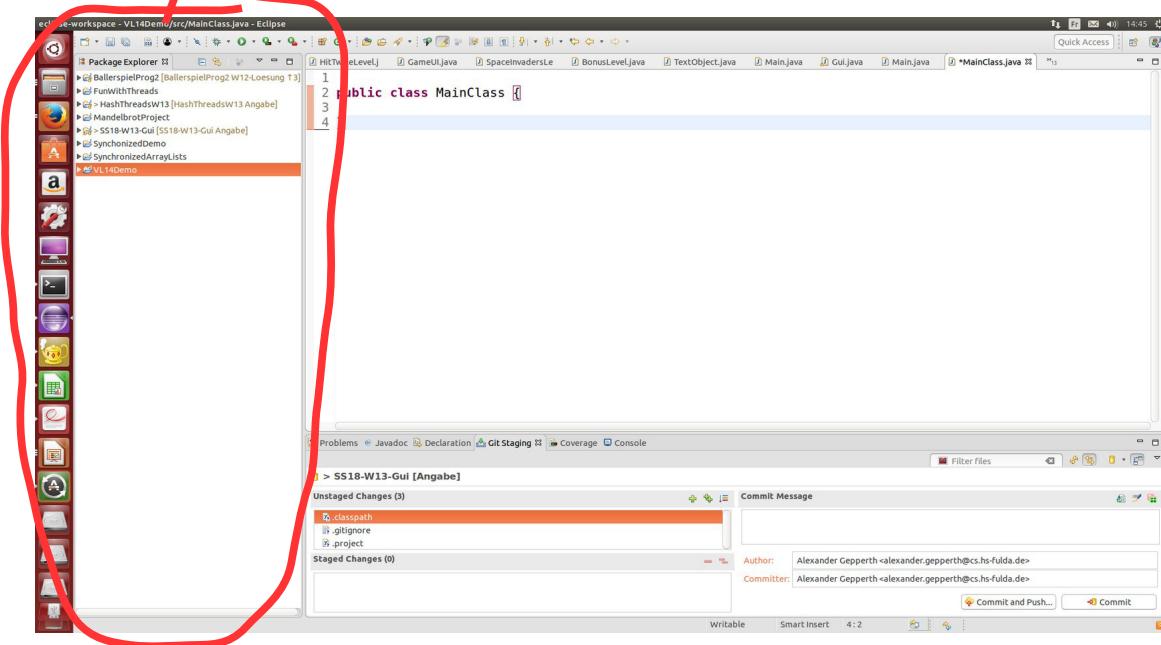
- Workspace = Sammlung aller eigenen Projekte





Der Eclipse-Workspace

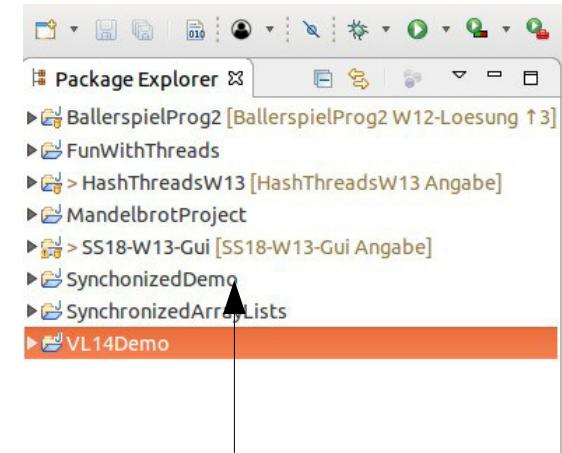
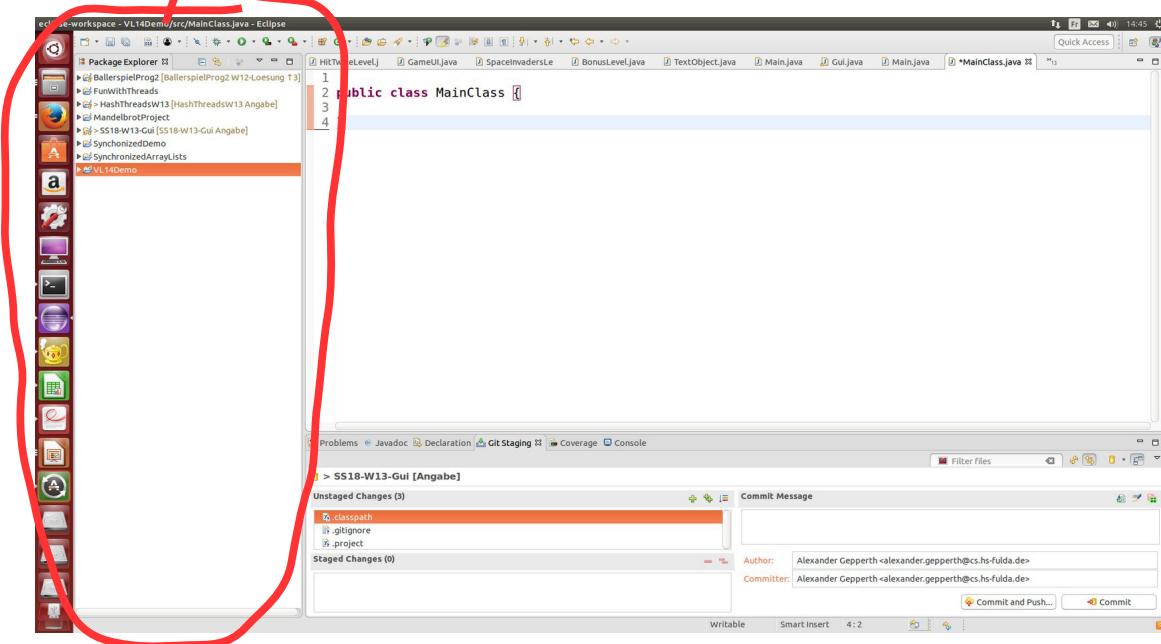
- Workspace = Sammlung aller eigenen **Projekte**





Der Eclipse-Workspace

- Workspace = Sammlung aller eigenen **Projekte**

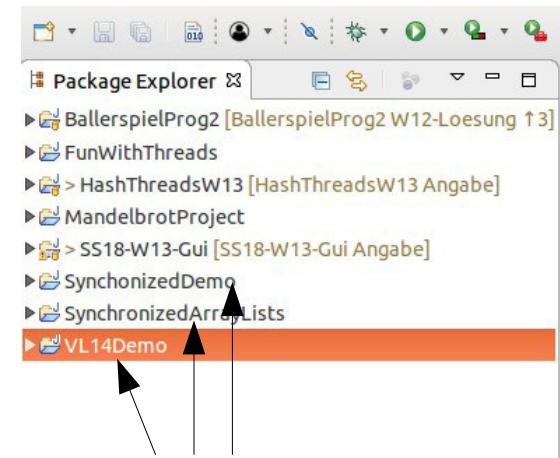
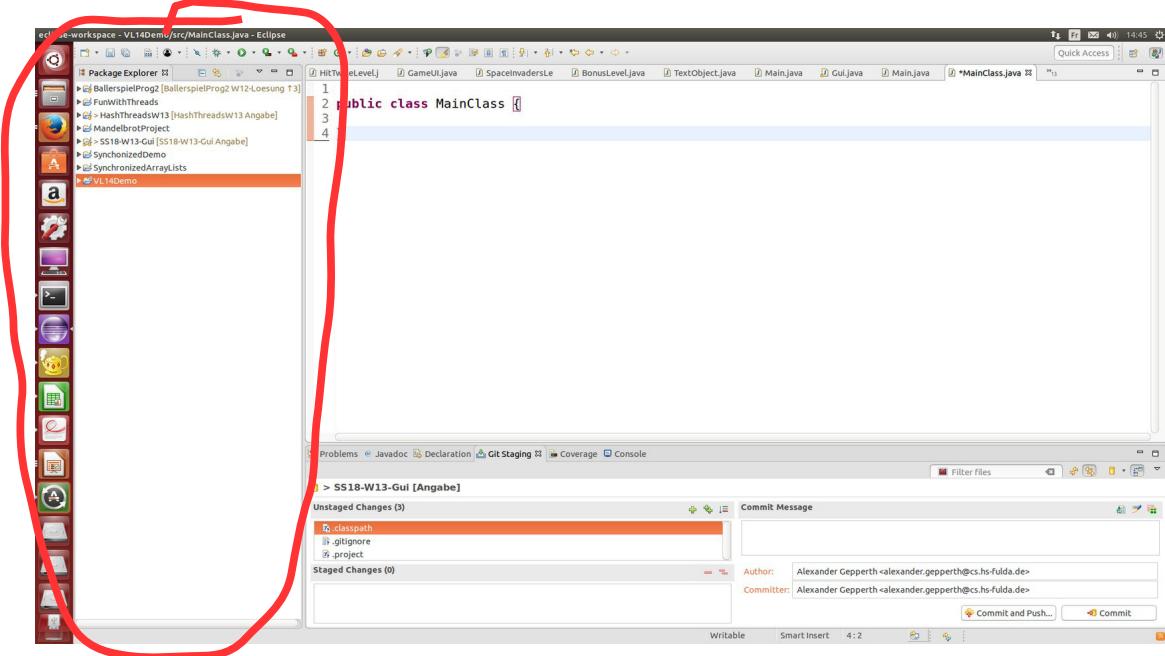


Projekte



Eclipse-Projekte

- Projekt ist meist ein ausführbares Programm (muss aber nicht)

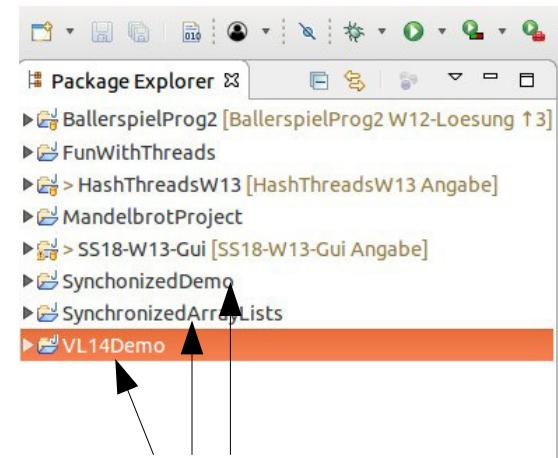


Projekte



Eclipse-Projekte

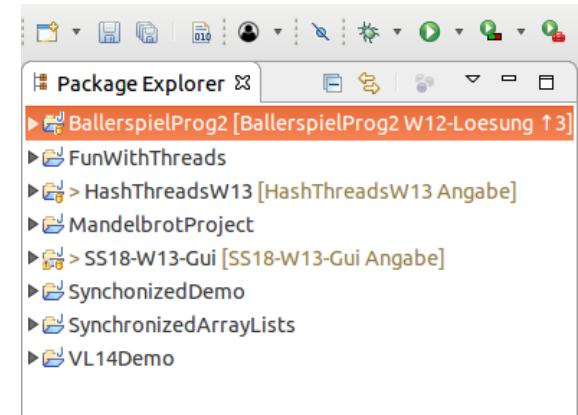
- **Projekt** ist meist ein ausführbares Programm (muss aber nicht)
- Sammlung von **Klassen** in einer oder mehreren **Packages**
- zusätzliche Datenfiles, alles was nötig ist um das Projekt erfolgreich auszuführen (laufenzulassen)
- Einstellungen zum Compilieren und Ausführen





Eclipse-Projekte

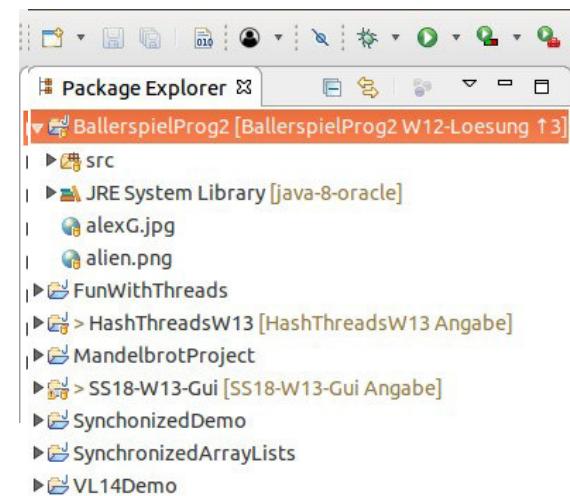
- **Projekt** ist meist ein ausführbares Programm (muss aber nicht)
- Sammlung von **Klassen** in einer oder mehreren **Packages**
- zusätzliche Datenfiles, alles was nötig ist um das Projekt erfolgreich auszuführen (laufenzulassen)
- Einstellungen zum Compilieren und Ausführen





Eclipse-Projekte

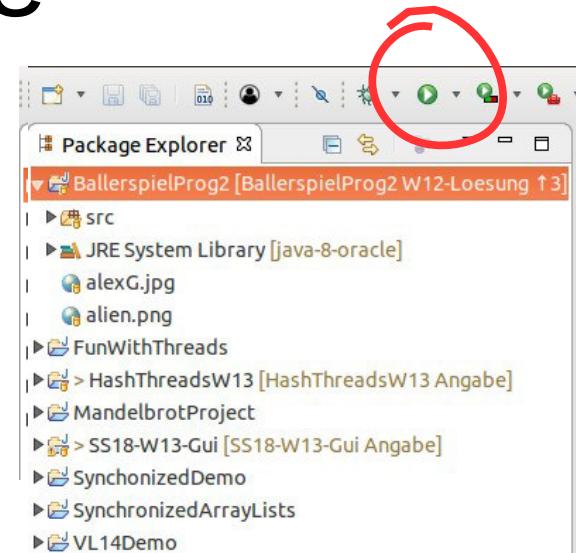
- Eclipse-Java-Projekte sind als Baumstruktur organisiert
- Ebenen/Elemente der Baumstruktur
 - Verzeichnisse
 - Packages
 - Java-Dateien
 - Klassen
 - » Methoden





Eclipse-Projekte

- Eclipse-Projekte die korrekt konfiguriert sind, kann man direkt ausführen!
- Konsolenausgabe kann ebenfalls in Eclipse beobachtet werden!





Eclipse-Projekte: Konsolenausgabe

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows several projects: BallerspielProg2 [BallerspielProg2 W12-Lösung], FunWithThreads, HashThreadsW13 [HashThreadsW13 Angabe], MandelbrotProject, SS18-W13-Gui [SS18-W13-Gui Angabe], SynchronizedDemo, SynchronizedArrayLists, and VL14Demo.
- Code Editor:** Displays Main.java code. A red oval highlights the static variable declaration and the while loop condition.

```
1 *
38*import java.lang.Runnable ;
41
42 // hier wird eine Klasse benötigt die Runnable implementiert!
43 class BackgroundCode implements Runnable {
44
45     @Override
46     public void run() {
47         while(true) {
48             if (Main.userInput.equals("Ping")) {
49                 System.out.println("Exiting!");
50                 System.exit(0);
51             }
52         }
53     }
54 }
55
56 }
57
58 public class Main {
59
60     static volatile String userInput = "";
61
62 }
```

- Console:** Shows the output of the application's execution. A red oval highlights the user input "Ping".

```
<terminated> Main [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Jul 3, 2018, 2:57:56 PM)
Enter text:
Test
Enter text:
Ping
Enter text:
Exiting!
```



Top Eclipse-Features

- Fehlererkennung:
rotes Kreuz am
Rand!!
- MouseOver auf
Kreuz gibt
Hilfestellung!
- aber: selbst Nachdenken ist sinnvoll!

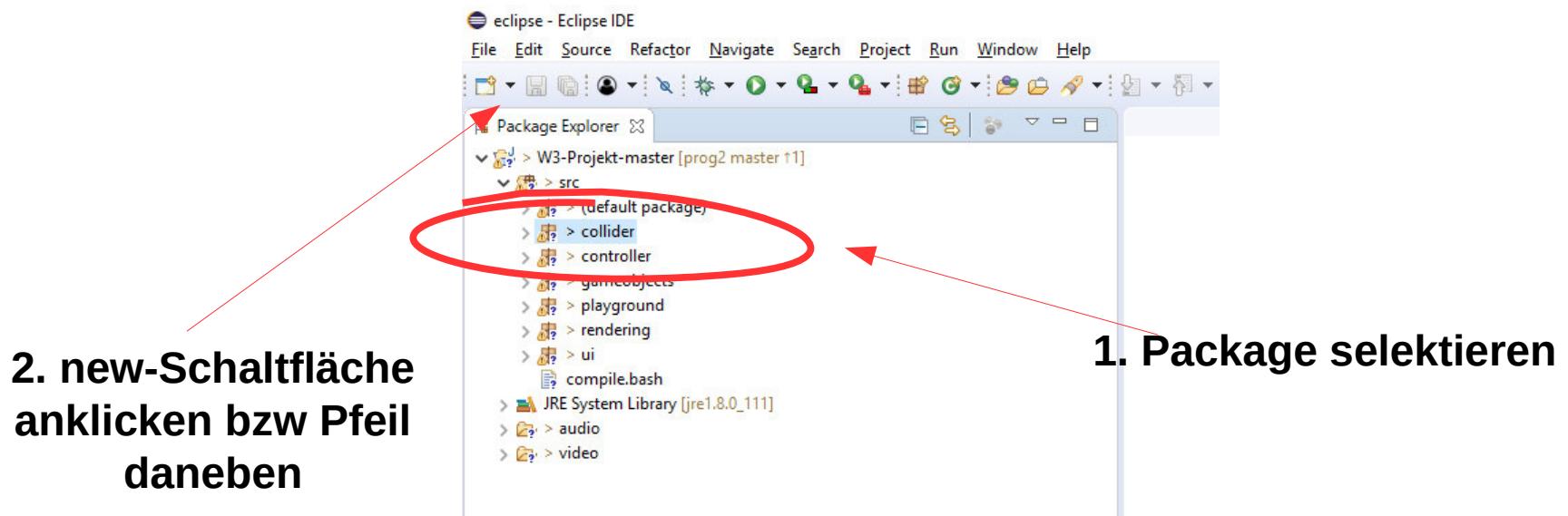
A screenshot of the Eclipse IDE interface. The top bar shows several Java files: HitTwiceLevel.java, GameUI.java, SpaceInvadersLe, BonusLevel.java, and Main.java. Below the bar is a code editor window containing the following Java code:

```
1
2 class TesteAufTeilbarkeit {
3
4     public static void main(String[] args) {
5         // TODO Auto-generated method stub
6
7         System.out.println("Alex" );
8
9     }
10
11 }
12 }
```

The code editor highlights line 7 with a red vertical bar and a red circle with a cross (error marker) at the start of the line. A tooltip or help message is visible over the error marker, indicating "System.out.println("Alex");".



Erstellen einer neuen Klasse im Projekt





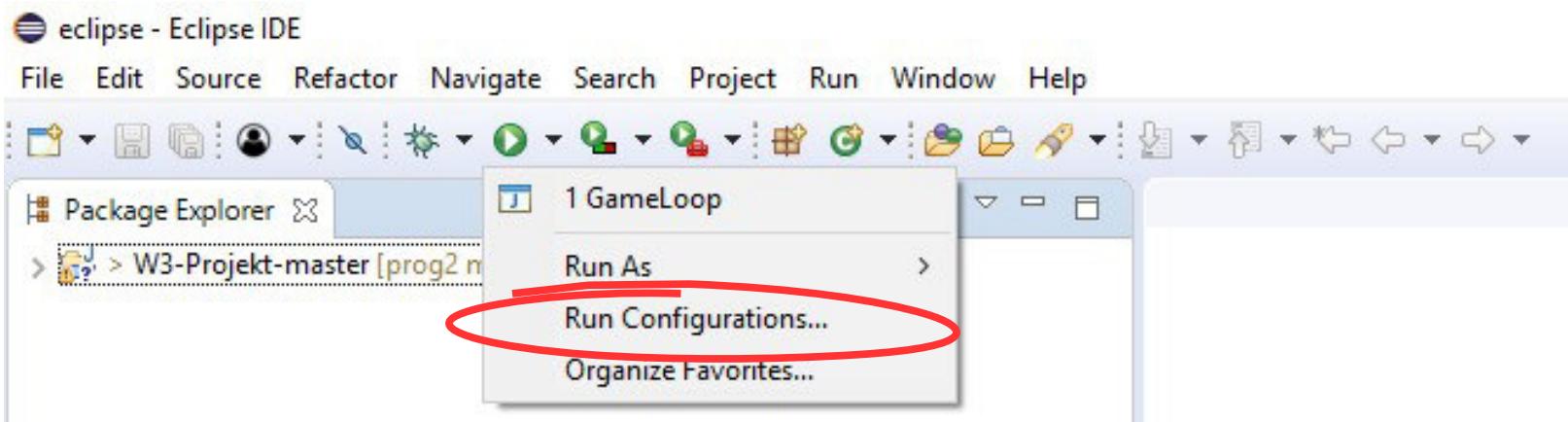
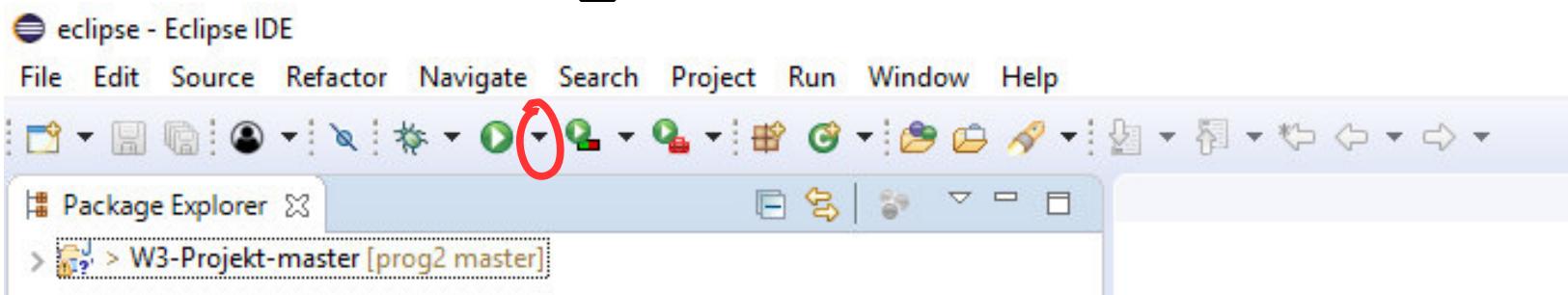
Erstellen einer neuen Klasse im Projekt

The screenshot shows the Eclipse IDE interface. On the left, the 'New' dialog is open, with the 'Class' option selected and highlighted by a red oval. On the right, the 'New Java Class' dialog is displayed, also with several fields highlighted by red ovals:

- Name:** The 'Name' field is highlighted with a red oval, with a red arrow pointing from the text 'Name wählen' (Select name) to it.
- Superclass:** The 'Superclass' dropdown is highlighted with a red oval, with a red arrow pointing from the text 'Oberklasse wählen' (Select superclass) to it.
- main() automatisch erzeugen lassen:** The 'Which method(s) would you like to create?' section contains three checkboxes: 'public static void main(String[] args)', 'Constructor(s)', and 'Inherited abstract methods'. The first checkbox is highlighted with a red oval, with a red arrow pointing from the text 'main() automatisch erzeugen lassen' (Allow automatic generation of main()) to it.

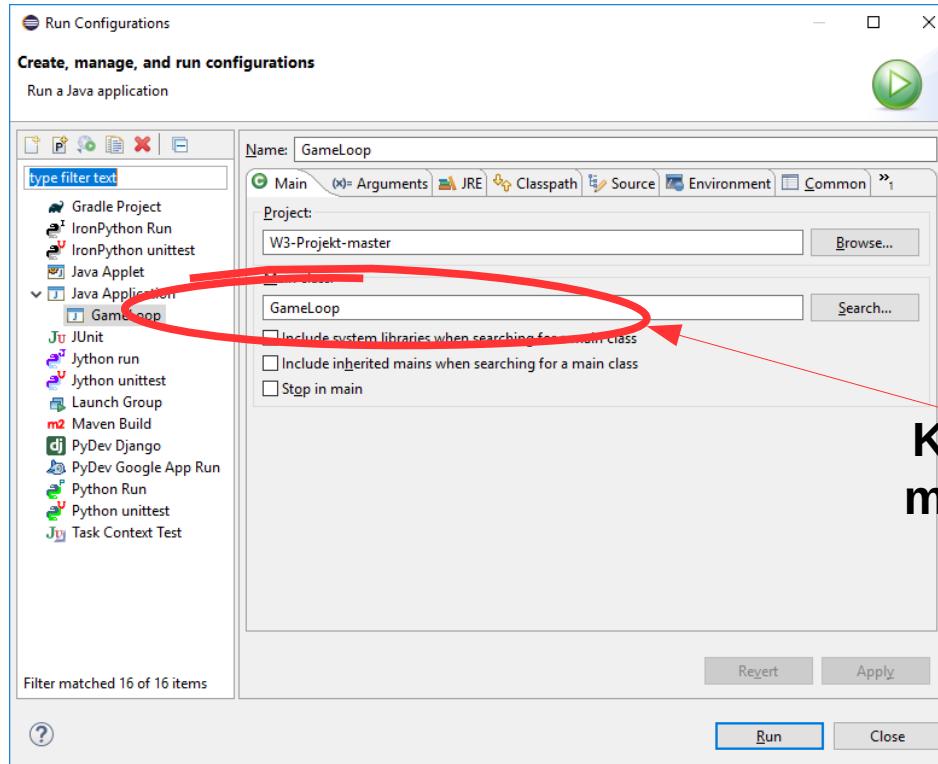


Selektieren welche Klasse ausführt wird



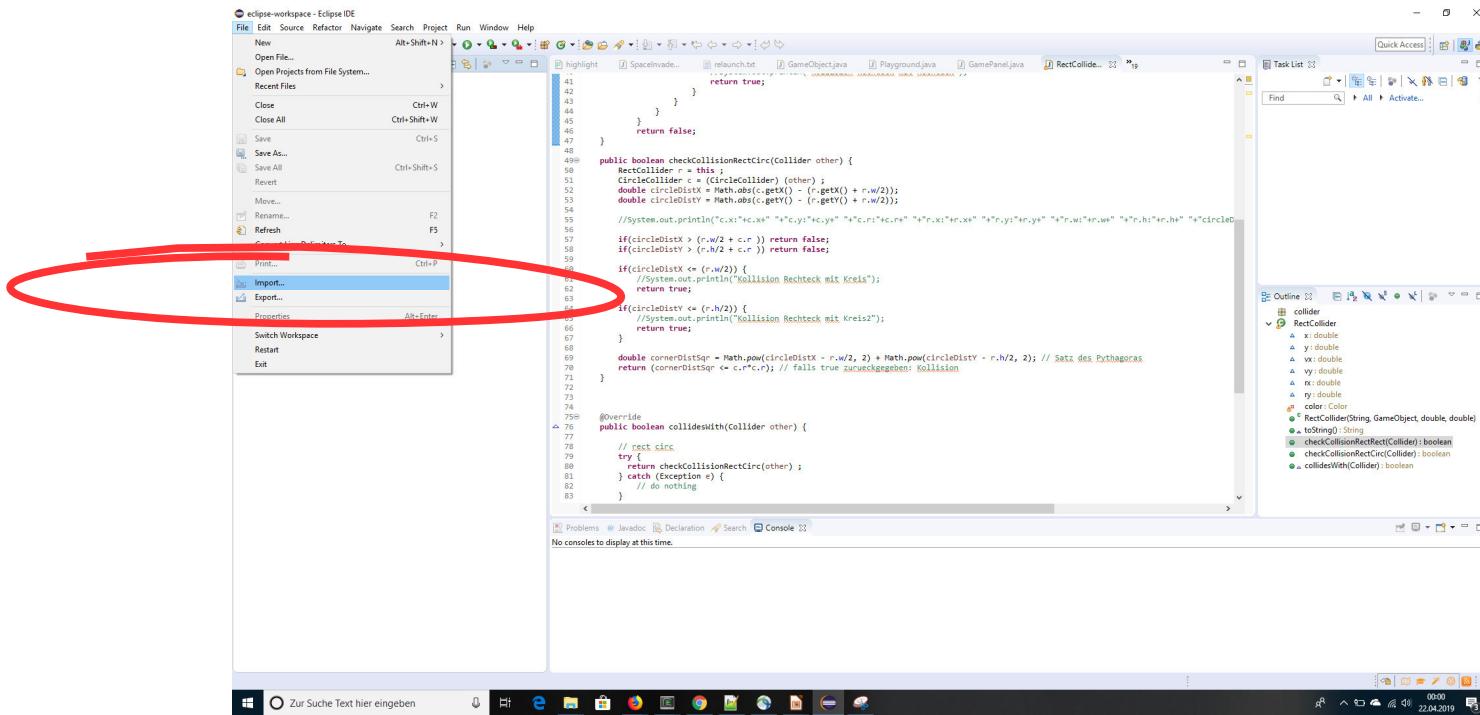


Selektieren welche Klasse ausgeführt wird



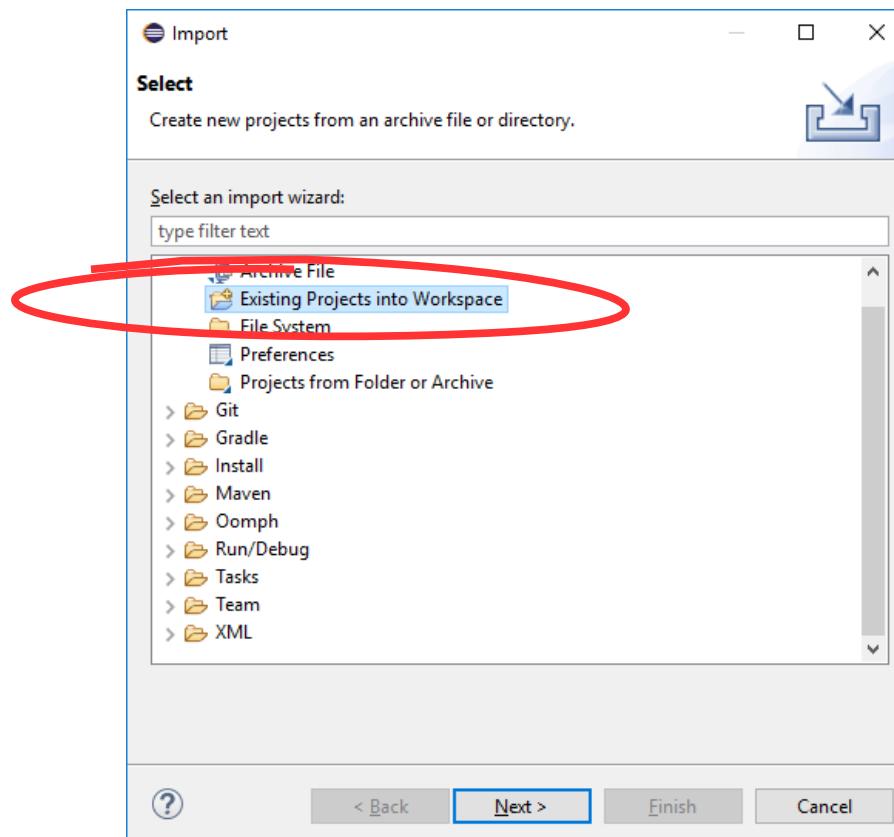


Import von gezippten Projekten



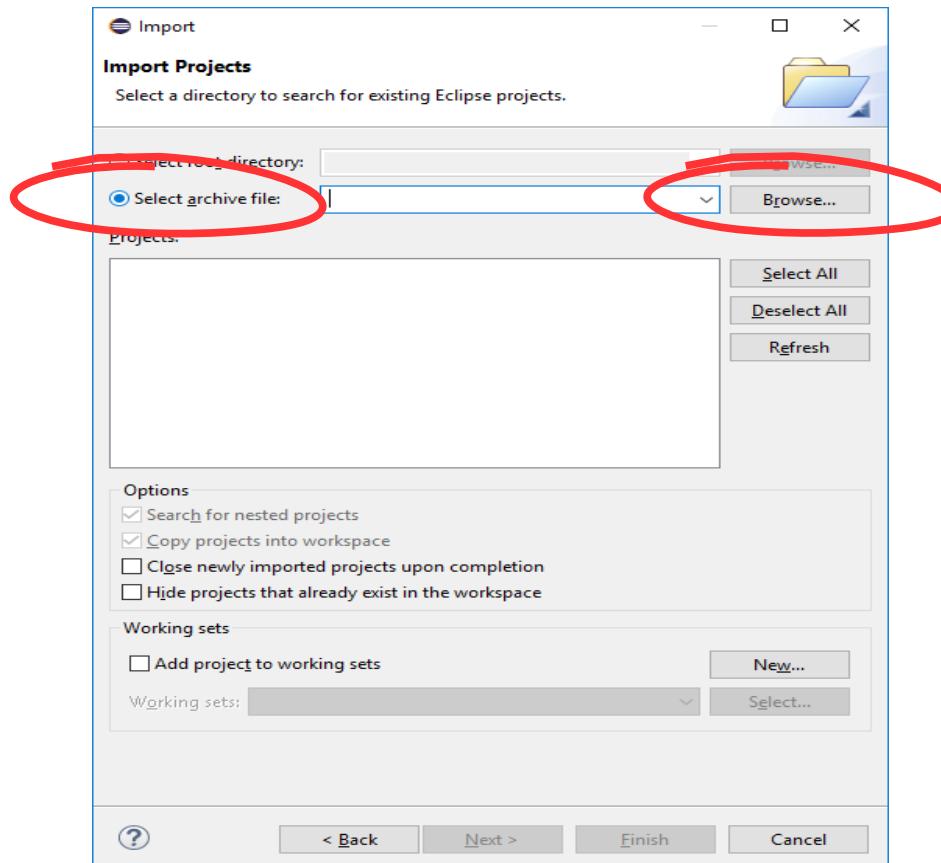


Import von gezippten Projekten





Import von gezippten Projekten



ZIP-Datei selektieren



Fortgeschrittene IDE-Funktionen

- Eclipse kann sehr weit reichende Operationen auf Ihrem Quellcode ausführen
 - Formatieren (z.B. Google Java Style)
 - Klassen umbenennen
 - Importe organisieren
 - try/catch einfügen
 - Schleife einfügen
 - Aus/Einkommentieren
 - ...



Fortgeschrittene IDE-Funktionen

- Eclipse erlaubt es Ihnen zu debuggen!
 - das bedeutet: ein Programm Schritt für Schritt auszuführen
 - und nach jedem Schritt alle Variablen/Attribute anzusehen
 - damit kann man komplexe Fehler sehr effizient finden!

Demo



CUT: Q&A



Programmierung 2

Vorlesung 2: JVM, Projekte, Packages

April 2022

alexander.gepperth@cs.hs-fulda.de



Java-Grundlagen



Hochsprachen

- Hochsprachen beschreiben Algorithmen unabhängig vom Prozessor/Computer
- Nicht immer komplett möglich (Datentypen, Big/Little-Endianness, Dateisysteme, ...)
- Hochsprachen: C, C++, C#, JavaScript, Java, Python, Lisp, Prolog, ...
 - lesbar, portabel
 - ggf. langsam



Kap. 1.2

Maschinensprache

- Alle modernen Computer haben eine eigene “Programmiersprache”, die so genannte **Maschinensprache**
- Maschinensprache wird vom Prozessor verstanden
- Maschinensprache fkt. nur mit best. Prozessor
 - **schnell**
 - **wenig lesbar**
 - **kaum portabel**

```

org 100h
start:
    mov dx, meldung1
    mov ah, 9h
    int 021h
    mov ah, 01h      ;Wert über die Tastatur einlesen
    int 021h
    cmp al, '5'
    ja 11
    mov dx, meldung2
    mov ah, 9h
    int 021h
    jmp ende
11:   mov dx, meldung3
    mov ah, 9h
    int 021h
ende:
    mov ah, 4Ch
    int 21h
section .data
meldung1: db 'Bitte Zahl eingeben:', 13, 10, '$'
meldung2: db 13, 10, '<= 5!', 13, 10, '$'
meldung3: db 13, 10, '> 5!', 13, 10, '$'
  
```



Kap. 1.2

Was ist ein Compiler?

- Compiler übersetzen Hochsprachen-Code in Maschinensprache f. einen best. Computer/Prozessor
 - schneller Code
 - Code ist lesbar und portabel

```
import java.util.Scanner ;
Scanner sc = new Scanner (System.in);
String c = sc.nextLine();
if (c.charAt(0) > '5')
    System.out.println("> 5");
else
    System.out.println("<= 5");
```



```
org 100h
start:
    mov dx,meldung1
    mov ah,9h
    int 021h
    mov ah, 01h ;Wert über die Tastatur einlesen
    int 021h
    cmp al, '5'
    ja 11
    mov dx,meldung2
    mov ah,9h
    int 021h
    jmp ende
11: mov dx,meldung3
    mov ah,9h
    int 021h
ende:
    mov ah,4ch
    int 21h
section .data
meldung1: db 'Bitte Zahl eingeben:', 13, 10, '$'
meldung2: db 13, 10, '<= 5!', 13, 10, '$'
meldung3: db 13, 10, '> 5!', 13, 10, '$'
```



Kap. 1.2

Die Java Virtual Machine (JVM)

- Java ist eine Compilersprache
- Java-Programme werden stets für denselben Computer/Prozessor kompiliert: die JVM
- **Java Virtual Machine (JVM)**: Modell eines (simulierten) Computers inklusive Prozessor
- Kompilierte Java-Programme laufen auf jedem Rechner, der eine JVM implementiert!



Kap. 1.2

Die Java Virtual Machine (JVM)

- Grund für Benutzung der JVM:
 - Compilierung von Hochsprachen-Programmen auf verschiedenen Rechnern oft problematisch
 - Einfacher, eine JVM zu implementieren als einen Compiler
 - Garantierte Portabilität falls JVM korrekt implementiert ist (kann geprüft werden)



CUT: Q&A

- Vorlesungs-Quiz 1 machen!



Von der Klasse zum Projekt



Von der Klasse zum Projekt

- Manche Probleme können mit einer Klasse gelöst werden
- manche nicht, erfordern Zusammenspiel vieler Klassen
- Java definiert Regeln für solche **Projekte**



Regeln für Java-Projekte

- Es darf nur eine public-Klasse (& beliebig viele nicht-public-Klassen) pro .java-File geben
- Es darf nur eine public-Klasse mit main()-Methode geben
- jede Klasse wird in genau ein .class-File compiliert
- .class-Files werden per import eingebunden

```
import java.util.Scanner ;
```



Gliederung von Projekten



Gliederung von Projekten

- Große Projekte enthalten viele Klassen
- Klassen können thematisch in **Packages** gegliedert werden
- Packages: bekannt?



Packages

- Schon gesehen:

```
import java.util.Scanner ;
```



Packages

- Was macht die Anweisung
“import java.util.Scanner ;” ?
- Macht die Klasse Scanner aus dem
Package java.util verfügbar!



Packages

- Package: Sammlung von Klassen, oft thematisch organisiert
- Bekannte Packages des JRE:
 - `java.lang` (Standard)
 - `java.util`
 - `java.io`
 - `java.awt` und `javax.swing`
- Wir sehen: Packages können Packages enthalten!



Packages nutzen

voll qualifizierter
Klassenname

```
import java.util.Scanner ;
```

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält



Packages nutzen

```
import java.util.Scanner ;
```

Package

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält



Packages nutzen

```
import java.util.Scanner ;
```

Unterpackage

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält



Packages nutzen

```
import java.util.Scanner ;
```

einfacher Klassenname

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält



Der import-Mechanismus

```
import java.util.Scanner ;
```

- Wozu dient der voll qualifizierte Klassename?
 - import muss Position der .class-Datei im Dateisystem kennen, um die Klasse zu nutzen
 - Packages oder Unterpackages: Verzeichnisse im Dateisystem
 - voll qualifizierter Name: bestimmt die Position der .class-Datei im Dateisystem



Der import-Mechanismus

```
import java.util.Scanner ;
```

- Bei diesem `import`-Befehl wird folgendes gemacht:
 - Auswertung von **CLASSPATH**: Liste mit Suchpfaden
 - `import` versucht, in jedem <suchpfad> die Datei <suchpfad>/controller/ObjectController.class zu finden
 - Es wird geprüft, ob das .class-File auch eingebunden werden darf (public/package-public)



Zugriffsrechte für Klassen: public, package-public

- 2 Möglichkeiten für Top-Level Klassen
 - public: jede andere Klasse kann sie importieren

```
public class TestClass { }
```
 - package-public: nur Klassen aus dem selben Package können sie importieren

```
class TestClass { }
```
 - private/protected: später



Eigene Packages definieren

- Schlüsselwort package als erstes Kommando einer Datei + voll qualifizierter Package-Name

```
package controller ;
```

- alle Klassen in Datei gehören zu diesem Package
- Compiler weiss wohin er die .class-Datei schreiben soll (**WOHIN in diesem Fall??**)



CUT: Q&A

- Vorlesungs-Quiz 2 machen!

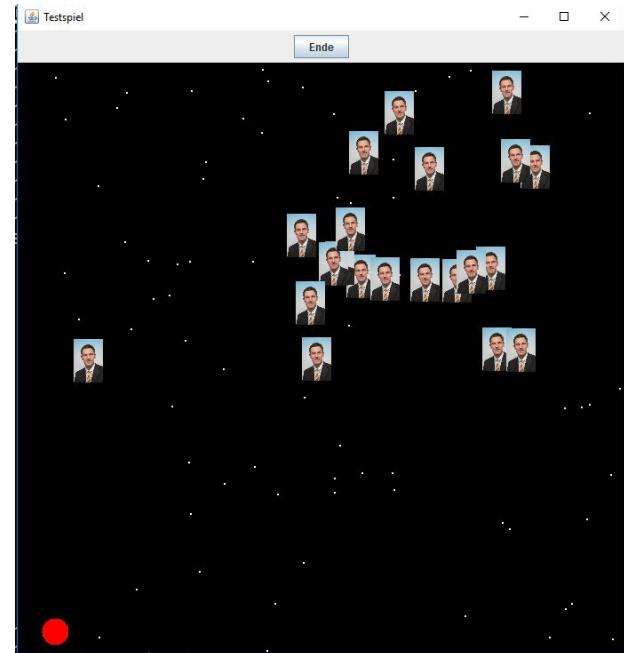


Projekt: Prog2Shooter



Projekt: Prog2Shooter

- Ziele:
 - ist nicht besonders schwer etwas interessantes zu programmieren
 - roter Faden durch die Vorlesung: Verdeutlichung von Programmierkonzepten immer anhand dieses Projekts
 - Gegenstand der meisten freien Übungen
 - Motivation selbst zu programmieren und/oder zu gestalten
 - KEIN Ersatz für oder Konkurrenz zu professionellen Game-Engines





Das Prog2Shooter-Projekt

- Enthält ca. 50 Klassen, ~10.000 Zeilen Code
- konzipiert als Bibliothek zur **einfachen** und **flexiblen** Erstellung von 2D-Spielen
- d.h. auch ganz andere Spiele können anhand der vorhandenen Klassen gebaut werden
- **Wiederverwendbarkeit:** andere Programmierer können auf diesem Projekt aufbauen



Das Prog2Shooter-Projekt

- Arbeitsteilung zwischen Hauptklassen des Projekts
 - GameLoop implementiert die "Game Loop"
 - GameUI implementiert die GUI
 - Playground beschreibt ein Level
 - GameObject beschreibt ein einzelnes Objekt
 - Controller definiert das Verhalten eines Objekts
 - Artist zeichnet ein Objekt
 - Collider stellt Kollisionen von Objekten fest



Das Prog2Shooter-Projekt

- **Klassen** des Projekts organisiert in **Packages**:
 - default alle ausführbaren Hauptklassen
 - playground alle Level-Klassen
 - gameobject alle Objekt-Klassen
 - controller alle Controller-Klassen
 - rendering Klassen zum Zeichnen spezieller Objekte
 - collider alle Typen von Collidern
 - gameui alles was mit Menüführung zu tun hat



Das Prog2Shooter-Projekt

- Ziel: maximale **Wiederverwendbarkeit** und Kombinationsfähigkeit
- Umsetzung durch Prinzipien der **Objektorientierung**: Vererbung, Aggregation, Delegation, abstrakte Klassen, Polymorphie, Typsubstitution, ...



Das Prog2Shooter-Projekt

- Verhalten (inklusive Steuerung), Kollisionserkennung und Darstellung frei kombinierbar
- Verhalten, Kollisionserkennung, Darstellung leicht erweiterbar/modifizierbar
- gleiche Objekte in verschiedenen Levels einsetzbar, andere Interaktionen
- einzelne Aspekte von Levels leicht erweiterbar/modifizierbar



Das Prog2Shooter-Projekt

- Mögliche Weiterentwicklungen (Bachelor-Arbeiten!!):
 - Z-Buffer
 - Panning für Jump&Run-Spiele
 - Effiziente Kollisionserkennung
 - Sound
 - Level-Editoren/ Einlesen offener Formate
 - Skripting (JPython, ...)



Inspektion in Eclipse



DEMO!!

- Pacman
- Breakout
- SpaceInvaders



Programmierung 2

Vorlesung 3: Hinter den Kulissen I

Alexander Gepperth, April 2022



Variablen in Java



Kap. 1.2

Ein Wort zu Variablen

- eine Variable ist für den Compiler ein Name für einen **Speicherbereich**
 - Startadresse
 - Länge
- Deklaration der Variable **reserviert** den Speicherbereich (auf Stack, Heap oder Method Area)
- Speicherbereich: Größe hängt von Datentyp ab

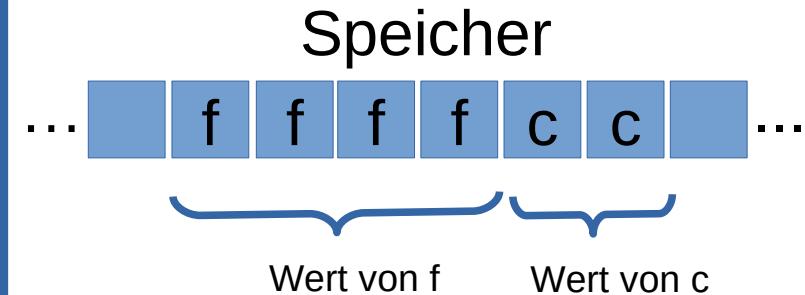


Kap. 2.3

Ein Wort zu Variablen

- Größe von Variablen im Speicher

```
float f ; // 4 Byte
int i ; // 4 Byte
char c ; // 2 Byte
String s ; // 4 Byte
int [] k ; // 4 Byte
```



- warum 4 Byte für String und int [] ?
→ 2 Arten von Variablen!



Primitive Datentypen



Kap. 1.2

Typen von Variablen: primitive Datentypen

- Name für Speicherbereich (Start, Länge)
- In diesem Bereich ist ein **Wert** gespeichert
- Beispiele: `char`, `int`, `float`, `double`, ...

```
char c ;
```

Code

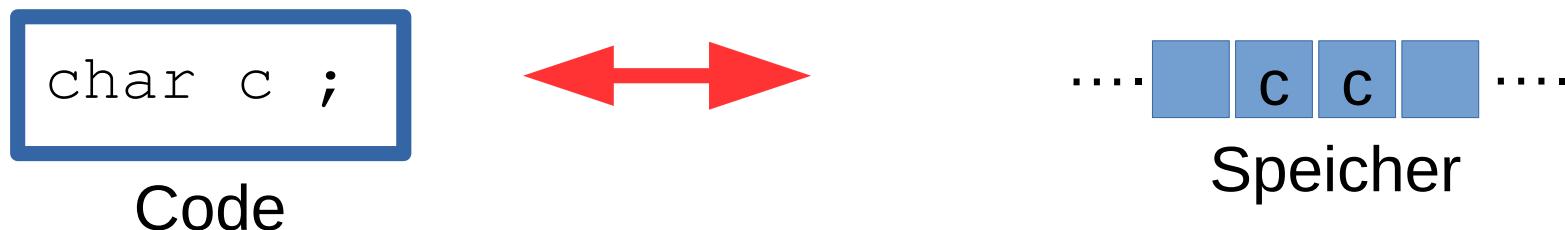
- keine primitiven Typen: `int []`, `float []`, `double []`



Kap. 1.2

Typen von Variablen: primitive Datentypen

- Name für Speicherbereich (Start, Länge)
- In diesem Bereich ist ein **Wert** gespeichert
- Beispiele: char, int, float, double, ...



- keine primitiven Typen: int[], float[], double[]



Kap. 1.2

Benutzung von primitiven Datentypen

- bei primitiven Datentypen genügt es, sie zu deklarieren um sie zu benutzen
 - Speicherbereich wird durch Deklaration reserviert
 - kann jetzt gelesen und beschrieben werden

```
char c ;  
c = '1' ;  
if (c == '1') ...
```

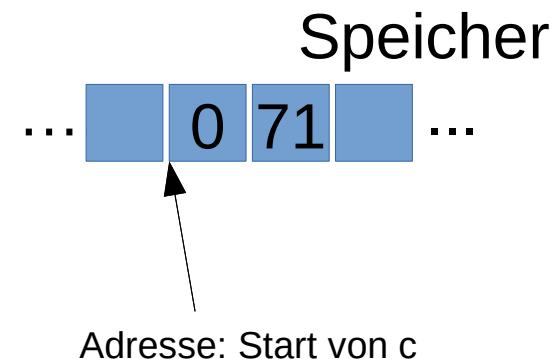


Kap. 1.2

Typen von Variablen: primitive Datentypen

- was passiert wenn ein primitiver Datentyp beschrieben wird?

```
char c = '1' ;  
c = '5' ;
```



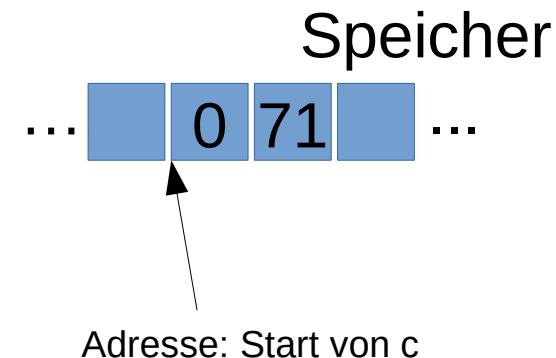


Kap. 1.2

Typen von Variablen: primitive Datentypen

- was passiert wenn ein primitiver Datentyp beschrieben wird?
 - Adresse wird nachgeschlagen

```
char c = '1' ;  
c = '5' ;
```



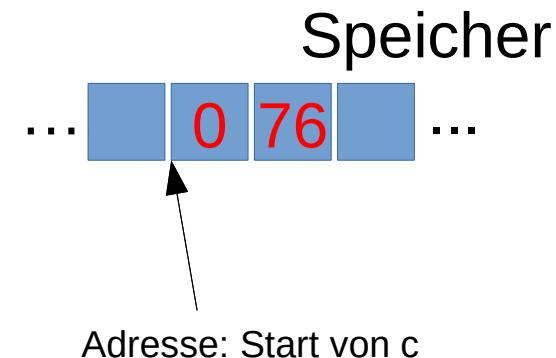


Kap. 1.2

Typen von Variablen: primitive Datentypen

- was passiert wenn ein primitiver Datentyp beschrieben wird?
 - Adresse wird nachgeschlagen
 - Speicherbereich wird geändert

```
char c = '1' ;  
c = '5' ;
```





Referenzvariablen



Kap. 3.4

Typen von Variablen: Referenzvariablen

- Ebenfalls: Name für Speicherbereich (Start, Länge)
- dort ist **eine Referenz** gespeichert, **kein Wert!**
- **Referenz**: Speicheradresse des Werts (woanders im Speicher)

```
char[] c ;
```

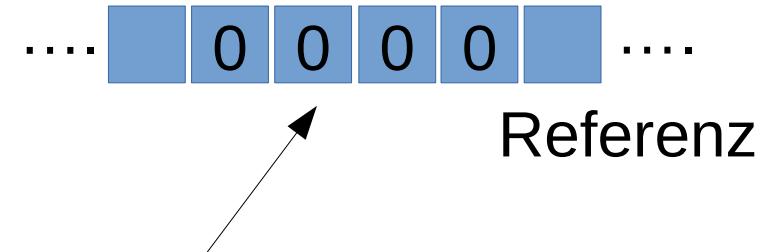


Kap. 3.4

Typen von Variablen: Referenzvariablen

- Ebenfalls: Name für Speicherbereich (Start, Länge)
- dort ist **eine Referenz** gespeichert, **kein Wert!**
- **Referenz**: Speicheradresse des Werts (woanders im Speicher)

```
char[] c ;
```



Ref. werden auf null
initialisiert!



Kap. 3.4

Typen von Variablen: Referenzvariablen

- Ebenfalls: Name für Speicherbereich (Start, Länge)
- dort ist **eine Referenz** gespeichert, **kein Wert!**
- **Referenz:** Speicheradresse des Werts (woanders im Speicher)

```
char[] c = new char[5];
```





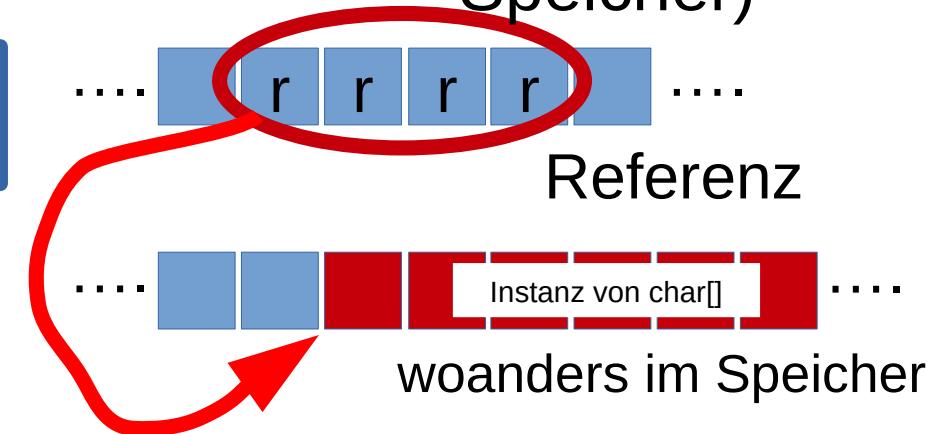
Kap. 3.4

Typen von Variablen: Referenzvariablen

- Ebenfalls: Name für Speicherbereich (Start, Länge)
- dort ist **eine Referenz** gespeichert, **kein Wert!**
- **Referenz**: Speicheradresse des Werts (woanders im Speicher)

```
char[] c = new char[5];
```

**Referenzen sind
Pfeile („wo kann ich
den Wert finden?“)**





Kap. 3.4

Benutzung von Referenzvariablen

- Deklaration der Referenzvariable reserviert Speicher..
- ... aber nur für die Referenz (Anfangswert null) ...

```
char[] c ;  
c[3] = '0' ;
```



zeigt "nirgendwo"
hin

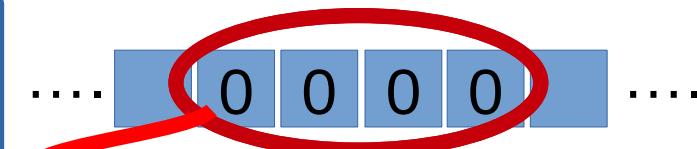


Kap. 3.4

Benutzung von Referenzvariablen

- Deklaration der Referenzvariable reserviert Speicher..
- ... aber nur für die Referenz (Anfangswert null) ...
- ... nicht für den Wert auf den die Referenz zeigen soll!

```
char[] c ;  
c[3] = '0' ;
```



Funktioniert nicht:
NullPointerException !

zeigt "nirgendwo"
hin

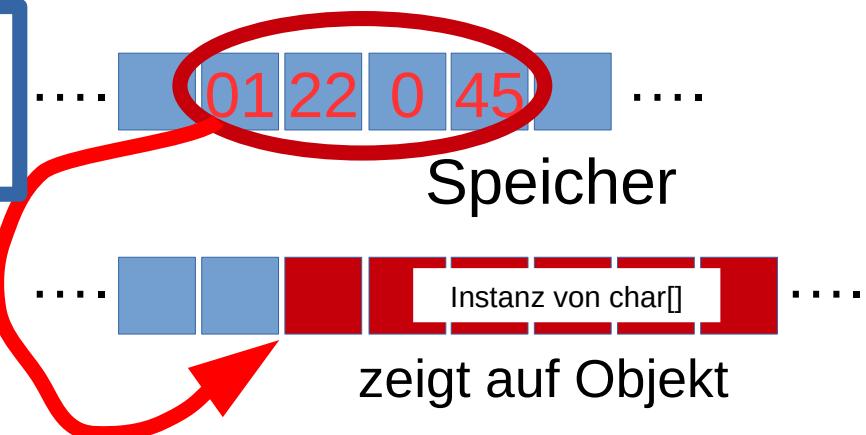


Kap. 3.4

Benutzung von Referenzvariablen

- Deklaration der Referenzvariable reserviert Speicher..
- .. aber nur für die Referenz selbst ..
- .. nicht für das Objekt auf das die Referenz zeigen soll!

```
char[] c = new char [5];  
c[3] = '0';
```



**new reserviert Speicher
für das char[]-Objekt
und gibt die Adresse
zurück!**



Kap. 3.4

Benutzung von Referenzvariablen

- Was passiert bei folgendem Code?

```
char [] c = new char [5];  
c[1] = '0';
```



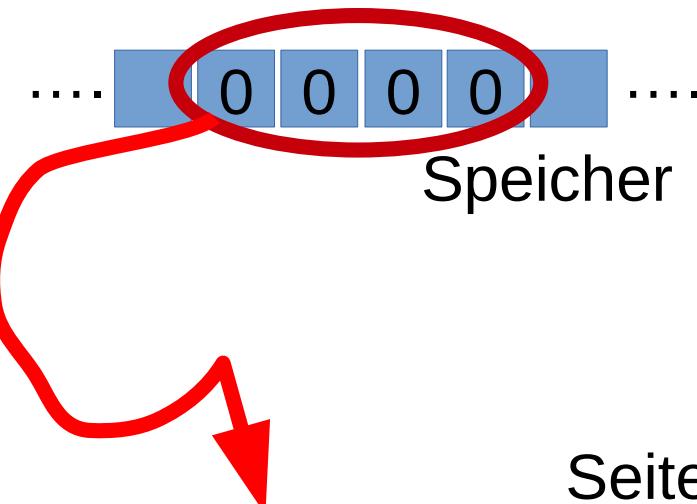
Kap. 3.4

Benutzung von Referenzvariablen

- Was passiert bei folgendem Code?

```
char[] c = new char [5];  
c[1] = '0';
```

- Speicher für Referenz reservieren,
zeigt nirgendwo hin





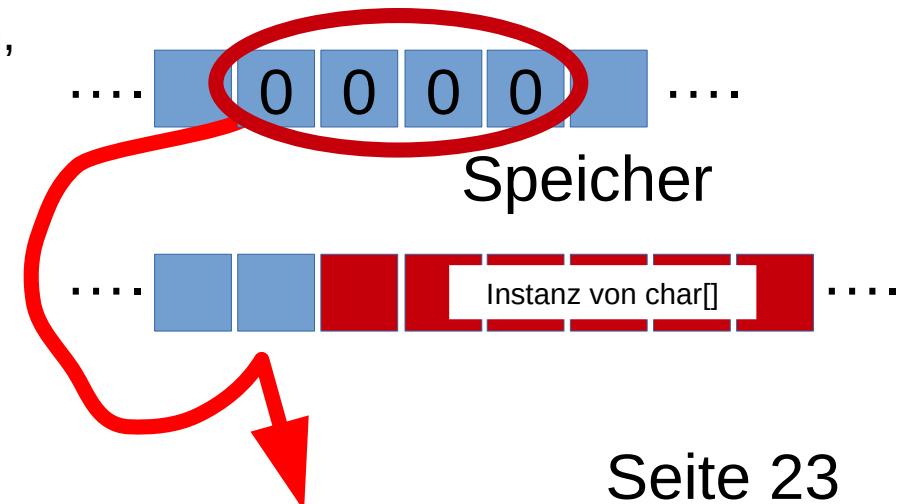
Kap. 3.4

Benutzung von Referenzvariablen

- Was passiert bei folgendem Code?

```
char [] c = new char [5];
c[1] = '0';
```

- Speicher für Referenz reservieren, zeigt nirgendwo hin
- Speicher für Objekt reservieren





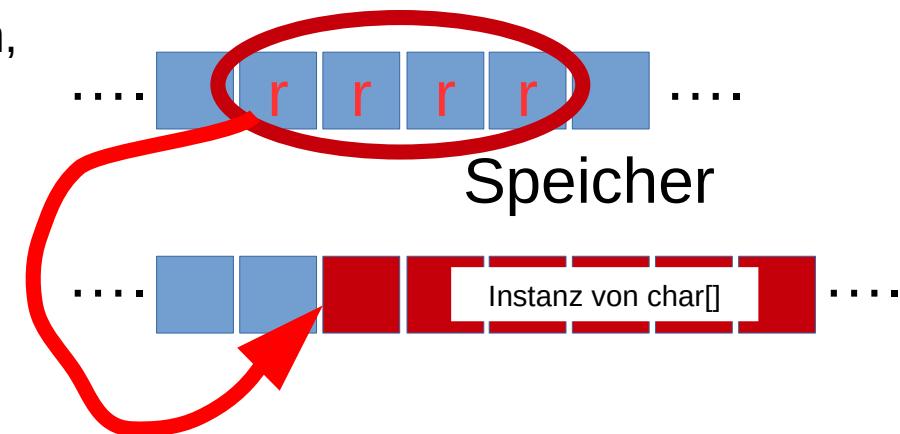
Kap. 3.4

Benutzung von Referenzvariablen

- Was passiert bei folgendem Code?

```
char[] c = new char [5];  
c[1] = '0';
```

- Speicher für Referenz reservieren, zeigt nirgendwo hin
- Speicher für Objekt reservieren
- Pfeil/Referenz "umbiegen"





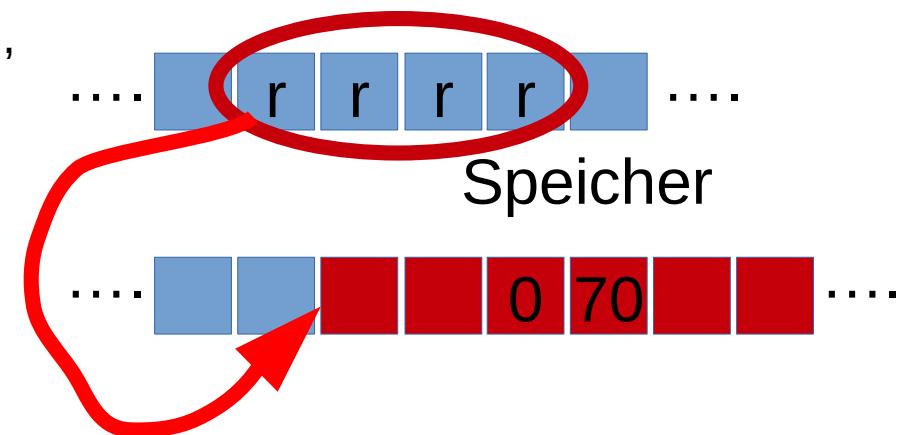
Kap. 3.4

Benutzung von Referenzvariablen

- Was passiert bei folgendem Code?

```
char[] c = new char[5];  
c[1] = '0';
```

- Speicher für Referenz reservieren, zeigt nirgendwo hin
- Speicher für Objekt reservieren
- Pfeil/Referenz "umbiegen"
- Wert schreiben





Kap. 3.4

Referenzvariablen sind Pfeile

- Wichtig: Unterscheidung zwischen Referenzvariable und dem Objekt auf das sie zeigt
- Zuweisungen an Referenzvariable **verbiegen den Pfeil!**

```
char[] v1 = new char[5];
v1[0] = 'C';
char[] v2;
v2 = v1;
System.out.println(v2[0]);
```

?



Kap. 3.4

Referenzvariablen sind Pfeile

- Wichtig: Unterscheidung zwischen Referenzvariable und dem Objekt auf das sie zeigt
- Zuweisungen an Referenzvariable **verbiegen den Pfeil!**

```
char[] v1 = new char[5];
v1[0] = 'C';
char[] v2;
v2 = v1;
System.out.println(v2[0]);
```

Ausgabe:
C



Kap. 3.4



Referenzvariablen sind Pfeile

- Wichtig: Unterscheidung zwischen Referenzvariable und dem Objekt auf das sie zeigt
- Zuweisungen an Referenzvariable **verbiegen den Pfeil!**

```
char[] v1 = new char[5];
v1[0] = 'C';
char[] v2;
v2 = v1;
v2[0] = 'X';
System.out.println(v1[0]);
```

?



Kap. 3.4

Referenzvariablen sind Pfeile

- Wichtig: Unterscheidung zwischen Referenzvariable und dem Objekt auf das sie zeigt
- Zuweisungen an Referenzvariable **verbiegen den Pfeil!**
- Über den verbogenen Pfeil alle Operationen möglich!

```
char [ ] v1 = new char [5];  
v1[0] = 'c' ;  
char [ ] v2 ;  
v2 = v1 ;  
v2[0] = 'X' ;  
System.out.println(v1[0]);
```

Ausgabe:
X



Kap. 3.4

Referenzvariablen sind Pfeile

- Wichtig: Unterscheidung zwischen Referenzvariable und dem Objekt auf das sie zeigt
- Zuweisungen an Referenzvariable **verbiegen den Pfeil!**

```
int [] v1 = new int [5];
v1[0] = 1000 ;
int [] v2 = {1,2,3,4} ;
v1 = v2 ;
System.out.println(v1[0]);
```

?



Kap. 3.4

Referenzvariablen sind Pfeile

- Wichtig: Unterscheidung zwischen Referenzvariable und dem Objekt auf das sie zeigt
- Zuweisungen an Referenzvariable **verbiegen den Pfeil!**

```
int [] v1 = new int [5];
v1[0] = 1000 ;
int [] v2 = {1,2,3,4} ;
v1 = v2 ;
System.out.println(v1[0]);
```

Ausgabe:
1

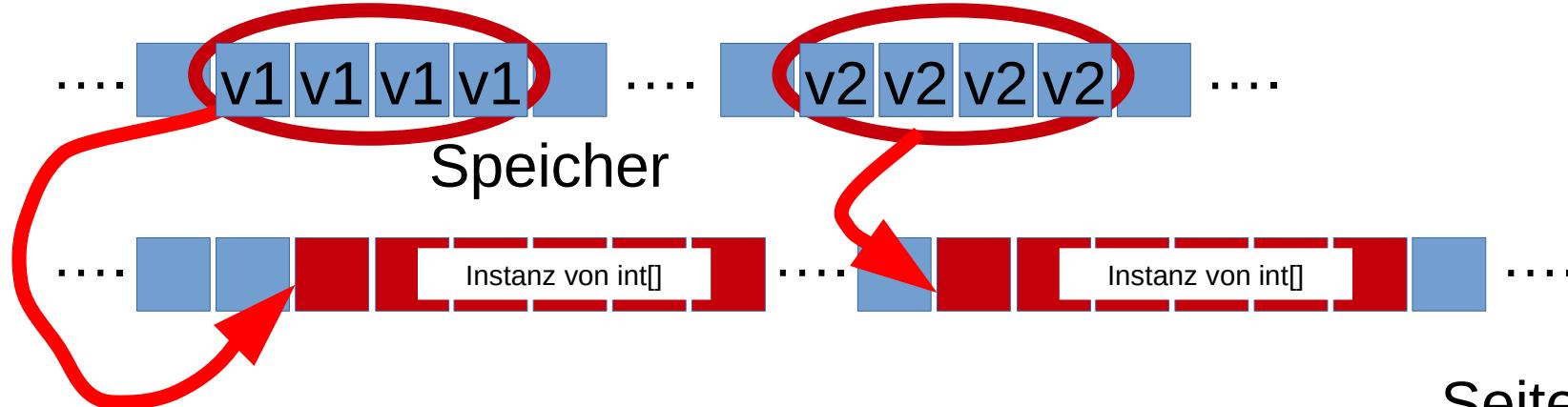


Kap. 3.4

Referenzvariablen sind Pfeile

- Grafische Darstellung
(vor `v1 = v2`)

```
int [] v1 = new int [5];
v1[0] = 1000 ;
int [] v2 = {1,2,3,4} ;
v1 = v2 ;
System.out.println(v1[0]);
```



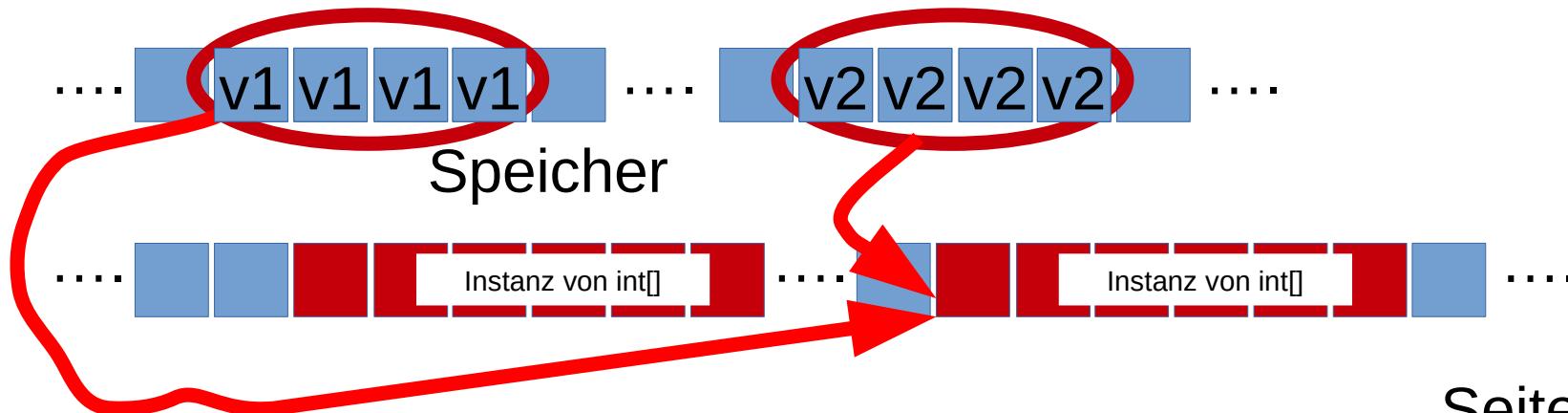


Kap. 3.4

Referenzvariablen sind Pfeile

- Grafische Darstellung
(nach `v1 = v2`)
- klar dass Operationen auf A auch B ändern!

```
int [] v1 = new int [5];
v1[0] = 1000 ;
int [] v2 = {1,2,3,4} ;
v1 = v2 ;
System.out.println(v1[0]);
```



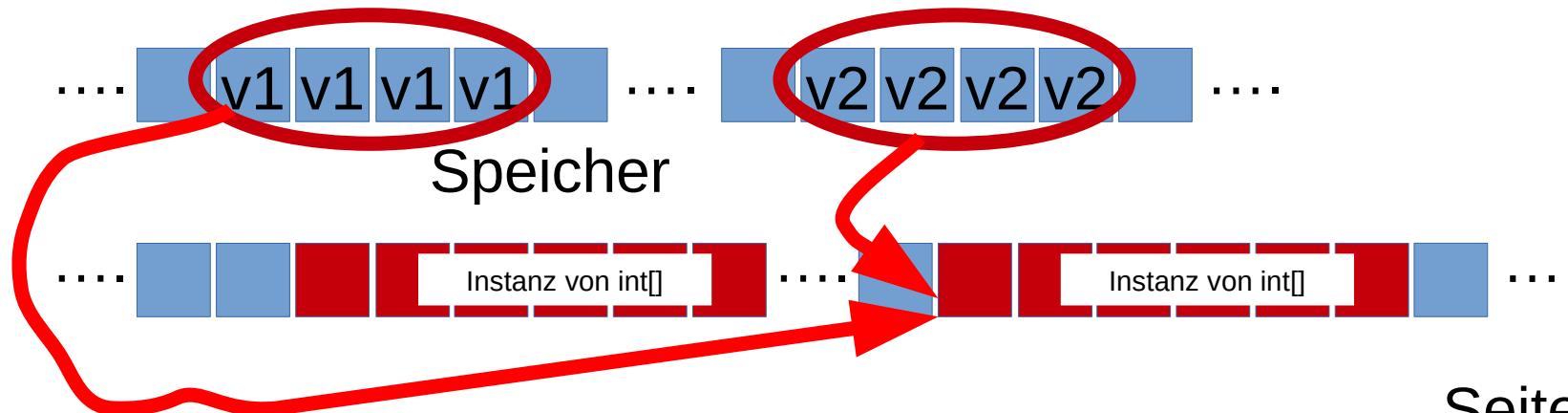


Kap. 3.4

Garbage Collection

- Durch das Verbiegen von Referenzen können Objekte “verlorengehen”
- Objekte ohne Referenz werden **automatisch** gelöscht: **Garbage Collection**

```
int[] v1 = new int [5];
v1[0] = 1000 ;
int[] v2 = {1,2,3,4} ;
v1 = v2 ;
System.out.println(v1[0]);
```





Zusammenfassung

- Jede Variable ist entweder Referenz oder primitiv
- In beiden Fällen sind Variablen Namen für Speicherbereiche
 - primitiver Datentyp: Speicherbereich repräsentiert Wert
 - Referenzdatentyp: Speicherbereich repräsentiert Adresse eines Werts(“Pfeil”)



Zusammenfassung

- Referenzvariablen sind:
 - Arrays primitiver Typen: `char[]`, `float[]`, `int[]`, ...
 - Instanzen von Klassen:
 - Vordefinierte Klassen: `java.util.Scanner`, `String`, ...
 - selbst definierte Klassen: `Playground`, `GameObject`, `GameLoop`, ...
- Primitive Datentypen sind: `int`, `float`, `char`, `double`, `byte`, `long`, ...



Kap. 3.4

Zusammenfassung

- Primitive Datentypen sind einfach zu handhaben
- Referenzdatentypen sind mächtig, erfordern allerdings Verständnis vor allem bei
 - Zuweisungen (bereits behandelt)
 - Testen auf Gleichheit
 - Parameterübergabe an Methoden



CUT: Q&A



Programmierung 2

Vorlesung 4: Vererbung

Alexander Gepperth, Mai 2022

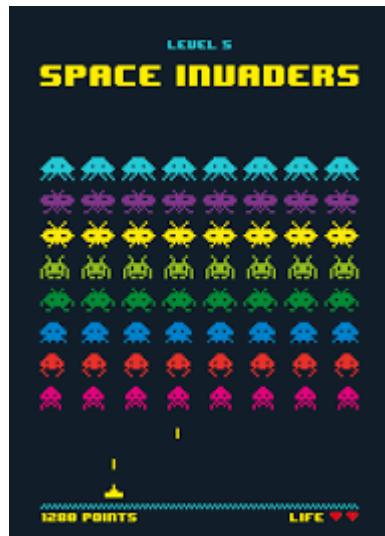


Plan heute

- Prog2Shooter: die Klassen GameLoop und SpaceInvadersLevel
- Objektorientierung und Vererbung



Das Prog2Shooter-Projekt





Die Klasse GameLoop

- zeichnet den aktuellen Stand des Spiels
 - Definiert eine Frame-Rate $1/\Delta$
 - erhöht Spielzeit um Δ
- Update des Spielzustands
- Reagiert auf GUI-Aktionen (Buttons, Menü)
- Definiert die Level des Spiels



Die Klasse GameLoop

- Wichtige Methoden:
 - void runGame(String [] args)
führt das komplette Spiel aus! Der Parameter args kommt aus main()
 - public static void main(String [] args)
muss eine Instanz einer Unterklasse von GameLoop erstellen und runGame ausführen
 - Playground nextLevel(Playground currentLevel) : definiert Level



Die Klasse GameLoop

- Level-Definition erfolgt durch:

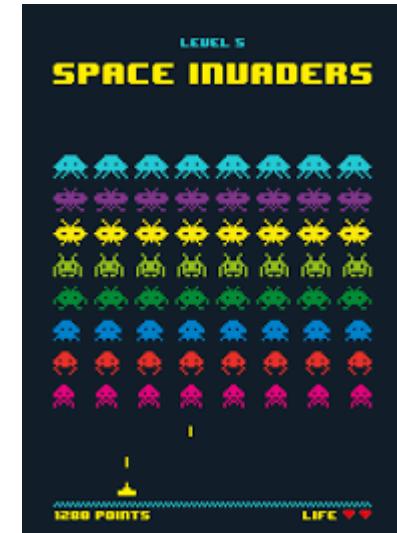
Playground nextLevel(Playground currentLevel)

- von runGame aufgerufen, erzeugt eine Instanz von Playground, welche das nächste Level beschreibt
- falls Argument currentLevel gleich null
→ erstes Level
- falls null zurückgegeben wird → Spielende



Die Klasse SpaceInvadersLevel

- Einzelne Level werden durch Unterklassen von `Playground` beschrieben
- `SpaceInvadersLevel` definiert ein Level vom Typ des alten “Space Invaders”-Spiels
 - Aliens starten oben und sinken langsam nach unten
 - Aliens schießen gelegentlich
 - Aliens müssen abgeschossen werden bevor sie unten ankommen

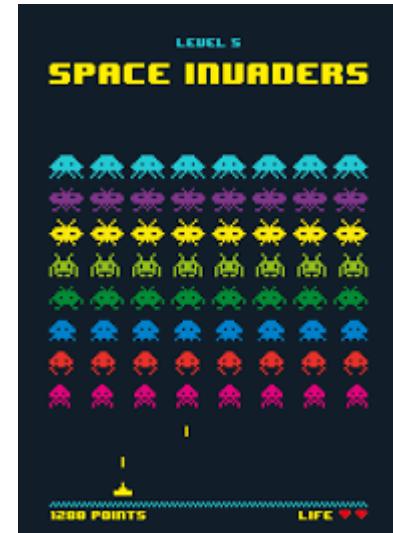




Die Klasse SpaceInvadersLevel

- Wichtige Methoden:

- `void prepareLevel(String id)`
wird 1x zu Level-Beginn von `runGame` aufgerufen.
Der Parameter `id` kann später über die Methode
`getName()` abgefragt werden.
– wird normalerweise nicht überschrieben--
- `void applyGameLogic()`
wird bei jedem update-Schritt aufgerufen. Prüft Kollisionen und
führt entsprechende Aktionen aus.
– wird normalerweise nicht überschrieben –

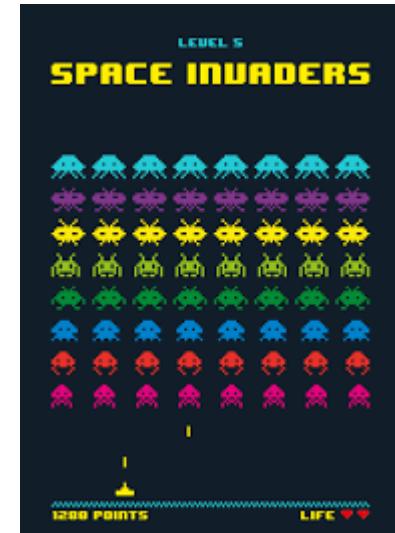




Die Klasse SpaceInvadersLevel

- Diese Methoden definieren wichtige Eigenschaften eines Levels:

- protected String getStartupMessage()
- double calcEnemyShotProb()
- protected double calcEnemySpeedX()
- protected int calcNrEnemies()
- protected int calcNrCollect()
- protected void createEnemies()
- protected void createCollectables()
- protected void createEgoObject()
- void actionIfEnemyIsHit(GameObject e, GameObject shot)
- void actionIfEgoCollidesWithEnemy(GameObject enemy, GameObject ego)
- void actionIfEgoCollidesWithCollect(GameObject collect, GameObject ego)





Vererbung: Teaser

- Problem: die Klasse GameLoop besitzt eine Methode nextLevel()
 - diese Methode lädt alle Level des Spiels
 - dadurch wird definiert welche Levels überhaupt gespielt werden können
- **Wenn man an dieser Methode etwas ändern will...**
 - **C-Ansatz: copy&paste von GameLoop**
 - **Java-Ansatz: Vererbung**



CUT: Q&A



Vererbung: Grundlagen



Kap. 5.8 (bis 5.14)

OOP: Vererbung für Dummies

- Java-Klassen können ihre Eigenschaften an andere **vererben!**

```
class Oberklasse {  
    int x = 3 ;  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
class Unterklasse extends Oberklasse {  
    public static void main(String[] a) {  
        Unterklasse u = new Unterklasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```





Kap. 5.8 (bis 5.14)

OOP: Vererbung für Dummies

- Java-Klassen können ihre Eigenschaften an andere **vererben!**

```
class Oberklasse {  
    int x = 3 ;  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
class Unterklasse extends Oberklasse {  
    public static void main(String[] a) {  
        Unterklasse u = new Unterklasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```

3

14



Kap. 5.8 (bis 5.14)

OOP: Vererbung für Dummies

- Terminologie:
 - **Oberklasse**: vererbt ihre Eigenschaften, steht nach `extends`
 - **UnterkLASSE**: erbt Eigenschaften, steht vor `extends`
- Synonyme:
 - Oberklasse: *superclass*, *parent class*
 - UnterkLASSE: *abgeleitete Klasse*, *derived class*, *child class*

```
class Oberklasse {  
    int x = 0 ;  
    public int getX() {  
        return this.x ;  
    }  
}  
  
class UnterkLASSE extends Oberklasse {  
    public static void main(String[] a){  
        UnterkLASSE u = new UnterkLASSE() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```



Kap. 5.8 (bis 5.14)

OOP: Bedeutung von Vererbung

- Unterkasse besitzt **alle** Attribute/Methoden von Oberkasse
- Ausnahme: Konstruktoren werden nicht vererbt (später)
- Unterkasse kann **zusätzliche** Attribute/Methoden definieren
- Unterkasse kann existierende Methoden durch eigene **ersetzen** (Synonyme: *überschreiben*, *override*)

```
class Oberklasse {  
    int x = 0 ;  
    public int getX() {  
        return this.x ;  
    }  
}  
  
class Unterkasse extends Oberklasse {  
    public static void main(String[] a){  
        Unterkasse u = new Unterkasse() ;  
        System.out.println( u.getX() ) ;  
    }  
}
```



Kap. 5.8 (bis 5.14)

OOP: Bedeutung von Vererbung

- UnterkLASSE hat alles was OberKLASSE auch hat
- plus: evtl. zusätzliche Attribute und Methoden
- aber: überschriebene Methoden können natürlich andere Dinge tun
- **UnterkLASSE kann als Spezialisierung der OberKLASSE angesehen werden**
- (später: UnterkLASSE kann stets anstelle von OberKLASSE benutzt werden)



Vererbung: Überschreiben von Methoden

- Überschreiben/Ersetzen einer Methode in einer UnterkLASSE
 - Name der Methode, Rückgabetyp und Parameter müssen **exakt** übereinstimmen
 - **Annotation** @Override erlaubt dies zur Compile-Zeit zu prüfen
 - ursprüngliche Methode in der UnterkLASSE weiter verfügbar durch super.methodename(. . .)



Beispiel aus dem Prog2Shooter-Projekt

- Selektives Ersetzen einzelner Methoden, um einzelne Aspekte eines Levels oder Spiels anzupassen

```
class PacmanGame extends GameLoop {  
  
    @Override  
    public Playground nextLevel  
        (Playground currentLevel) {  
        ..  
        ..  
    }  
}
```



Beispiel aus dem Prog2Shooter-Projekt

- Klasse GameLoop definiert nextLevel **damit** die Methode später überschrieben werden kann
- Ebenso: Klasse SpaceInvadersLevel
- Designer von GameLoop hat das Überschreiben im Design der Klasse vorgesehen
- → OOP: wiederverwenbare Software
- → **OOP: Designer definiert nicht nur aktuelle sondern auch spätere Benutzung seines Codes**



Cut: Q&A



Die Klasse Object



Die Klasse Object

- Alle Klassen in Java erben von Object
 - falls keine Oberklasse → automatisch Object



Kap. 5.9

Die Klasse Object

- Alle Klassen in Java erben von Object
 - falls keine Oberklasse → automatisch Object

```
class KeineOberklasse {  
  
    public void methode1() {  
    }  
}
```



Die Klasse Object

- Alle Klassen in Java erben von Object
 - falls keine Oberklasse → automatisch Object

```
class KeineOberklasse extends Object {  
  
    public void methode1() {  
    }  
}
```

äquivalent!!



Die Klasse Object

- Alle Klassen in Java erben von Object
- Folglich: jede Klasse besitzt alle Methoden von Object, z.B.
 - equals()
 - toString()
 - clone()
 - hashCode()
 - ...



Kap. 5.9

Das Substitutionsprinzip

- Eine UnterkLASSE ist Spezialfall ihrer OberKLasse: kann **formal** alles was die OberKLasse auch kann (und evtl. mehr)



Kap. 5.9

Das Substitutionsprinzip

- Eine Unterkelasste ist Spezialfall ihrer Oberklasse: kann **formal** alles was die Oberklasse auch kann (und evtl. mehr)
 - da, wo Referenz auf Oberklasse erwartet wird, kann auch Referenz auf Unterklasste stehen
 - allerdings: über die Referenz auf Oberklasse dürfen nur Methoden aus Oberklasse aufgerufen werden (**WARUM?**)
 - insbesondere: da wo Referenz auf Object erwartet wird, kann jede Klasse stehen (aber: nur Methoden aus Object verfügbar)



Kap. 5.9

Das Substitutionsprinzip

```
class Oberklasse {  
    public void printSomething() {  
        System.out.println("Oberklasse");  
    }  
}  
  
class Unterklasse extends Oberklasse {  
  
    @Override  
    public void printSomething() {  
        System.out.println("Unterkelas");  
    }  
  
    void printSomethingElse() {  
        System.out.println("ZUSÄTZLICH");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
  
        Oberklasse ref = new Unterklasse() ;  
        ref.printSomething() ;  
        ref.printSomethingElse() ;  
    }  
}
```

Legal?



Kap. 5.9

Das Substitutionsprinzip

```

class Oberklasse {
    public void printSomething() {
        System.out.println("Oberklasse");
    }
}

class Unterklasse extends Oberklasse {

    @Override
    public void printSomething() {
        System.out.println("Unterklasse");
    }

    void printSomethingElse() {
        System.out.println("ZUSÄTZLICH");
    }
}

public class Main {
    public static void main(String[] args) {
        Oberklasse ref = new Unterklasse() ; ✓
        ref.printSomething() ; ✓
        ref.printSomethingElse() ; ✗
    }
}

```



Code-Demo



CUT: Q&A



Polymorphie



Kap. 5.11

Polymorphie

```
Object ref = new String("Ich bin die Unterklasse") ;  
System.out.println("Der Hash Code ist " + ref.hashCode()) ;
```

- Wenn eine Referenz auf Object vorliegt...
- ... diese allerdings auf ein Objekt vom Typ String zeigt...
- ... welches die Methode hashCode() überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - Object.hashCode()
 - String.hashCode()



Kap. 5.11

Polymorphie

```
Object ref = new String("Ich bin die Unterklasse") ;  
System.out.println("Der Hash Code ist " + ref.hashCode()) ;
```

- Wenn eine Referenz auf Object vorliegt...
- ... diese allerdings auf ein Objekt vom Typ String zeigt...
- ... welches die Methode hashCode() überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - Object.hashCode()
 - String.hashCode()
- kann er nicht wissen!!



Kap. 5.11

Polymorphie

- Wenn eine Referenz auf Oberklasse vorliegt...
- ... diese allerdings auf ein Objekt vom Typ Unterklasse zeigt...
- ... welches die Methode printSomething() überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - Oberklasse.printSomething()
 - Unterklasse.printSomething()
- kann er nicht wissen!!

```
class Oberklasse {  
    public void printSomething() {  
        System.out.println("Oberklasse");  
    }  
  
class Unterklasse extends Oberklasse {  
  
    @Override  
    public void printSomething() {  
        System.out.println("Unterklasse");  
    }  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        Oberklasse ref = new Unterklasse();  
        ref.printSomething();  
    }  
}
```

Ausgabe?



Kap. 5.11

Polymorphie

- Wenn eine Referenz auf Oberklasse vorliegt...
- ... diese allerdings auf ein Objekt vom Typ Unterklasse zeigt...
- ... welches die Methode printSomething() überschrieben hat...
- ... wie weiss der Compiler (zur Compile-Zeit), welche Methode aufgerufen werden muss?
 - Oberklasse.printSomething()
 - Unterklasse.printSomething()
- Lösung: JVM verifiziert zur **Laufzeit**, auf welchen Typ eine Referenz **wirklich** zeigt
 - hier: Unterklasse
 - JVM ruft Unterklasse.printSomething() auf

```

class Oberklasse {
    public void printSomething() {
        System.out.println("Oberklasse");
    }
}

class Unterklasse extends Oberklasse {

    @Override
    public void printSomething() {
        System.out.println("Unterklasse");
    }
}

public class Main {

    public static void main(String[] args) {

        Oberklasse ref = new Unterklasse();
        ref.printSomething();
    }
}

```

Unterklasse



Kap. 5.9

Formal: Polymorphie

- Bei Methodenaufrufen weiß der Compiler aufgrund des Substitutionsprinzips nicht, auf welche Instanz eine Referenz zeigt (Oberklasse oder abgeleitete Klassen)
- Bei Methodenaufrufen wird zur Laufzeit geprüft, auf was eine Referenz tatsächlich zeigt
- Es wird immer die Methode der Klasse aufgerufen, auf die die Referenz zeigt
- Folge: selbe Referenz kann bei Methodenaufrufen unterschiedliches Verhalten haben (Polymorphie)



Beispiele für Typsubstitution und Polymorphie aus dem Projekt

```
class SpaceInvadersGame extends GameLoop {  
    // ...  
  
    public static void main(String[] args) {  
        GameLoop game = new SpaceInvadersGame();  
        game.runGame(args);  
    }  
}
```



Cut: Q&A



Zugriffsrechte & Vererbung



Kap. 5.2

Zugriffsrechte und Vererbung

- Betrifft hpts. Methoden und Attribute
- Bisher kennen wir die Zugriffsrechte
 - `public`: jeder darf zugreifen
 - `package-public`: nur Klassen des eigenen Packages können zugreifen
 - für Vererbung ist dies nicht ausreichend!



Zugriffsrechte und Vererbung

- Betrifft hpts. Methoden und Attribute
- Bisher kennen wir die Zugriffsrechte
 - `public`: jeder darf zugreifen
 - `package-public`: nur Klassen des eigenen Packages können zugreifen
 - für Vererbung ist dies nicht ausreichend!

warum?



Zugriffsrechte und Vererbung

- Vier Typen von Zugriffsrechten für Klassen, Methoden und Attribute
 - public (Schlüsselwort `public`): alle Klassen dürfen zugreifen
 - package-public (kein Schlüsselwort): alle Klassen im selben Package dürfen zugreifen
 - protected (Schlüsselwort `protected`): Unterklassen dürfen zugreifen
 - private (Schlüsselwort `private`): nur eigene Klasse darf zugreifen



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!



Programmierung 2

Vorlesung 5: Vererbung II

Alexander Gepperth, Mai 2022



Heute

- Recap: Vererbung
- Recap: Überschreiben von Methoden
- Recap: Substitutionsprinzip
- **NEU:** Konstruktoren
- **NEU:** Abstrakte Basisklassen



Recap: Überschreiben von Methoden

- Überschreiben/Ersetzen einer Methode in einer Unterklasse
 - Name, Rückgabetyp und Parameter müssen **exakt** übereinstimmen
 - **Annotation** @Override erlaubt dies zur Compile-Zeit zu prüfen
 - ursprüngliche Methode weiter verfügbar durch z.B. super.methodename(. . .)



- Vererbung: kurze Wiederholung
- Technische Details:
 - Konstruktoren
 - Überschreiben von Methoden
 - Das Substitutionsprinzip**
 - Die Klasse Object
 - Polymorphie
 - Zugriffsrechte
- Projekt: die Klassen Playground und SpaceInvadersLevel

Kap. 5.9



Das Substitutionsprinzip

- Eine UnterkLASSE ist Spezialfall ihrer OberKLASSE: kann **formal** alles was die OberKLASSE auch kann (und evtl. mehr)



Das Substitutionsprinzip

- Eine UnterkLASSE ist Spezialfall ihrer OberKLASSE: kann **formal** alles was die OberKLASSE auch kann (und evtl. mehr)
 - da, wo Referenz auf OberKLASSE erwartet wird, kann auch Referenz auf UnterkLASSE stehen
 - allerdings: über die Referenz auf OberKLASSE dürfen nur Methoden aus OberKLASSE aufgerufen werden (**WARUM?**)
 - insbesondere: da wo Referenz auf Object erwartet wird, kann jede Klasse stehen (aber: nur Methoden aus Object verfügbar)

- Vererbung: kurze Wiederholung
- Technische Details:
 - Konstruktoren
 - Überschreiben von Methoden
 - Das Substitutionsprinzip
 - Die Klasse Object
 - Polymorphie
 - Zugriffsrechte
- Projekt: die Klassen Playground und SpaceInvadersLevel



Kap. 5.9

```
class Oberklasse {  
    public void printSomething() {  
        System.out.println("Oberklasse");  
    }  
}  
  
class Unterklasse extends Oberklasse {  
    @Override  
    public void printSomething() {  
        System.out.println("Unterklasse");  
    }  
    void printSomethingElse() {  
        System.out.println("ZUSÄTZLICH");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Oberklasse ref = new Unterklasse();  
        ref.printSomething();  
        ref.printSomethingElse();  
    }  
}
```

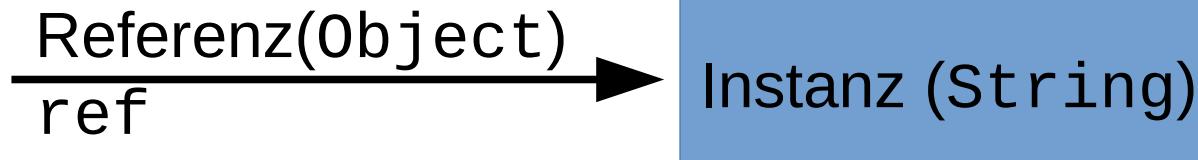
Legal?



Polymorphie

- Merkregel:
 - Typ der Referenz bestimmt, welche Methoden aufgerufen werden **können**
 - Typ der Instanz bestimmt, welche Methode tatsächlich aufgerufen **wird**

```
Object ref = new String("Ich bin die Unterklasse") ;  
System.out.println("Der Hash Code ist " + ref.hashCode()) ;
```



- Vererbung: kurze Wiederholung
- Technische Details:
 - Konstruktoren
 - Überschreiben von Methoden
 - Das Substitutionsprinzip
 - Die Klasse Object
 - Polymorphie
 - Zugriffsrechte
- Projekt: die Klassen Playground und SpaceInvadersLevel



Kap. 5.9

```
class Oberklasse {  
    public void printSomething() {  
        System.out.println("Oberklasse");  
    }  
}  
  
class Unterklasse extends Oberklasse {  
    @Override  
    public void printSomething() {  
        System.out.println("Unterklasse");  
    }  
    void printSomethingElse() {  
        System.out.println("ZUSÄTZLICH");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Oberklasse ref = new Unterklasse(); ✓  
        ref.printSomething(); ✓  
        ref.printSomethingElse(); ✗  
    }  
}
```

- Vererbung: kurze Wiederholung
- Technische Details:
 - Konstruktoren
 - Überschreiben von Methoden
 - Das Substitutionsprinzip
 - Die Klasse Object
 - Polymorphie
 - Zugriffsrechte
- Projekt: die Klassen Playground und SpaceInvadersLevel



public, protected, private,

• • •

- Vier Typen von Zugriffsrechten für Klassen, Methoden und Attribute
 - public (Schlüsselwort `public`): alle Klassen dürfen zugreifen
 - package-public (kein Schlüsselwort): alle Klassen im selben Package dürfen zugreifen
 - protected (Schlüsselwort `protected`): Unterklassen dürfen zugreifen
 - private (Schlüsselwort `private`): nur eigene Klasse darf zugreifen



CUT



Konstruktoren



Konstruktoren

- Spezielle Methoden die bei der Instanziierung einer Klasse aufgerufen werden **müssen**
- Ziel: Initialisierung von Attributen
- Eine Klasse kann mehrere Konstruktoren haben (verschieden Parameter)



Kap. 5.8.6

Konstruktoren

- Terminologie:
 - **Standardkonstruktor** ist ein Konstruktor ohne Parameter
 - **parametrisierter Konstruktor** ist ein Konstruktor mit Parametern
 - Parameter werden ggf. bei der Instanziierung übergeben

```
String standard = new String() ;  
String parametrisiert = new String("xx") ;
```



Kap. 5.8.6

Konstruktoren

- Falls eine Klasse **keinen eigenen Konstruktor** definiert:
 - Standardkonstruktor wird vom Compiler automatisch erzeugt & hinzugefügt
 - dieser automatische Standardkonstruktor tut nichts, außer den Standardkonstruktor der Oberklasse aufzurufen!
- falls **eigener Konstruktor** definiert (parametrisiert oder Standard): kein automatisch erzeugter Standardkonstruktor!



Kap. 5.8.6

Konstruktoren

- Beispiel:
 - Deklaration von Klasse1
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standardkonstruktor

```
class Klasse1 {  
    int x = 0 ;  
  
    public Klasse1 () {  
        this.x = 0;  
    }  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse1 o = new Klasse1 () ;
```

OK ?



Kap. 5.8.6

Konstruktoren

- Beispiel:
 - Deklaration von Klasse1
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standardkonstruktor

```
class Klasse1 {  
    int x = 0 ;  
  
    public Klasse1 () {  
        this.x = 0;  
    }  
  
    public int getX () {  
        return this.x ;  
    }  
}
```

```
Klasse1 o = new Klasse1 () ;
```



ja, Standardkonstruktor existiert!



Kap. 5.8.6

Konstruktoren

- Beispiel:
 - Deklaration von Klasse2
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standardkonstruktor

```
class Klasse2 {  
    int x = 0 ;  
  
    public int getX() {  
        return this.x ;  
    }  
}
```

```
Klasse2 o = new Klasse2();
```

OK ?



Kap. 5.8.6

Konstruktoren

- Beispiel:
 - Deklaration von Klasse2
 - kein eigener Konstruktor definiert
 - Instanziierung mit Standardkonstruktor

```
class Klasse2 {
    int x = 0 ;

    public int getX() {
        return this.x ;
    }
}
```

```
Klasse2 o = new Klasse2();
```



ja, Standardkonstruktor
automatisch erzeugt!



Kap. 5.8.6

Konstruktoren

- Beispiel:
 - Deklaration von Klasse3
 - jetzt: parametrisierter Konstruktor definiert
 - Instanziierung mit Standardkonstruktor

```
Klasse3 o = new Klasse3();
```

```
class Klasse3 {  
    int x = 0;  
  
    public Klasse3(int x) {  
        this.x = x;  
    }  
  
    public int getX() {  
        return this.x;  
    }  
}
```

OK ?



Kap. 5.8.6

Konstruktoren

- Beispiel:
 - Deklaration von Klasse3
 - jetzt: parametrisierter Konstruktor definiert
 - Instanziierung mit Standardkonstruktor

```
Klasse3 o = new Klasse3();
```

```
class Klasse3 {  
    int x = 0;  
  
    public Klasse3(int x) {  
        this.x = x;  
    }  
  
    public int getX() {  
        return this.x;  
    }  
}
```

nicht ok, weil kein Standardkonstruktor erzeugt wurde!



Kap. 5.8 (bis 5.14)

Konstruktoren & Vererbung

- Vererbungsmechanismus gilt NICHT für Konstruktoren!
 - jede Unterklasse muss eigene Konstruktoren definieren
 - falls das nicht passiert, wird ein Standardkonstruktor erzeugt der den Std.K. der Oberklasse aufruft
 - ein Konstruktor der Unterklasse muss als erste Aktion einen Konstruktor der Oberklasse aufrufen: mit `super()` oder `super(param1, param2, ...)`
 - falls das nicht getan wird, wird automatisch der Standardkonstruktor der Oberklasse aufgerufen (Fehler falls nicht-existent)



CUT: Q&A



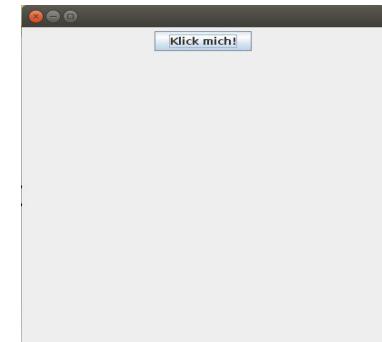
Abstrakte Basisklassen



Kap. 5.12

Abstrakte Basisklassen

- Klassen können verschiedene Ziele haben:
 - darf nicht abgeleitet werden (String, Integer, ...), definiert mit Schlüsselwort `final`
 - kann abgeleitet werden (um Methoden zu ersetzen)
 - **muss** abgeleitet werden (abstrakte Basisklasse), definiert mit Schlüsselwort `abstract`





Abstrakte Basisklassen: Eigenschaften

- können **abstrakte Methoden** enthalten (ohne Code) die erst in Unterklassen definiert werden
- daher: A.B. **müssen** abgeleitet werden
- daher: A.B. **können nicht** instanziert werden
- mindestens eine abstrakte Methode → Klasse ist abstrakte Basisklasse
- Konstruktoren möglich, werden ja nicht vererbt



Kap. 5.12

Abstrakte Basisklassen: Definition

Markierung als abstrakt

```
abstract class AbstractBaseClass {  
    public int attribute1 ;  
  
    abstract void method1() ;  
  
    abstract void method2 () ;  
  
    void method3 () {  
        System.out.println("Nicht abstrakt!");  
    }  
  
    AbstractBaseClass () {  
    }  
}
```

abstrakte Methoden haben
keine Implementierung!

nicht-abstrakte Methoden
möglich!

Konstruktor kann nicht-
abstrakt sein!



Kap. 5.12

Abstrakte Basisklassen: Instanzierung

- A.B. können nicht instanziert werden da “Methoden fehlen”
- Methoden können in abgeleiteten Klassen durch Überschreiben “nachgereicht” werden
- Regel:
 - eine Klasse, die von einer A.B. ableitet kann instanziert werden wenn alle abstrakten Methoden durch “normale” überschrieben wurden
 - sonst: Klasse ist wieder abstrakt



Kap. 5.12

Abstrakte Basisklassen: Verwendungszweck

- A.B. definieren Methoden die in abgeleiteten Klassen auf jeden Fall vorhanden sind → Schnittstellendefinition
- falls die Implementierung von der abgeleiteten Klasse sowieso überschrieben werden soll: kann genausogut **abstract** sein!
- wichtig: A.B. kann auch nicht-abstrakte Methoden enthalten
(gemeinsame Funktionalität aller Unterklassen)



Beispiele aus dem Projekt

- GameLoop (Methode nextLevel ist abstrakt)
- SpaceInvadersLevel (Methode getName ist abstrakt)



Cut: Q&A



Zusammenfassung

- Neue Sprachelemente:
 - `abstract` zur Deklaration abstrakter Methoden und Klassen
 - `super()` oder `super(param1, ...)` zum Aufruf von Konstruktoren der Oberklasse



Zusammenfassung

- Nicht ganz neue Sprachelemente:
 - extends zur Angabe der Oberklasse der Vererbung
 - super.methode () oder
super.methode (param1, ...) zum Aufruf von Methoden der Oberklasse
 - private, protected, public (und package-public) zur Zugriffskontrolle für Methoden und Attribute



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!



Motivation für Vererbung

- Copy&Paste vermeiden,
Wiederverwendung gemeinsamen Codes!
- Baukastensystem durch Überschreiben
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!
- Copy&Paste vermeiden!



Programmierung 2

Hinter den Kulissen III
Alexander Gepperth, Mai 2022



Kap. 5.9

Wiederholung: Substitutionsprinzip und Polymorphie



Wiederholung: Substitutionsprinzip und Polymorphie

- Vererbung von Oberklasse zu Unterklasse:
"ist-Spezialfall-von"-Beziehung
- Unterklasse ist Spezialfall von Oberklasse
- Folge: da wo Oberklasse erwartet wird
kann auch immer Unterklasse stehen!



Kap. 5.9

Wiederholung: Substitutionsprinzip und Polymorphie

- Insbesondere: Referenz auf Oberklasse darf auf Instanz von Unterklasse zeigen!

```
Oberklasse referenz = new Unterklasse() ;  
referenz.methode();
```

Referenzdeklaration

Instanzerzeugung
mit new



Kap. 5.9

Wiederholung: Substitutionsprinzip und Polymorphie

- Insbesondere: Referenz auf Oberklasse darf auf Instanz von Unterklasse zeigen!

```
Oberklasse referenz = new Unterklasse() ;  
referenz.methode() ;
```

- Typ der Referenz bestimmt, welche Methoden aufgerufen werden **können**
- Typ der Instanz bestimmt, welche Methoden **tatsächlich** ausgeführt werden



Kap. 5.9

Wiederholung: Substitutionsprinzip und Polymorphie

- Insbesondere: Referenz auf Oberklasse darf auf Instanz von Unterklasse zeigen!

```
Oberklasse referenz = new Unterklasse() ;  
referenz.methode() ;
```

- Typ der Referenz bestimmt, welche Methoden aufgerufen werden können
- **Falls die Referenz methode() nicht besitzt,
→ Aufruf illegal!**



Wiederholung: Substitutionsprinzip und Polymorphie

- Grundidee bei Behandlung der Substitution durch Java:
maximale Prüfbarkeit zur Compile-Zeit
- Es geht nicht darum, ob ein Code zur **Laufzeit** funktioniert,
sondern ob dies zur **Compile-Zeit** garantiert werden kann!
- Begrifflichkeiten (für Test.java, siehe **Demo**):
 - Compile-Zeit: Übersetzung in Maschinencode für die JVM
(es wird **nichts** ausgeführt): Test.java → Test.class
 - Laufzeit: Ausführung von Test.class



CUT:Q&A



Exkurs: warum Polymorphie notwendig ist

- Eclipse-Beispiel: Projektaufgabe dieser Woche



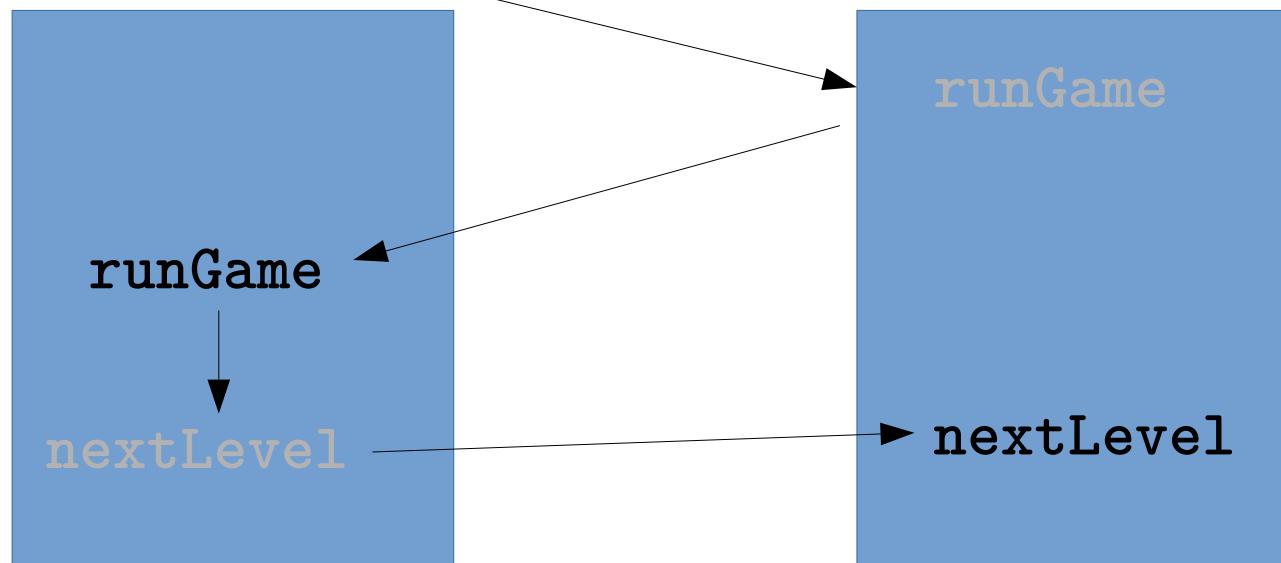
Exkurs: warum Polymorphie notwendig ist

- Betrachten wir die Klasse GameLoop
 - Wird abgeleitet in z.B. MyGame, überschreibt nextLevel()
 - MyGame wird instanziert in main
 - In main: mg.runGame wird aufgerufen
 - Vor dem ersten Level: runGame ruft nextLevel auf



Exkurs: warum Polymorphie notwendig ist

mg.runGame()



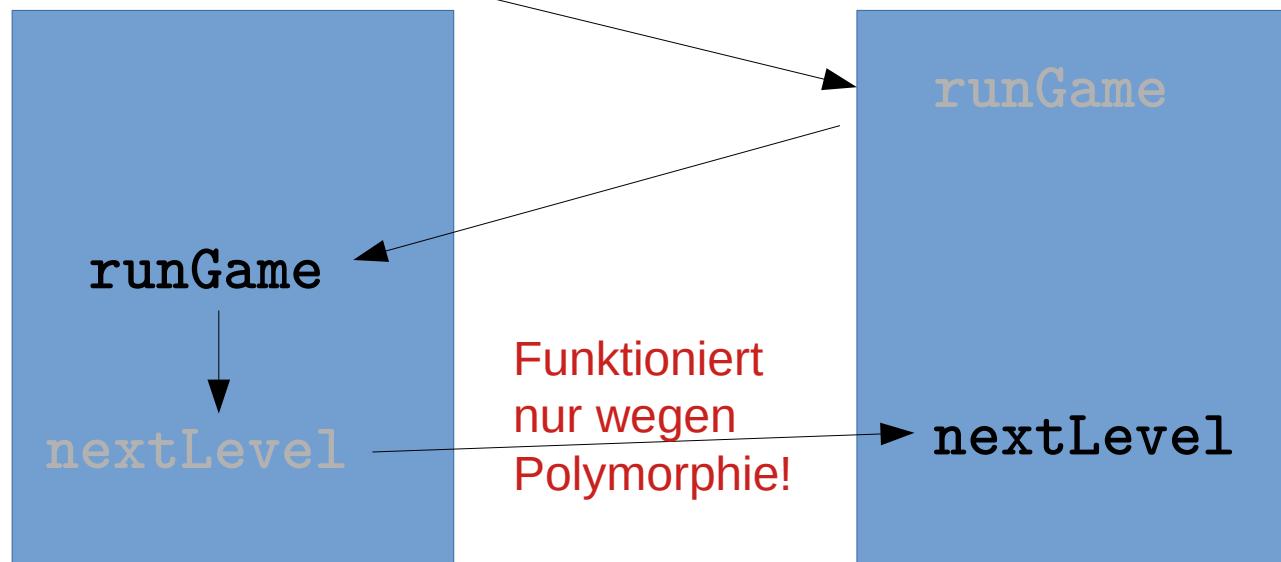
GameLoop:
Referenz this
hat Typ GameLoop

MyGame:
Referenz this
hat Typ MyGame



Exkurs: warum Polymorphie notwendig ist

mg.runGame()



GameLoop:
Referenz this
hat Typ GameLoop

MyGame:
Referenz this
hat Typ MyGame



Fazit

- Überschreiben von Methoden ist gewollt, damit alten Methoden neue Funktionalitäten untergeschoiben werden können
- Dafür ist Polymorphie zwingend nötig!
- Vorsicht: Polymorphie gilt nicht für Attribute!
Hier entscheidet allein der Typ der Referenz
(Demo)!



Primitive Datentypen und Referenzdatentypen



Kap. 3.4

Zusammenfassung

- Jede Variable ist entweder Referenz oder primitiv
- In beiden Fällen sind Variablen Namen für Speicherbereiche
 - primitiver Datentyp: Speicherbereich repräsentiert Wert
 - Referenzdatentyp: Speicherbereich repräsentiert Adresse eines Werts



Kap. 3.4

Zusammenfassung

- Referenzvariablen sind:
 - Arrays primitiver Typen: `char[]`, `float[]`, `int[]`, ...
 - Instanzen von Klassen:
 - Vordefinierte Klassen: `java.util.Scanner`, `String`, ...
 - selbst definierte Klassen: `Playground`, `GameObject`, `GameLoop`, ...
- Primitive Datentypen sind: `int`, `float`, `char`, `double`, `byte`, `long`, ...



Kap. 3.4

Zusammenfassung

- Primitive Datentypen sind einfach zu handhaben
- Referenzdatentypen sind mächtig, erfordern allerdings Verständnis vor allem bei
 - Zuweisungen (bereits behandelt)
 - Testen auf Gleichheit
 - Parameterübergabe an Methoden



Testen auf Gleichheit



Kap. 3.7

Testen auf Gleichheit mit ==

- Vergleicht stets Speicherinhalt von 2 Variablen
 - primitive Datentypen: Wert
 - Referenzen: Speicheradresse!
 - testet ob Referenzen auf das selbe Objekt zeigen
(„dasselbe Objekt referenzieren“)
 - testen NICHT ob die referenzierten Objekte dieselben Werte haben!
 - **VORSICHT!!**
- 
- A large green checkmark is positioned above the first bullet point, indicating a correct statement. A large red question mark is positioned above the second bullet point, indicating a common misconception or warning.



Kap. 3.7

Testen auf Gleichheit mit ==

- Beispiel für Referenzvariablen

```
String s = "Hall" ;  
String s2 = "0" ;  
String g = "Hallo" ;  
if ((s+s2) == g) {  
    System.out.println("gleich!");  
} else {  
    System.out.println("ungleich!");  
}
```

Ausgabe?



Kap. 3.7

Testen auf Gleichheit mit ==

- Beispiel für Referenzvariablen
- Vergleich testet auf Referenz!

```
String s = "Hall" ;  
String s2 = "0" ;  
String g = "Hallo" ;  
if ((s+s2) == g) {  
    System.out.println("gleich!");  
} else {  
    System.out.println("ungleich!");  
}
```

ungleich!



Kap. 3.7

Testen auf Gleichheit mit ==

- Beispiel für primitive Typen

```
int a = 3 ;
int b = 3 ;
if (a == b) {
    System.out.println("gleich!");
} else {
    System.out.println("ungleich!");
}
```

Ausgabe?



Kap. 3.7

Testen auf Gleichheit mit ==

- Beispiel für primitive Typen
- Vergleich test auf Wert!

```
int a = 3 ;  
int b = 3 ;  
if (a == b) {  
    System.out.println("gleich!");  
} else {  
    System.out.println("ungleich!");  
}
```

gleich!



Kap. 3.7

Testen auf Gleichheit mit ==

- Merken:
 - bei primitiven Datentypen wird immer der Wert verglichen
 - bei Referenztypen wird immer die Referenz (der Pfeil) verglichen, nicht der Wert
 - falls der Wert verglichen werden soll: spezielle Methoden benutzen!



Kap. 3.7



Testen auf Gleichheit mit ==

- Korrigiertes Beispiel für Referenzvariablen
- Haben zwei String den gleichen Wert?

```
String s = "Hall" ;  
String s2 = "0" ;  
String g = "Hallo" ;  
if ((s+s2) == g) {  
    System.out.println("gleich!");  
} else {  
    System.out.println("ungleich!");  
}
```

== ist sinnlos



Kap. 3.7

Testen auf Gleichheit mit ==

- Korrigiertes Beispiel für Referenzvariablen
- Haben zwei String den gleichen Wert?

```
String s = "Hall" ;  
String s2 = "0" ;  
String g = "Hallo" ;  
if ((s+s2).equals (g)) {  
    System.out.println("gleich!");  
} else {  
    System.out.println("ungleich!");  
}
```

benutze equals!



Kap. 3.7



Testen auf Gleichheit mit equals

- Korrigiertes Beispiel für Referenzvariablen
- Haben zwei String den gleichen Wert?

```
String s = "Hall" ;  
String s2 = "0" ;  
String g = "Hallo" ;  
if ((s+s2).equals (g)) {  
    System.out.println("gleich!");  
} else {  
    System.out.println("ungleich!");  
}
```

gleich!



Pass-by-value und pass-by-reference



Kap. 3.7

Pass-by-value und pass-by-reference

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?

```
void f( int x ) {  
    x = 3 ;  
}
```



Kap. 3.7

Pass-by-value und pass-by-reference

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?



```
void f( int x ) {  
    x = 3 ; // legal?  
}
```



Kap. 3.7

Pass-by-value und pass-by-reference

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?

```
void f( int x ) {  
    x = 3 ; // legal!  
}
```





Kap. 3.7

Pass-by-value und pass-by-reference

Parameter ist primärer Datentyp

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?
 - was hat das "außerhalb" für Konsequenzen?

```
void f( int x ) {  
    x = 3 ; // legal!  
}
```



```
int intVar = 0 ;  
f(intVar) ;  
System.out.println(intVar);
```





Kap. 3.7

Pass-by-value und pass-by-reference

Parameter ist primärer Datentyp

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?
 - was hat das "außerhalb" für Konsequenzen?
keine!

```
void f( int x ) {  
    x = 3 ; // legal!  
}
```



```
int intVar = 0 ;  
f(intVar) ;  
System.out.println(intVar);
```

Ausgabe: 0



Kap. 3.7

Pass-by-value und pass-by-reference

anderer Fall:
Referenzdatentyp!

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?



```
void f( int [] x ) {  
    x [0]= 3 ; //legal?  
}
```



Kap. 3.7

Pass-by-value und pass-by-reference

anderer Fall:
Referenzdatentyp!

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?
 - was hat das “außerhalb” für Konsequenzen?

```
void f( int [ ] x ) {  
    x [0]= 3 ; //legal!  
}
```



```
int [ ] intArray = {1,2,3} ;  
f(intArray) ;  
System.out.println(intArray[0]);
```





Kap. 3.7

Pass-by-value und pass-by-reference

anderer Fall:
Referenzdatentyp!

- Java-Methoden können Parameter haben
 - dürfen Sie Parameter verändern?
 - was hat das “außerhalb” für Konsequenzen?

Änderung!

```
void f( int [] x ) {  
    x [0] = 3 ; //legal!  
}
```

```
int [] intArray = {1,2,3} ;  
f(intArray) ;  
System.out.println(intArray[0]);
```

Ausgabe: 3



Kap. 3.7

Pass-by-value und pass-by-reference

- Offensichtlich unterschiedliches Verhalten, wenn primitive oder Referenzdatentypen an Methoden übergeben werden
 - primitive Datentypen: können nicht geändert werden (**pass-by-value**)
 - Referenzdatentypen: können geändert werden (**pass-by-reference**)
- **Warum?**



Kap. 3.7

Pass-by-value und pass-by-reference

- Java benutzt ein sehr einfaches Verfahren, um Argumente an Methoden zu übergeben: die **Kopie!**
 - der Speicherbereich jedes Parameters wird kopiert und die Methode arbeitet auf der Kopie
 - Unterschied zwischen primitiven und Referenzdatentypen?

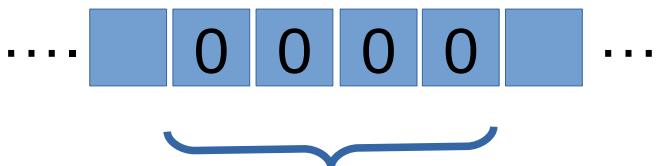


Kap. 3.7

Pass-by-value

```
int intVar = 0 ;  
f(intVar) ;
```

```
void f( int x ) {  
    x = 3 ;  
}
```



Wert von intVar
außerhalb der Methode

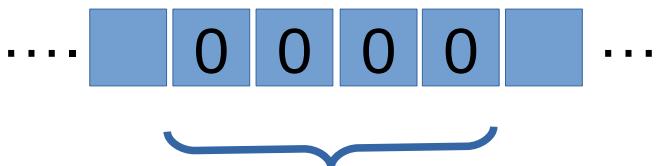


Kap. 3.7

Pass-by-value

```
int intVar = 0 ;  
f(intVar) ;
```

```
void f( int x ) {  
    x = 3 ;  
}
```



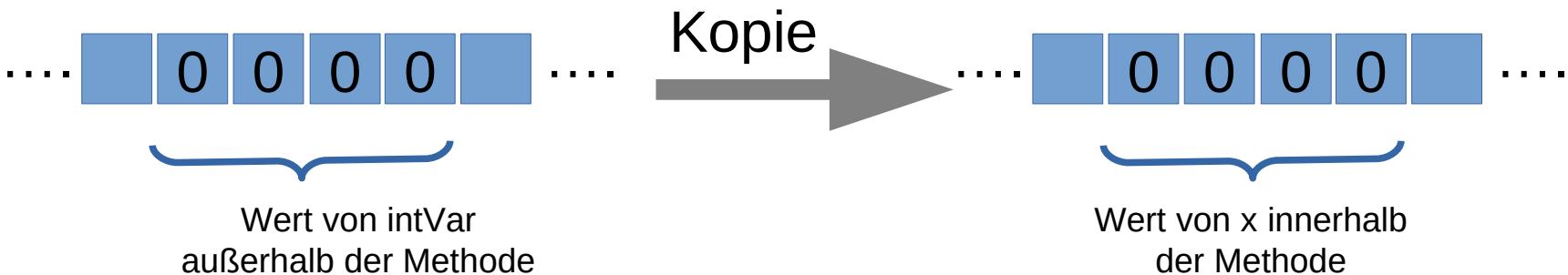
Wert von intVar
außerhalb der Methode



Pass-by-value

```
int intVar = 0 ;  
f(intVar) ;
```

```
void f( int x ) {  
    x = 3 ;  
}
```

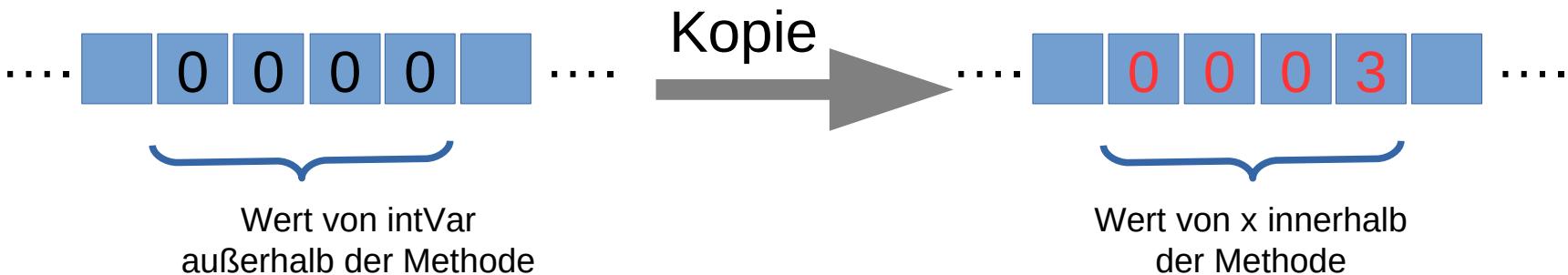




Pass-by-value

```
int intVar = 0 ;  
f(intVar) ;
```

```
void f( int x ) {  
    x = 3 ;  
}
```

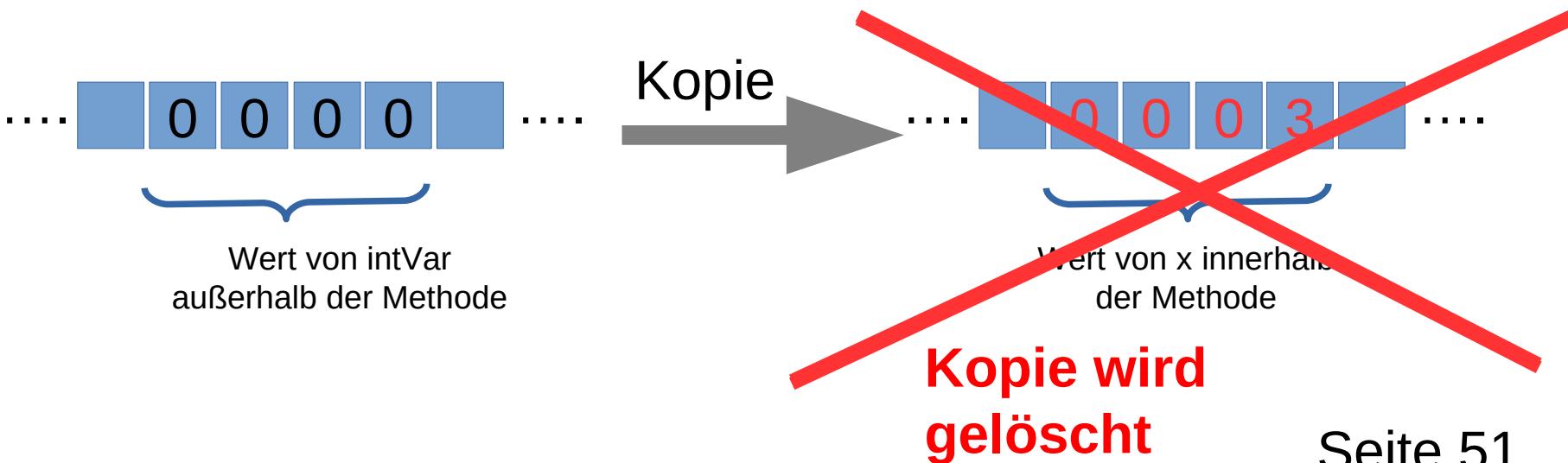




Pass-by-value

```
int intVar = 0 ;  
f(intVar) ;
```

```
void f( int x ) {  
    x = 3 ;  
}
```



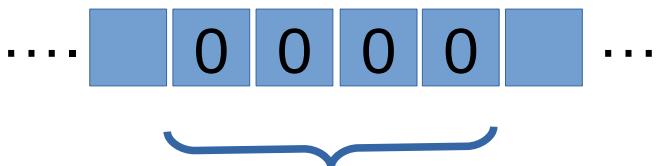


Kap. 3.7

Pass-by-value

```
int intVar = 0 ;  
f(intVar) ;
```

```
void f( int x ) {  
    x = 3 ;  
}
```



Wert von intVar
außerhalb der Methode

**Wert wurde nicht
verändert!**

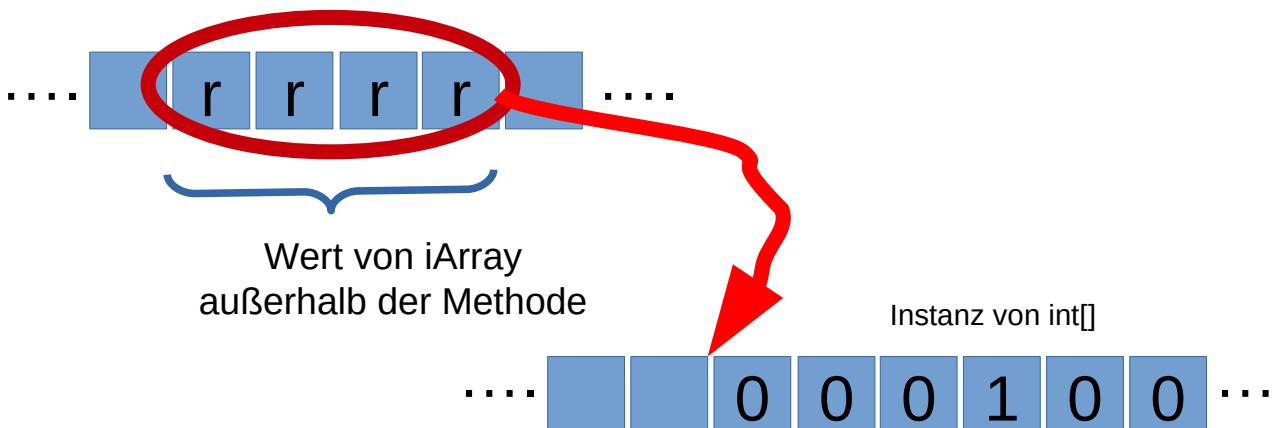


Kap. 3.7

Pass-by-reference

```
int[] iArray = {1, 2, 3};  
f(iArray);
```

```
void f( int[] x ) {  
    x[0] = 3;  
}
```



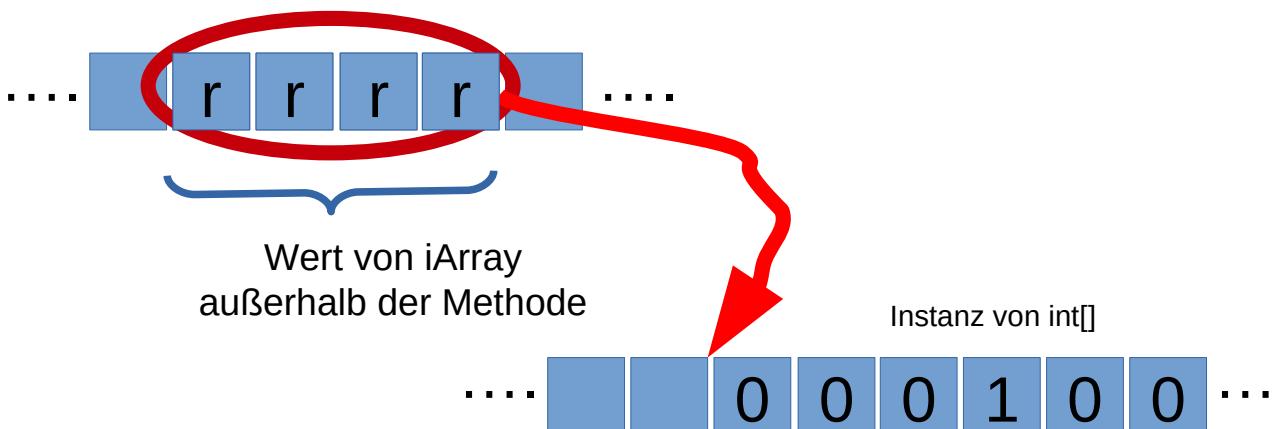


Kap. 3.7

Pass-by-reference

```
int[] iArray = {1, 2, 3};  
f(iArray);
```

```
void f( int[] x ) {  
    x[0] = 3;  
}
```



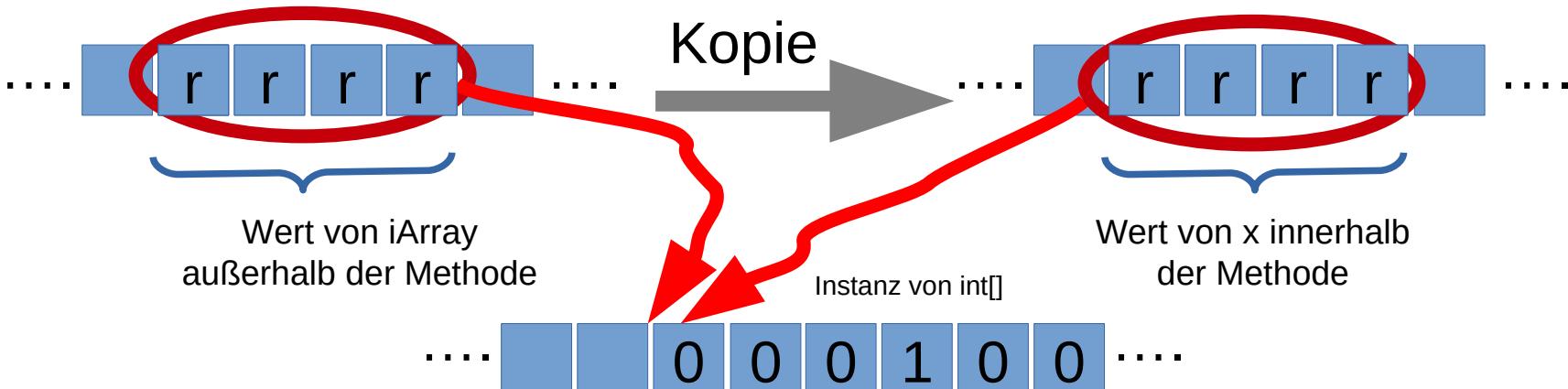


Kap. 3.7

Pass-by-reference

```
int[] iArray = {1, 2, 3};  
f(iArray);
```

```
void f( int[] x ) {  
    x[0] = 3;  
}
```

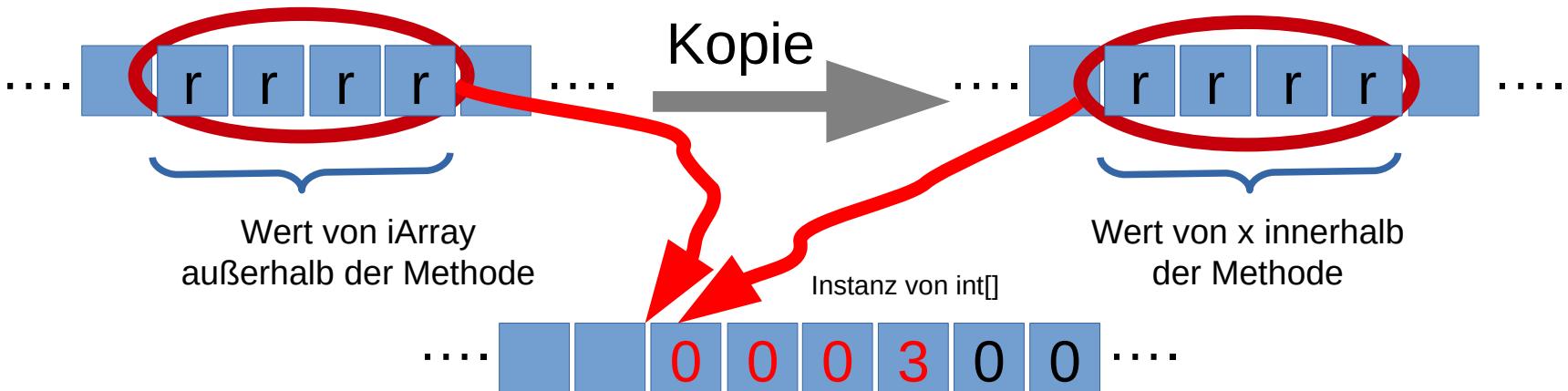




Pass-by-reference

```
int[] iArray = {1, 2, 3};  
f(iArray);
```

```
void f( int[] x ) {  
    x[0] = 3;  
}
```



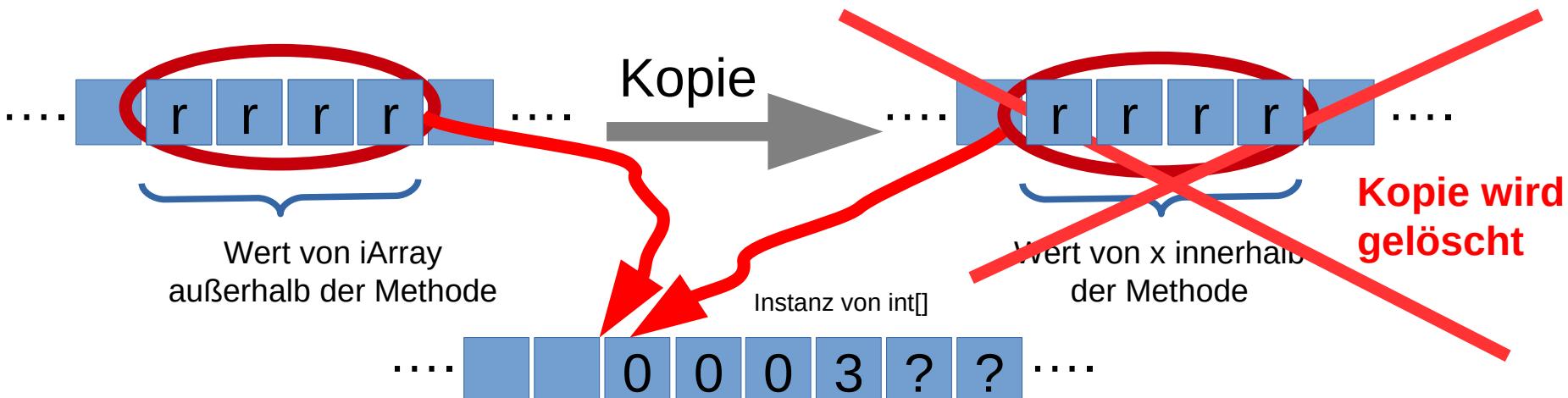


Kap. 3.7

Pass-by-reference

```
int[] iArray = {1, 2, 3};  
f(iArray);
```

```
void f( int[] x ) {  
    x[0] = 3;  
}
```



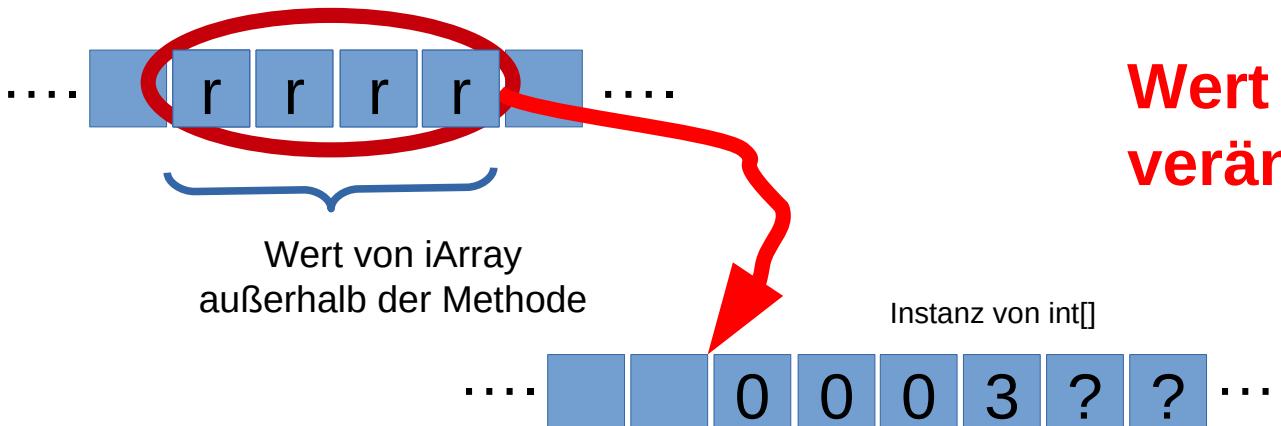


Kap. 3.7

Pass-by-reference

```
int[] iArray = {1, 2, 3};  
f(iArray);
```

```
void f( int[] x ) {  
    x[0] = 3;  
}
```



**Wert wurde
verändert!**



Das erklärt einiges...

- Warum überhaupt Referenzdatentypen?
 - weil es oft gewünscht ist dass Methoden “außen liegende” Daten verändern:
Seiteneffekte
 - weil nur die Referenz kopiert wird nicht die Daten → schneller, sparsamer



Das erklärt einiges...

- Warum überhaupt primitive Datentypen?
 - Erbschaft von C/C++
 - sind speichereffizienter, vor allem wenn sie in Arrays benutzt werden
 - primitiver Datentyp benötigt keine Referenz (4 Byte)
 - primitiver Datentyp benötigt keine zusätzliche Reservierung mit `new`



Kap. 8.2

Wrapper-Datentypen als Ersatz für primitive Datentypen

- Für jeden primitiven Datentyp definiert Java einen Wrapper-Datentyp als “Ersatz”
 - Integer (int)
 - Float (float)
 - Double (double)
 - Character (char)
 - Boolean (bool)
 - Byte (byte)
 - Short (short)
 - Long (long)



Kap. 8.2

Wrapper-Datentypen als Ersatz für primitive Datentypen

- Jeder dieser Referenztypen bietet sehr nützliche Methoden an:
 - equals, toString, valueOf, ...
 - existieren als statische oder Objektmethoden
- Erzeugung durch Konstruktor oder valueOf()
- Boxing



Kap. 8.2

Wrapper-Datentypen als Ersatz für primitive Datentypen

```
// Konstruktor  
Integer intVar = new Integer(20) ;  
intVar = Integer.valueOf ("20") ; // valueOf  
intVar = Integer.valueOf (20) ; // valueOf  
intVar = 20 ; // Boxing
```

→ Übungen!



Programmierung 2

Collections und Generics

Alexander Gepperth, Juni 2022



Arrays in Java

- Sehr nützliches Konstrukt zur Verwaltung von Werten **dieselben** Typs
- Sowohl primitive als auch Referenzdatentypen sind erlaubt!

```
String[] stringArray = new String[42] ;  
int[] intArray = new int[43] ;  
Integer[] otherIntArray = new Integer [44] ;
```



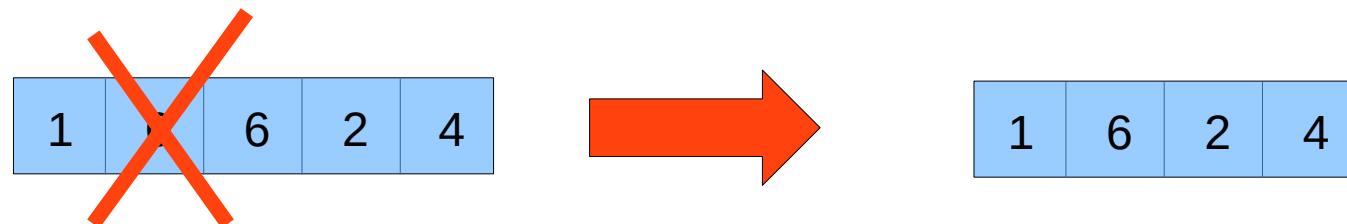
Einschub: Arrays kaputtmachen!

- Bei Referenzdatentypen wurde eine problematische Annahme gemacht:
Kovarianz
Oberklasse [] x = new Unterklasse[10] ;
- Kann benutzt werden, um Instanzen verschiedener Klassen im Array zu speichern
- Siehe **DEMO**

Kap. 3.8

Arrays in Java

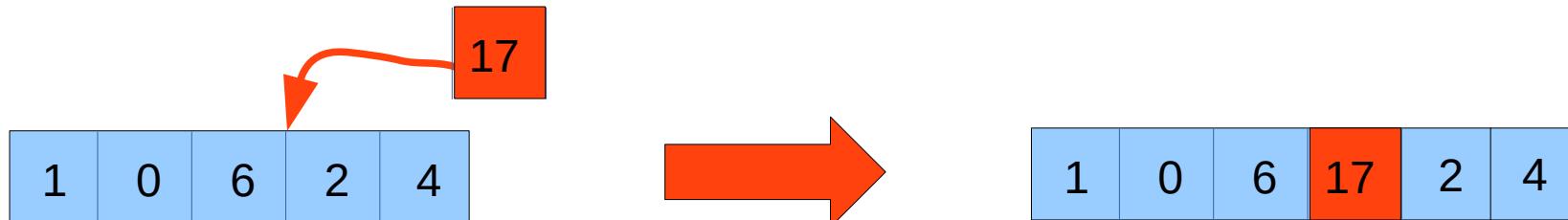
- Allerdings auch sehr unpraktisch:
 - können keine Werte löschen



Kap. 3.8

Arrays in Java

- Allerdings auch sehr unpraktisch:
 - können keine Werte löschen
 - können keine Werte einfügen





Collections

- "Generalisierte Arrays" zur Aufnahme vieler Objekte derselben Klasse (**NUR REFERENZDATENTYPEN!!**)
- Collection-Typen in der Java-Standardbibliothek:
 - ArrayList
 - LinkedList
 - HashMap
- können einfügen, löschen, ...
Ausgabe ?

```
ArrayList<Integer> t =  
    new ArrayList<Integer>() ;  
t.add(1) ; // anhängen  
t.add(0,2) ; // einfügen  
t.remove(0) ; // löschen  
System.out.println(t.get(0)) ;
```



Collections

- "Generalisierte Arrays" zur Aufnahme vieler Objekte derselben Klasse (**NUR REFERENZDATENTYPEN!!**)
- Collection-Typen in der Java-Standardbibliothek:
 - ArrayList
 - LinkedList
 - HashMap
- können einfügen, löschen, ...
Ausgabe **?1**

```
ArrayList<Integer> t =  
    new ArrayList<Integer>() ;  
t.add(1) ; // anhängen  
t.add(0,2) ; // einfügen  
t.remove(0) ; // löschen  
System.out.println(t.get(0)) ;
```



Collections

- Collection-Klassen sind verfügbar im Package `java.util`
- Müssen importiert werden!

```
ArrayList<Integer> t =  
    new ArrayList<Integer>() ;  
t.add(1) ;  
t.add(0,2) ;  
t.remove(0) ;  
System.out.println(t.get(0)) ;
```



Kap. 13.1

Collections

- Collection-Klassen sind verfügbar im Package `java.util`
- Müssen importiert werden!

```
import java.util.ArrayList ;  
ArrayList<Integer> t =  
    new ArrayList<Integer>() ;  
t.add(1) ;  
t.add(0,2) ;  
t.remove(0) ;  
System.out.println(t.get(0)) ;
```



Kap. 13.1

Collections

- Alle Collection-Typen sind **generische Datentypen**
- erhalten als spezielles Argument den Typ, der in ihnen gespeichert ist (analog zu Arrays)
 - generische Programmierung
 - "Generics", "generic classes"
 - Templates in C++
- Details später



Collections:

java.util.ArrayList

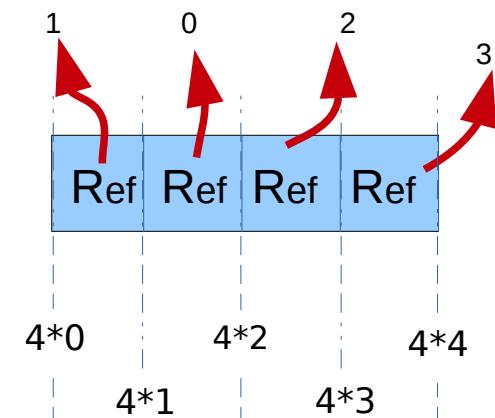
- Wann sollte ArrayList zum Einsatz kommen?
 - falls schneller Zugriff per Index auf beliebige Elemente wichtig ist (also: "was steht an Position 5?")
 - falls nicht eingefügt/gelöscht/angehängt werden muss
 - z.B. bei konstanter Zahl von Elementen!

Kap. 13.2

Collections:

java.util.ArrayList

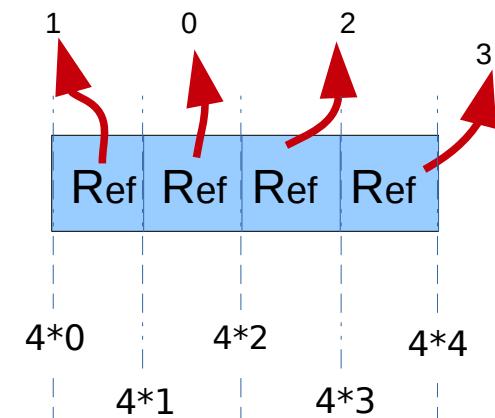
- interne Darstellung der Daten:
zusammenhängender, linearer Speicher
 - Referenzen auf Elemente stehen nacheinander im Speicher
 - n-te Referenz ist an Position $8*n$ (Referenz benötigt 8 Byte)
 - Zugriff über Index sehr effizient möglich!



Kap. 13.2

Collections: ArrayList

- wie funktioniert Finden per Index n?
 - trivial: berechne $8*n$
 - liefere Referenz an Position $8*n$ zurück
 - Geschwindigkeit unabhängig von der Zahl der Elemente!

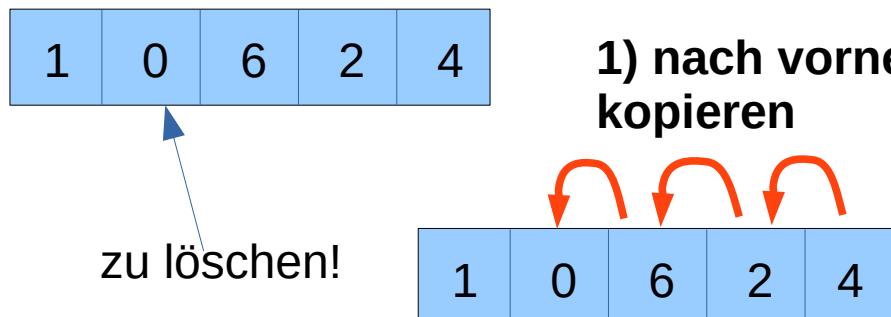




Kap. 13.2

Collections: ArrayList

- Wie funktioniert Löschen?
 - falls letztes Element: einfach, ArrayList verkürzen!
 - sonst: Elemente hinter dem gelöschten Element müssen "nach vorne" kopiert werden
 - sehr ineffizient bei vielen Elementen





Kap. 13.2

Collections: ArrayList

- Wie funktioniert Löschen?
 - falls letztes Element: einfach, ArrayList verkürzen!
 - sonst: Elemente hinter dem gelöschten Element müssen "nach vorne" kopiert werden
 - sehr ineffizient bei vielen Elementen

1	0	6	2	4
---	---	---	---	---

1) nach vorne
kopieren

zu löschen!

1	6	2	4	4
---	---	---	---	---

Kap. 13.2

Collections: ArrayList

- Wie funktioniert Löschen?
 - falls letztes Element: einfach, ArrayList verkürzen!
 - sonst: Elemente hinter dem gelöschten Element müssen "nach vorne" kopiert werden
 - sehr ineffizient bei vielen Elementen

1	0	6	2	4
---	---	---	---	---

zu löschen!

1) nach vorne
kopieren

1	6	2	4	4
---	---	---	---	---

1	6	2	4	4
---	---	---	---	---

2) verkürzen

Kap. 13.2

Collections: ArrayList

- Wie funktioniert Löschen?
 - falls letztes Element: einfach, ArrayList verkürzen!
 - sonst: Elemente hinter dem gelöschten Element müssen "nach vorne" kopiert werden
 - sehr ineffizient bei vielen Elementen

1	0	6	2	4
---	---	---	---	---

zu löschen!

1) nach vorne
kopieren

1	6	2	4	4
---	---	---	---	---

1	6	2	4	4
---	---	---	---	---

2) verkürzen

3) fertig

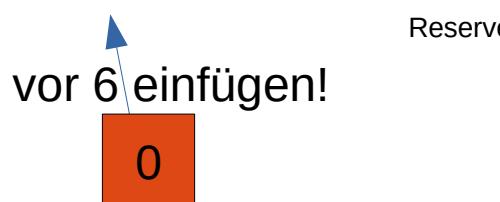
1	6	2	4
---	---	---	---



Collections: ArrayList

- Wie funktioniert Einfügen?
 - falls letztes Element: einfach, ArrayList verlängern!
 - sonst: Elemente hinter dem eingefügten Element müssen "nach hinten" kopiert werden
 - sehr ineffizient bei vielen Elementen
 - Annahme: Reserve vorhanden!

1	6	2	4	?
---	---	---	---	---



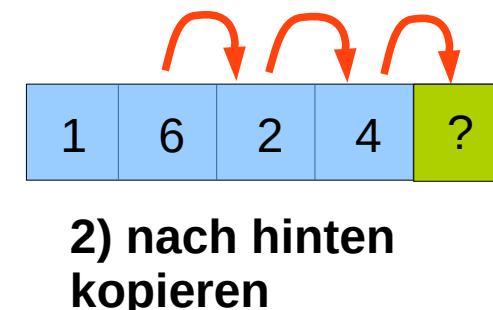
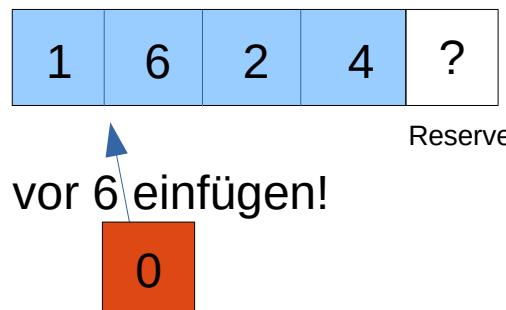
1	6	2	4	?
---	---	---	---	---

1) verlängern

Kap. 13.2

Collections: ArrayList

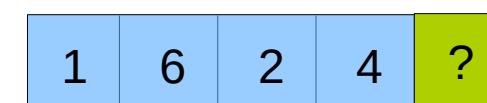
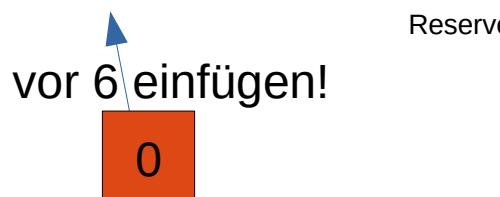
- Wie funktioniert Einfügen?
 - falls letztes Element: einfach, ArrayList verlängern!
 - sonst: Elemente hinter dem eingefügten Element müssen "nach hinten" kopiert werden
 - sehr ineffizient bei vielen Elementen
 - Annahme: Reserve vorhanden!



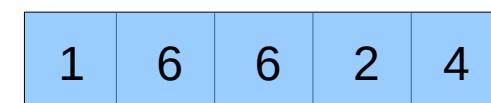


Collections: ArrayList

- Wie funktioniert Einfügen?
 - falls letztes Element: einfach, ArrayList verlängern!
 - sonst: Elemente hinter dem eingefügten Element müssen "nach hinten" kopiert werden
 - sehr ineffizient bei vielen Elementen
 - Annahme: Reserve vorhanden!



1) verlängern



2) nach hinten
kopieren

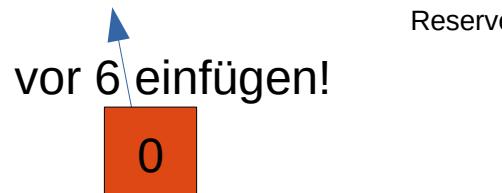
Seite 20



Collections: ArrayList

- Wie funktioniert Einfügen?
 - falls letztes Element: einfach, ArrayList verlängern!
 - sonst: Elemente hinter dem eingefügten Element müssen "nach hinten" kopiert werden
 - sehr ineffizient bei vielen Elementen
 - Annahme: Reserve vorhanden!

1	6	2	4	?
---	---	---	---	---



1	6	2	4	?
---	---	---	---	---

1) verlängern

1	6	6	2	4
---	---	---	---	---

2) nach hinten kopieren

1	0	6	2	4
---	---	---	---	---

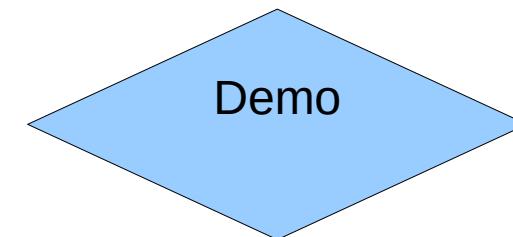
3) einfügen,
fertig

Seite 21



Collections: ArrayList

- Welche Methoden werden(u.a.) angeboten?
 - **get** Element mit bekanntem Index zurückliefern
 - **set** Element mit bekanntem Index setzen
 - **indexof** Elemente finden (erstes Auftreten)
 - **remove** Element mit bekanntem Index entfernen
 - **add** Element hinten anfügen
 - **add** Element an beliebiger Stelle einfügen
 - **size** Größe bestimmen
 - **clear** alles löschen





CUT: Q&A



Kap. 13.2

Collections: LinkedList

- Wann sollte eine `LinkedList` benutzt werden?
 - falls schnelles Finden per Index **nicht** wichtig ist (z.B. falls nur nach Werten gesucht wird: "ist Wert 5 in der `LinkedList` enthalten?")
 - falls oft eingefügt/gelöscht/angehängt werden soll
 - variable, nicht vorher bekannte Gesamtzahl von Elementen

Kap. 13.2

Collections: LinkedList

- Deklaration und Methoden ähnlich ArrayList

```
LinkedList<Integer> t = new LinkedList<Integer>() ;  
t.add(1) ; // anhängen  
t.add(2) ; // einfügen  
t.remove(0) ; // löschen  
System.out.println(t.get(0)) ;
```

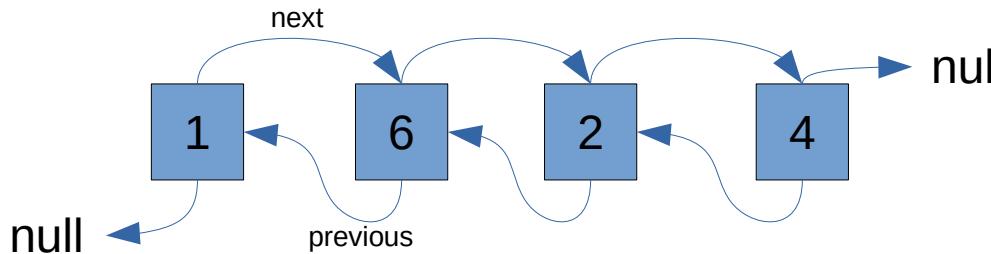
- ebenfalls generischer Datentyp



Kap. 13.2

Collections: LinkedList

- interne Darstellung der Daten: doppelt verkettete Liste
 - Elemente stehen irgendwo im Speicher (nicht nacheinander; nicht zusammenhängend)
 - Jeder Wert ist Teil eines Knotens: kennt vorheriges und nächstes Element

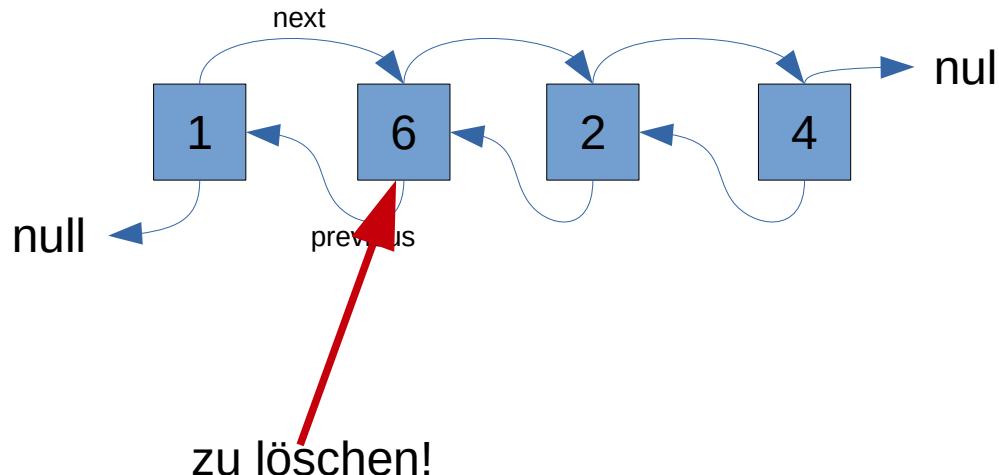


```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Kap. 13.2

Collections: LinkedList

- Wie funktioniert Löschen?
 - egal wo Knoten gelöscht wird: einfach, effizient!
 - Löschen: Verbiegen von previous und next

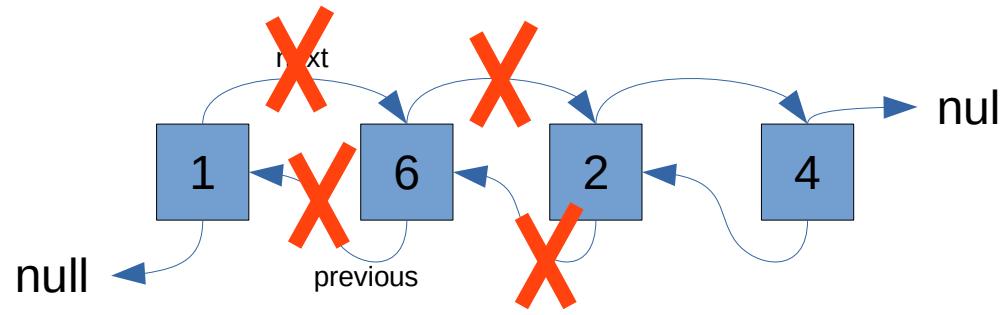


```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Kap. 13.2

Collections: LinkedList

- Wie funktioniert Löschen?
 - egal wo Knoten gelöscht wird: einfach, effizient!
 - Löschen: Verbiegen von previous und next



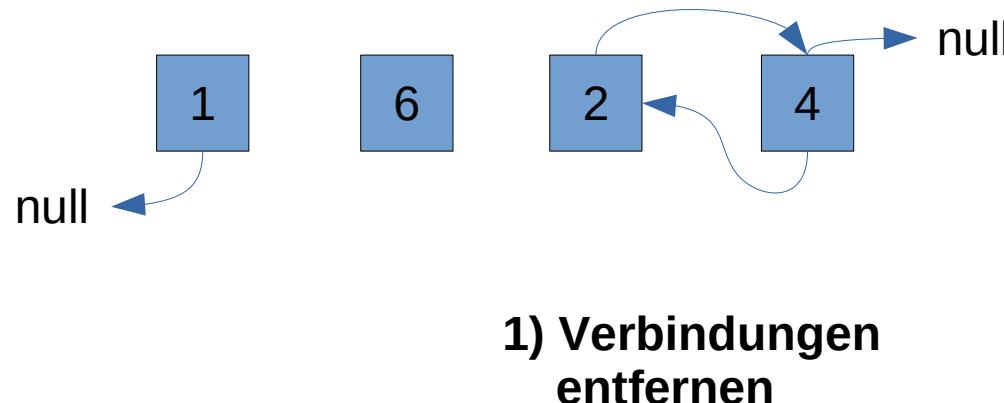
1) Verbindungen entfernen

```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Kap. 13.2

Collections: LinkedList

- Wie funktioniert Löschen?
 - egal wo Knoten gelöscht wird: einfach, effizient!
 - Löschen: Verbiegen von previous und next

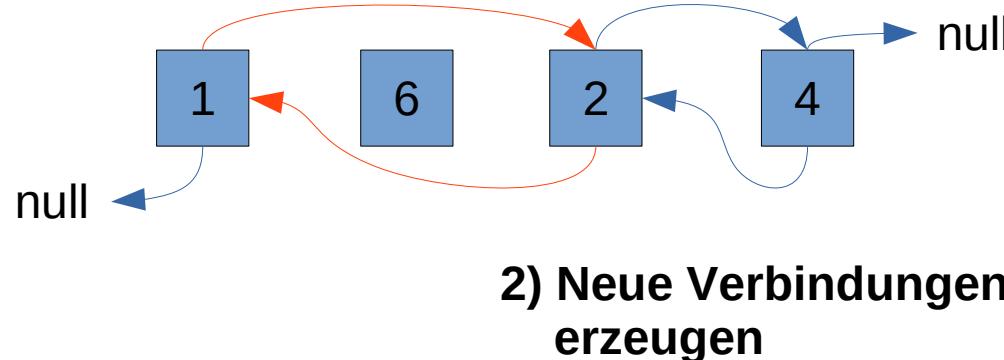


```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Collections: LinkedList

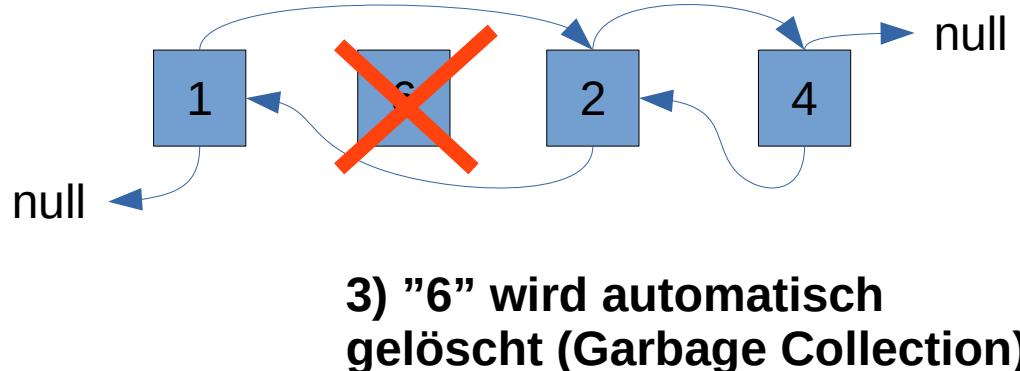
- Wie funktioniert Löschen?
 - egal wo Knoten gelöscht wird: einfach, effizient!
 - Löschen: Verbiegen von previous und next



```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Collections: LinkedList

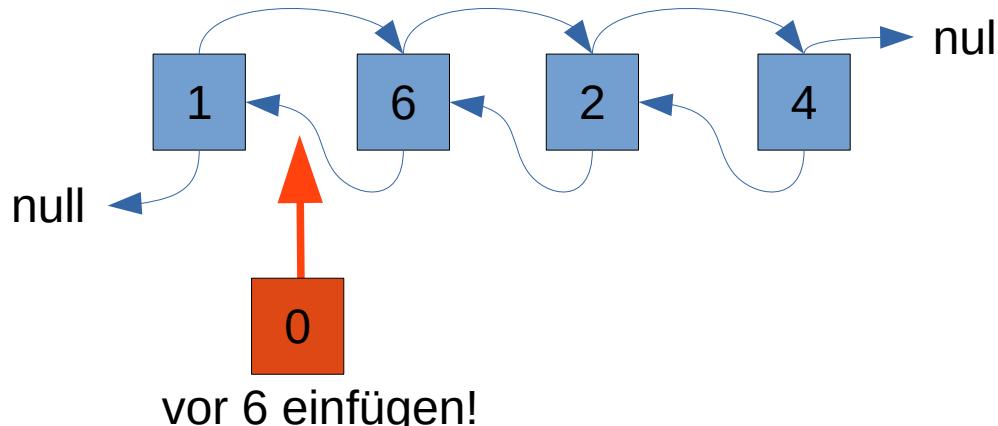
- Wie funktioniert Löschen?
 - egal wo Knoten gelöscht wird: einfach, effizient!
 - Löschen: Verbiegen von previous und next



```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Collections: LinkedList

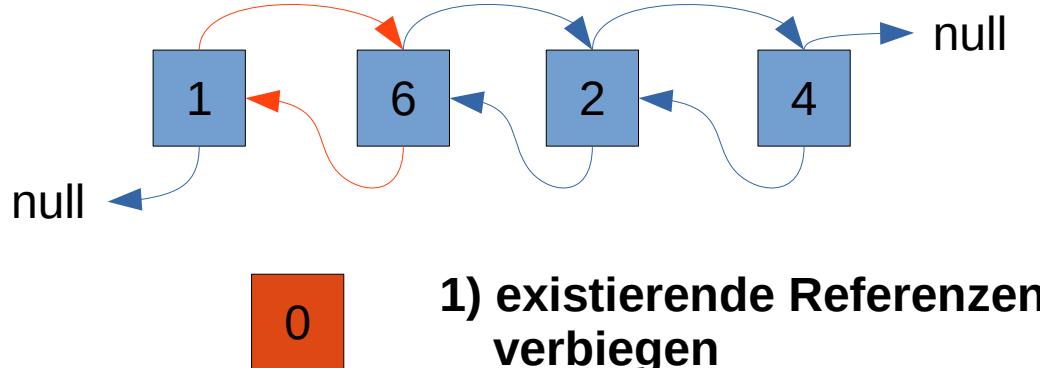
- Wie funktioniert Einfügen?
 - ähnlich wie beim Löschen: Verbiegen von Referenzen
 - ähnlich wie beim Löschen: einfach, effizient!



```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Collections: LinkedList

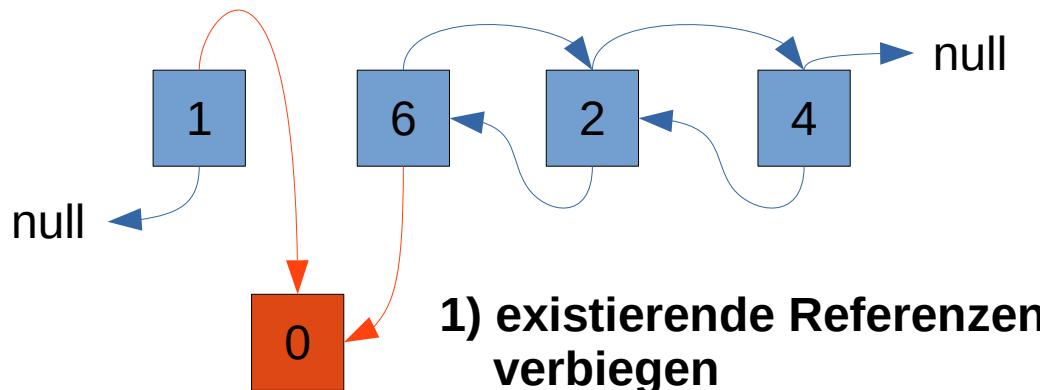
- Wie funktioniert Einfügen?
 - ähnlich wie beim Löschen: Verbiegen von Referenzen
 - ähnlich wie beim Löschen: einfach, effizient!



```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Collections: LinkedList

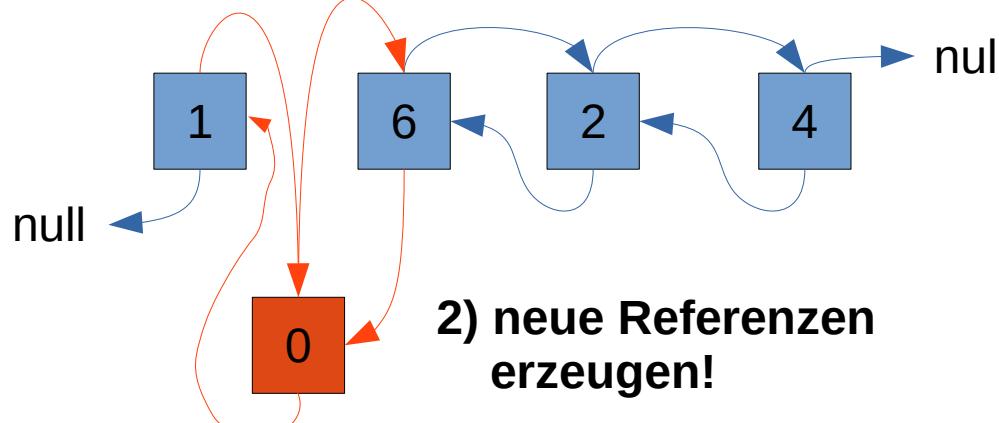
- Wie funktioniert Einfügen?
 - ähnlich wie beim Löschen: Verbiegen von Referenzen
 - ähnlich wie beim Löschen: einfach, effizient!



```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```

Collections: LinkedList

- Wie funktioniert Einfügen?
 - ähnlich wie beim Löschen: Verbiegen von Referenzen
 - ähnlich wie beim Löschen: einfach, effizient!

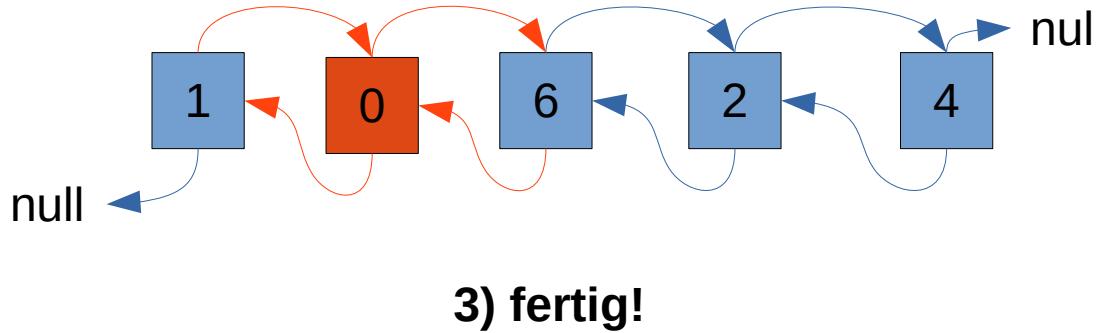


```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Collections: LinkedList

- Wie funktioniert Einfügen?
 - ähnlich wie beim Löschen: Verbiegen von Referenzen
 - ähnlich wie beim Löschen: einfach, effizient!



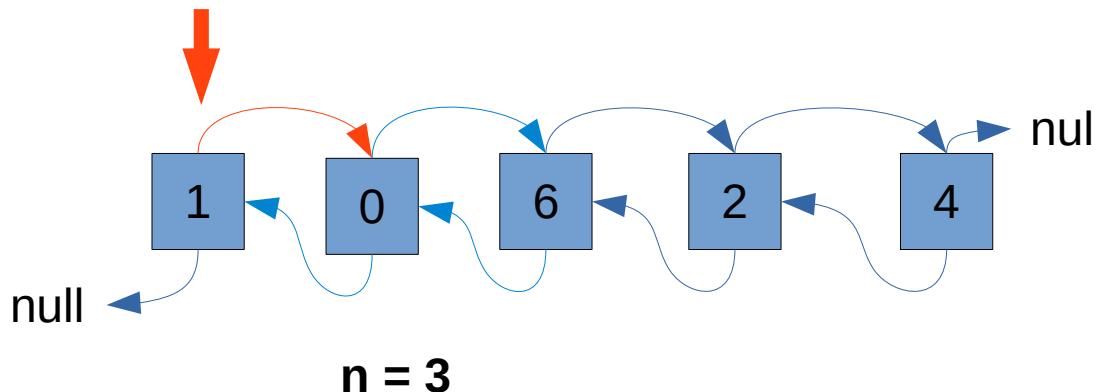
```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Kap. 13.2

Collections: LinkedList

- Wie funktioniert das Finden eines Elements per Index n?
 - starte beim Element 0
 - wiederhole n Mal: gehe zum nächsten Element
 - großes n → ineffizient!



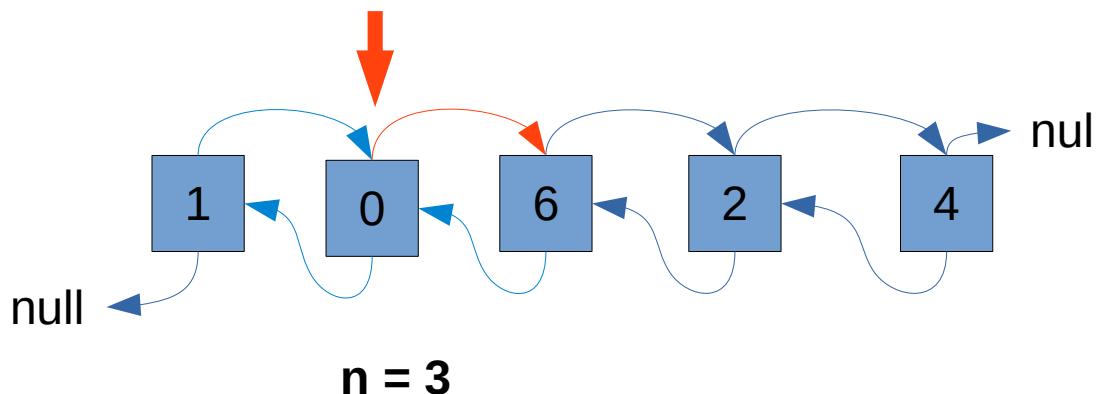
```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Kap. 13.2

Collections: LinkedList

- Wie funktioniert das Finden eines Elements per Index n?
 - starte beim Element 0
 - wiederhole n Mal: gehe zum nächsten Element
 - großes n → ineffizient!



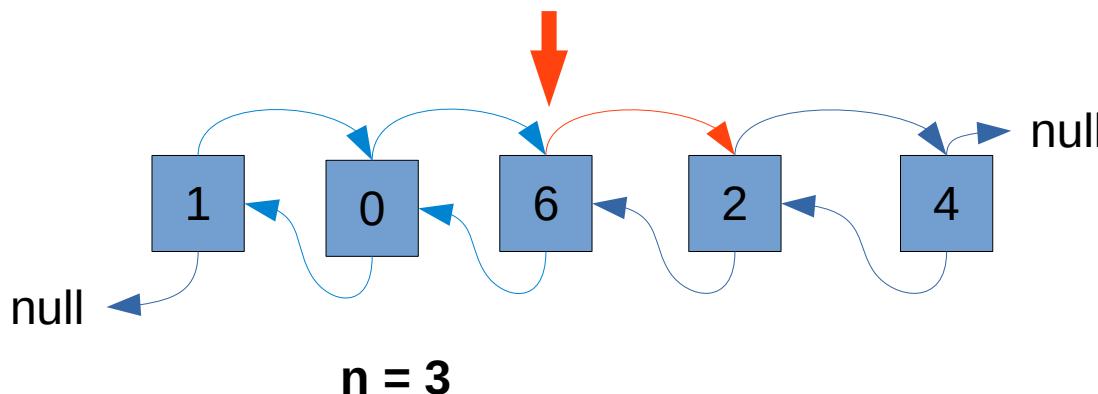
```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Kap. 13.2

Collections: LinkedList

- Wie funktioniert das Finden eines Elements per Index n?
 - starte beim Element 0
 - wiederhole n Mal: gehe zum nächsten Element
 - großes n → ineffizient!



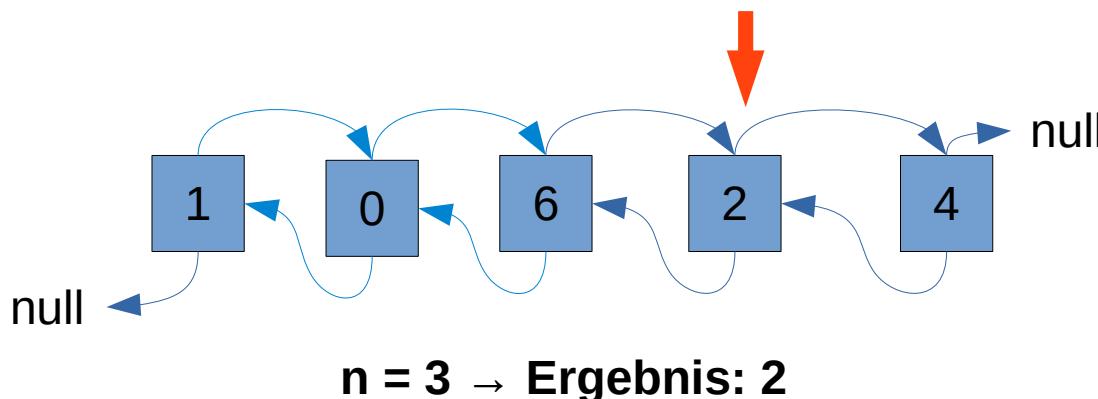
```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Kap. 13.2

Collections: LinkedList

- Wie funktioniert das Finden eines Elements per Index n?
 - starte beim Element 0
 - wiederhole n Mal: gehe zum nächsten Element
 - großes n → ineffizient!



```
class Node {  
    Node previous = null ;  
    Node next = null ;  
    Integer value ;  
}
```



Collections: LinkedList

- Welche Methoden werden(u.a.) angeboten?
 - **get** Element mit bekanntem Index zurückliefern
 - **set** Element mit bekanntem Index setzen
 - **indexOf** Elemente finden (erstes Auftreten)
 - **remove** Element mit bekanntem Index entfernen
 - **add** Element hinten anfügen
 - **add** Element an beliebiger Stelle einfügen
 - **size** Größe bestimmen
 - **clear** alles löschen

Demo



Generische Klassen

- Ziele: oberflächliches Verständnis generischer Klassen
- Vor allem: existierende generische Klassen benutzen (z.B. Collections)
- Einfache eigene generische Klassen verfassen



Generische Klassen nutzen

- Beispiel für generische Klassen: Collections

 **neues Java!!**

```
ArrayList<Integer> stuff = new ArrayList<Integer>();  
  
stuff.add( 42 ) ;
```

- generische Klasse hat einen oder mehrere
Typparameter
- Im Beispiel: Compiler weiß dadurch:
gespeicherte Objekte sind vom Typ Integer →
Prüfung möglich!

Kap. 9.1

Generische Klassen nutzen

- Beispiel für generische Klassen: Collections

 **neues Java!!**

```
ArrayList<Integer> stuff = new ArrayList<Integer>() ;  
stuff.add( 42 ) ;
```

- generische Klasse hat einen oder mehrere **Typparameter**
- Im Beispiel: Compiler weiß dadurch: gespeicherte Objekte sind vom Typ Integer → Prüfung möglich!



Generische Klassen definieren

- Definition generischer Klassen mit **Typparametern**

```
class GenericsExampleClass <T> {  
    private T attribute ;  
  
    public void setAttribute (T newValue) {  
        this.attribute = newValue ;  
    }  
  
    public T getAttribute() {  
        return this.attribute ;  
    }  
}
```

- Typparameter sind Platzhalter für beliebige Klasse



Generische Klassen definieren

- Definition generischer Klassen mit **Typparametern**

```
class GenericsExampleClass <T> {  
    private T attribute ;  
  
    public void setAttribute (T newValue) {  
        this.attribute = newValue ;  
    }  
  
    public T getAttribute() {  
        return this.attribute ;  
    }  
}
```

- Typparameter sind Platzhalter für beliebige Klasse
- nichts über Klasse bekannt



Instanzierung generischer Klassen

- **Typparameter** wird erst dann eingesetzt, wenn eine Instanz der Klasse erzeugt wird

```
GenClass<Integer> i = new GenClass<Integer>();
```

- Beliebig viele Instanzen mit anderen Typ-Parametern möglich

```
GenClass<Boolean> i = new GenClass<Boolean>();
```

- Typparameter ist Teil des Klassennamens!



Kap. 9.1

Instanzierung von Collections

- Alle Referenzdatentypen möglich:

```
LinkedList<Integer> inst1 = new LinkedList<Integer>() ;  
inst1.add(5) ;
```

- oder:

```
ArrayList<Boolean> inst2 = new ArrayList<Boolean>() ;  
inst2.add(true) ;
```

- oder:

```
ArrayList<Double> inst3 = new ArrayList<Double>() ;  
inst3.add(5.0) ;
```



Jetzt: Online-Doku verständlich!

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform
Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class LinkedList<E>

java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractList<E>
 java.util.AbstractSequentialList<E>
 java.util.LinkedList<E>

Type Parameters:
E - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently and at least one of the threads



Jetzt: Online-Doku verständlich!

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform
Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class LinkedList<E> Typparameter

java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractList<E>
 java.util.AbstractSequentialList<E>
 java.util.LinkedList<E>

Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently and at least one of the threads



Method Summary

Jetzt: Online-Doku verständlich!

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

Methods

Modifier and Type	Method and Description
boolean	<code>add(E e)</code> Appends the specified element to the end of this list.
void	<code>add(int index, E element)</code> Inserts the specified element at the specified position in this list.
boolean	<code>addAll(Collection<? extends E> c)</code> Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	<code>addAll(int index, Collection<? extends E> c)</code> Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	<code>addFirst(E e)</code> Inserts the specified element at the beginning of this list.
void	<code>addLast(E e)</code> Appends the specified element to the end of this list.
void	<code>clear()</code> Removes all of the elements from this list.
Object	<code>clone()</code> Returns a shallow copy of this LinkedList.
boolean	<code>contains(Object o)</code> Returns true if this list contains the specified element.
Iterator<E>	<code>descendingIterator()</code> Returns an iterator over the elements in this deque in reverse sequential order.
E	<code>element()</code> Retrieves, but does not remove, the head (first element) of this list.
E	<code>get(int index)</code> Returns the element at the specified position in this list.



Method Summary

Jetzt: Online-Doku verständlich!

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

Methods

Modifier and Type	Method and Description
boolean	add(E e) Appends the specified element to the end of this list.
void	add(int index, E element) Inserts the specified element at the specified position in this list.
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	addFirst(E e) Inserts the specified element at the beginning of this list.
void	addLast(E e) Appends the specified element to the end of this list.
void	clear() Removes all of the elements from this list.
Object	clone() Returns a shallow copy of this LinkedList.
boolean	contains(Object o) Returns true if this list contains the specified element.
Iterator<E>	descendingIterator() Returns an iterator over the elements in this deque in reverse sequential order.
E	element() Retrieves, but does not remove, the head (first element) of this list.
E	get(int index) Returns the element at the specified position in this list.



Sacken lassen...



Zusammenfassung

- Collection-Datentypen in Java: ArrayList, LinkedList (gibt noch viel mehr) und Vorteile über Arrays
- Collection-Datentypen sind **generische Klassen**
- Grund: bessere Prüfbarkeit durch den Compiler
- Kurzer Exkurs: wie funktioniert Definition und Instanzierung eigener generischer Klassen?



Ziele der Vorlesung

- Wissen dass es Collections gibt + Umgang mit den wichtigsten Typen
- Wissen dass es generische Klassen gibt
- Erkennen und passives Verstehen generischer Klassen
- Soweit daß Online-Dokumentation verstanden werden kann!



Programmierung 2

Vererbung III (static, Interfaces)

Alexander Gepperth, Mai 2022



static

- Bzw: Klassenmethoden und
Klassenattribute



static

- Bisher: alle Attribute und Methoden waren einer Instanz zugeordnet
 - Jede Instanz der selben Klasse hat unabhängige Attribute
 - Methoden werden über die Instanz aufgerufen und dürfen auf Attribute der eigenen Instanz zugreifen



Kap. 5.1, 5.4

static

- Analogie bisher: Klasse ist Stempel, Instanz ist Abdruck
 - gibt nur einen Stempel aber viele Abdrücke
 - Abdrücke können verschieden sein
(Attribute sind unterschiedlich)
 - Abdrücke sind unabhängig voneinander (da jeder eigenen Speicherbereich hat)





Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Es gibt Attribute und Methoden, die zur Klasse selbst gehören
- existieren bevor eine Instanz angelegt wurde (mit new)
- werden mit Schlüsselwort static deklariert





Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

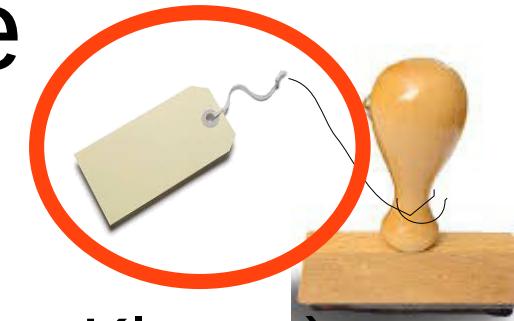
```
public class StaticDemo {  
  
    public static int staticAttr = 3 ;  
  
    public static int getStaticAttr(){  
        return StaticDemo.staticAttr ;  
    }  
  
    public static void setStaticAttr(int x) {  
        StaticDemo.staticAttr = x ;  
    }  
  
    public static void main(String[] args) {  
        /*Code */  
    }  
}
```



Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- static-Attribute
 - existieren vor Erstellung von Instanzen (bzw. ab Deklaration der Klasse)
 - "gehören zum Stempel" (es wird nur 1x Speicher reserviert egal wieviele Instanzen erzeugt werden)
 - best practice: immer Klassennamen voranstellen (StaticDemo.attribut1)

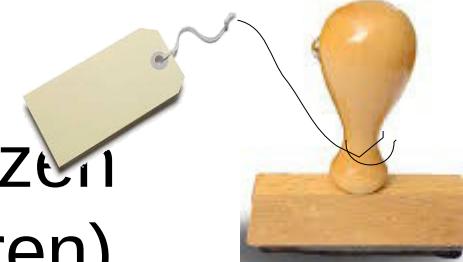




Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- static-Methoden existieren ohne Instanz
 - dürfen nicht `this` benutzen
 - dürfen nur static-Attribute benutzen
(die ebenfalls ohne Instanz existieren)
 - dürfen nur static-Methoden aufrufen
(sonst könnten normale Methoden ja `this` benutzen)
- aber: normale Methoden dürfen static-Methoden/Attribute nutzen

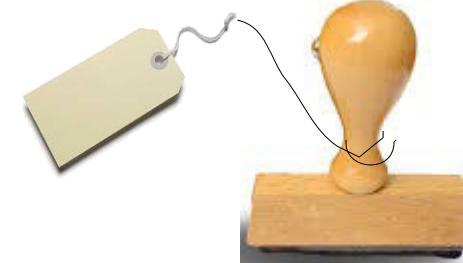




Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit static:
Sammlung von thematisch zusammengehörigen Methoden,
die keine Instanz benötigen
 - `Math.sin`, `Math.cos`, `Math.sqrt`
 - `Integer.valueOf`, `Integer.parseInt`

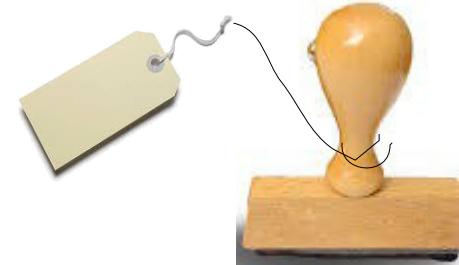




Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
global zugreifbare Konstanten
 - `Color.RED`, ...
 - `Math.PI`, ..
 - `SpaceInvadersLevel.EGOSPEED`

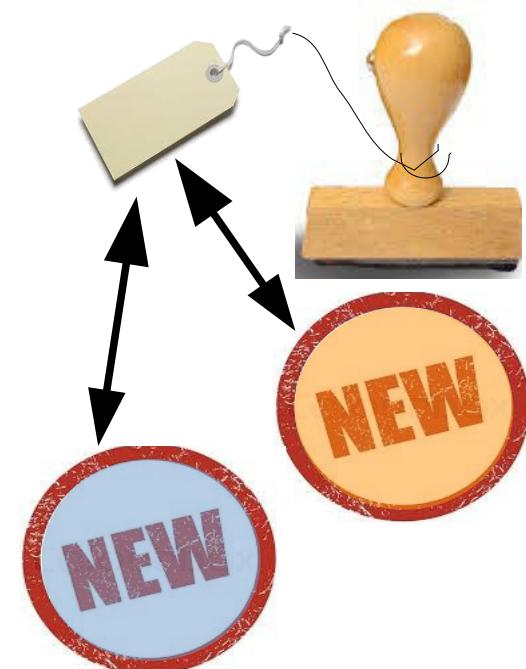




Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit static:
static-Attribut das private ist
 - kann von allen Instanzen einer Klasse (und nur von ihnen) benutzt werden
 - erlaubt allen Instanzen auf ein einzelnes, gemeinsames Attribut zuzugreifen → Kommunikation zwischen Instanzen





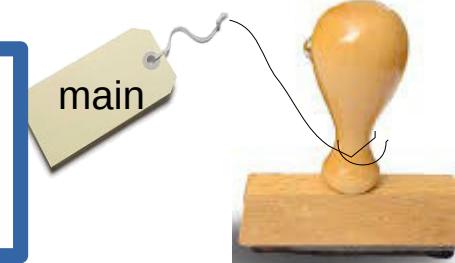
Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:

`main()`-Methode:

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



- Metode die von der JVM bei Programmstart automatisch aufgerufen wird (→ muss `public` sein)
- hat nichts mit Instanzen dieser Klasse zu tun → `static`
- `static`-Regeln gelten auch für `main()`

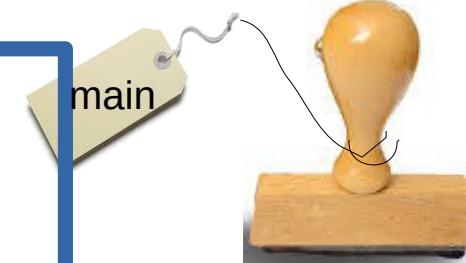


Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
`main()`-Methode:

```
class MyClass {  
  
    public int f(int x) {  
        return x ;  
    }  
  
    public static void main(String [] args) {  
        System.out.println(f(3)) ;  
    }  
}
```



OK?

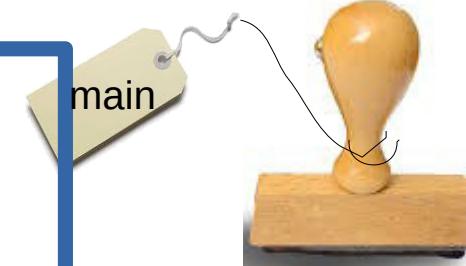


Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
`main()`-Methode:

```
class MyClass {  
  
    public int f(int x) {  
        return x ;  
    }  
  
    public static void main(String [] args) {  
        System.out.println(f(3)) ;  
    }  
}
```



nein weil f
nicht static
ist!

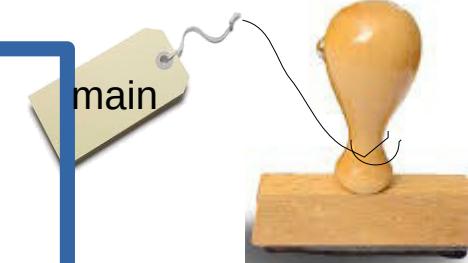


Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
`main()`-Methode:

```
class MyClass {  
  
    public int x = 3;  
  
    public static void main(String [] args) {  
        System.out.println(x) ;  
    }  
}
```



OK?

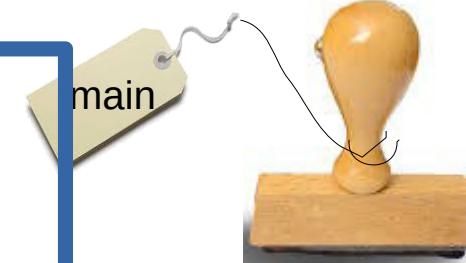


Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
`main()`-Methode:

```
class MyClass {  
  
    public int x = 3;  
  
    public static void main(String [] args) {  
        System.out.println(x) ;  
    }  
}
```



nein weil x
nicht static
ist!

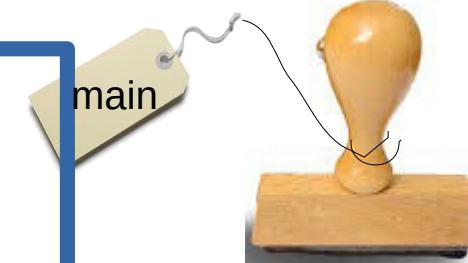


Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
`main()`-Methode:

```
class MyClass {  
  
    public int x = 3;  
  
    public static void main(String [] args) {  
        MyClass test = new MyClass() ;  
        System.out.println(test.x) ;  
    }  
}
```



OK?

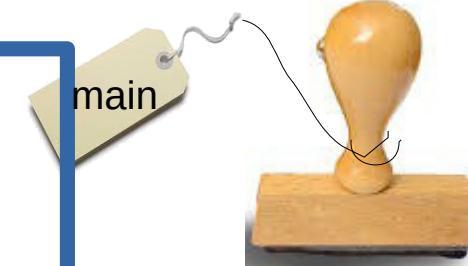


Kap. 5.3

OOP: Klassenmethoden und Klassenattribute

- Typische Konstruktionen mit `static`:
`main()`-Methode:

```
class MyClass {  
  
    public int x = 3;  
  
    public static void main(String [] args) {  
        MyClass test = new MyClass() ;  
        System.out.println(test.x) ;  
    }  
}
```



ja `test.x` ist eindeutig der Instanz `test` zugeordnet



CUT: Q&A



Kap. 5.13

Interfaces

- Abstrakte Klassen haben eine Doppelrolle:
 - Definition einer Schnittstelle für alle abgeleiteten Klassen
 - Auslagerung von gemeinsamen Funktionalitäten in abstrakte Basisklasse
- Interfaces haben nur ein Ziel: Definition von Schnittstellen!



Kap. 5.13

Interfaces

- Spezielles Element der Java-Sprache
- Wie abstrakte Klasse, bis auf
 - **alle** Methoden sind `public`
 - **alle** Methoden sind automatisch `abstract`
 - keine eigenen Instanzattribute erlaubt
 - kein Konstruktor erlaubt



Kap. 5.13

Interfaces definieren

- Beispiel:

```
interface Unusable {  
    int uselessMethod1() ;  
    void uselessMethod2() ;  
}
```

- alles ist implizit public
- kein Code, alles automatisch abstract
- keine Attribute!



Kap. 5.13

Interfaces benutzen

- Beispiel: Klasse **implementiert** Interface

```
class UselessClass implements Unusable {  
    public int uselessMethod1() {  
    }  
    public void uselessMethod2() {  
    }  
}
```

- neues Schlüsselwort `implements`, funktioniert wie `extends`
- Unterklasse muss alle Methoden überschreiben



Kap. 5.13

Bewertung von Interfaces

- Interfaces “tun nichts”:
 - definieren keine Attribute
 - implementieren keinen Code
- Wozu sind sie gut?



Kap. 5.13

Bewertung von Interfaces

- Interfaces “tun nichts”:
 - definieren keine Attribute
 - implementieren keinen Code
- Wozu sind sie gut?
 - Ausschließlich zur Definition von Schnittstellen
 - Unterklasse **muss** alle Methoden überschreiben

```
interface Unusable {  
    int uselessMethod1() ;  
    void uselessMethod2() ;  
}
```



Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

```
interface Schnittstelle {  
    public void machWas() ;  
}  
  
class Unterklasse implements Schnittstelle {  
    @Override  
    public void machWas() {  
        System.out.println("Was gemacht!") ;  
    }  
}
```



Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

```
interface Schnittstelle {  
    public void machWas() ;  
}  
  
class Unterklasse implements Schnittstelle {  
    @Override  
    public void machWas() {  
        System.out.println("Was gemacht!") ;  
    }  
}  
  
Unterklassereferenz = new Unterklasse();  
Schnittstellerefenz = new Unterklasse();
```





Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

```
interface Schnittstelle {  
    public void machWas() ;  
}  
  
class Unterklasse implements Schnittstelle {  
    @Override  
    public void machWas() {  
        System.out.println("Was gemacht!") ;  
    }  
}  
  
Unterklassereferenz = new Unterklasse();  
Schnittstellerefenz = new Unterklasse();
```





Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

```
interface Schnittstelle {  
    public void machWas() ;  
}  
  
class Unterklasse implements Schnittstelle {  
    @Override  
    public void machWas() {  
        System.out.println("Was gemacht!") ;  
    }  
}  
  
Unterklassereferenz = new Unterklasse();  
Schnittstelleireferenz = new Unterklasse();  
ireferenz.machWas();
```





Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

```
interface Schnittstelle {  
    public void machWas() ;  
}
```

```
class Unterklasse implements Schnittstelle {  
    @Override  
    public void machWas() {  
        System.out.println("Was gemacht!") ;  
    }  
}
```

```
Unterklassereferenz = new Unterklasse() ;  
Schnittstelleireferenz = new Unterklasse() ;  
ireferenz.machWas() ;
```



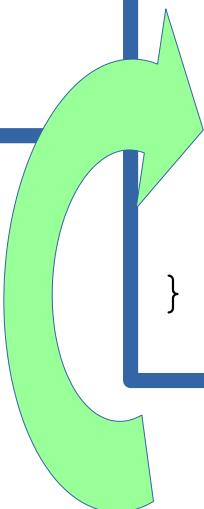


Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

```
interface Schnittstelle {  
    public void machWas() ;  
}
```

```
class Unterklasse implements Schnittstelle {  
    @Override  
    public void machWas() { wird aufgerufen, Polymorphie  
        System.out.println("Was gemacht!") ;  
    }  
}
```

```
Unterklassereferenz = new Unterklasse();  
Schnittstelleireferenz = new Unterklasse();  
ireferenz.machWas();
```





Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

- Fazit: Interfaces und abstrakte Klassen sind "normale" Klassen
- Typsubstitution und Polymorphie gelten wie bei "normalen" Klassen



Interfaces, abstrakte Klassen, Typsubstitution & Polymorphie

- Fazit: Interfaces und abstrakte Klassen sind "normale" Klassen
- Typsubstitution und Polymorphie gelten wie bei "normalen" Klassen
- Fazit des Fazits: nichts Neues!



CUT: Q&A



Programmierung 2

Iteration

Alexander Gepperth, Juni 2022



Iteration



Kap. 13.5

Iteration

- Ziel: Alle Werte in einer Collection ausdrucken!
- ArrayList: effizient? einfach?

```
ArrayList<Integer> x = new ArrayList<Integer>() ;  
  
// ArrayList füllen  
  
for (int i = 0; i < x.size(); i++) {  
    Integer elementI = x.get(i) ;  
    System.out.println(elementI) ;  
}
```



Kap. 13.5

Iteration

- Ziel: Alle Werte in einer Collection ausdrucken!
- einfach&effizient für ArrayList:

```
ArrayList<Integer> x = new ArrayList<Integer>() ;  
  
// ArrayList füllen  
  
for (int i = 0; i < x.size(); i++) {  
    Integer elementI = x.get(i) ;  
    System.out.println(elementI) ;  
}
```



Iteration

- Ziel: Alle Werte in einer Collection ausdrucken!
- **LinkedList**: effizient? einfach?

```
LinkedList<Integer> x = new LinkedList<Integer>() ;  
  
// LinkedList füllen  
  
for (int i = 0; i < x.size(); i++) {  
    Integer elementI = x.get(i) ;  
    System.out.println(elementI) ;  
}
```



Kap. 13.5

Iteration

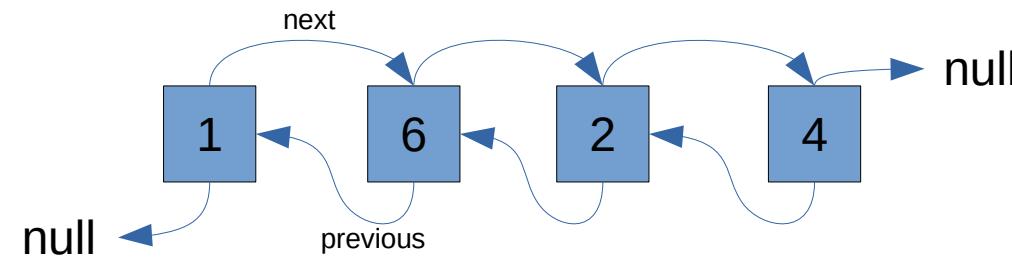
- Ziel: Alle Werte in einer Collection ausdrucken!
- einfach & sehr **ineffizient** für **LinkedList**:

```
LinkedList<Integer> x = new LinkedList<Integer>() ;  
  
// LinkedList füllen  
  
for (int i = 0; i < x.size(); i++) {  
    Integer elementI = x.get(i) ;  
    System.out.println(elementI) ;  
}
```

Kap. 13.5

Iteration

- Ziel: Alle Werte in einer Collection ausdrucken!
- einfach & sehr **ineffizient** für **LinkedList**:
- Grund: `.get (i)` sehr ineffizient für **LinkedList**
 - besser wäre: Sprung von einem Element zum nächsten über die `next`-Referenz
 - gibt aber keine `public`-Methoden die das können





Kap. 13.5

Iteration

- Diskussion des Beispiels:
 - beim einfachen **Durchlaufen** aller Elemente ist Zugriff per Index gar nicht von Vorteil
 - alles was man braucht ist eine Art "nächstes Element" - Methode
 - **gibt es!**

```
ArrayList<Integer> x = new ArrayList<Integer>() ;  
  
// ArrayList füllen  
  
for (int i = 0; i < x.size(); i++) {  
    Integer elementI = x.get(i) ;  
    System.out.println(elementI) ;  
}
```



Iteration

- Wünschenswert: Funktionalität zum Durchlaufen aller Elemente eines Collection-Datentyps
 - einheitlich: Durchlaufen funktioniert immer gleich
 - unabhängig: von der internen Implementierung des Datentyps und seinen Eigenschaften
 - sortiert:
 - sicher: Benachrichtigung sobald Durchlauf fertig ist



Kap. 13.5

Iteration

- In Java: Klassen, die das Interface `Iterable<E>` implementieren, können auf einheitliche Weise durchlaufen werden
- `Iterable<E>` schreibt nur eine einzige Methode vor: `iterator()`

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

- `iterator()` liefert Referenz auf ein Objekt, den sog. *Iterator* vom Typ `Iterator<E>`



Iteration

- Vorteile:
 - iterierbare Klassen können mit stets gleichem Code alle ihre Elemente durchlaufen lassen
 - jede iterierbare Klasse kann eine eigene, effiziente Implementierung von `Iterator<E>` bereitstellen
- Methoden von `Iterator<E>` zum Durchlaufen :
 - `boolean hasNext()` ;
 - `E next()` ;
 - `void remove()` ;



Kap. 13.5

Programmierung

- Sei `x` Instanz einer Klasse, die `Iterable <T>` implementiert
 - `ArrayList<Float> x = new ArrayList<Float> () ;
x.add(5.0) ;`
- `x.iterator()` liefert Unterklasse von `Iterator<Float>` zurück
 - `Iterator<Float> it = x.iterator() ;`
- der **Iterator** `it` "kennt" `x` und "weiß", wie er `x` effizient durchlaufen kann
 - `if(it.hasNext ()) {
System.out.println(it.next ()) ;
}`



Iteration

- Das geht auch mit for-Schleifen!

```
ArrayList<Integer> x = new ArrayList<Integer>() ;  
  
// ArrayList füllen  
  
for (Iterator<Integer> it = x.iterator(); it.hasNext() == true;) {  
    Integer elementI = it.next() ;  
    System.out.println(elementI) ;  
}
```

Kap. 13.5

Iteration

- Beispiel: Iteration über ArrayList:

```
ArrayList<Integer> x = new ArrayList<Integer>();  
  
// ArrayList füllen  
  
for (Iterator<Integer> it = x.iterator(); it.hasNext() == true;) {  
    Integer elementI = it.next();  
    System.out.println(elementI);  
}
```

it liefert nächstes Element

liefert Instanz einer Klasse zurück, die Iterator<Integer> implementiert

kann auf Daten in x zugreifen



Iteration

- Beispiel: Iteration über **LinkedList**:

```
LinkedList<Integer> x = new LinkedList<Integer>() ;  
  
// LinkedList füllen  
  
for (Iterator<Integer> it = x.iterator(); it.hasNext() == true;) {  
    Integer elementI = it.next() ;  
    System.out.println(elementI) ;  
}
```

it liefert nächstes Element

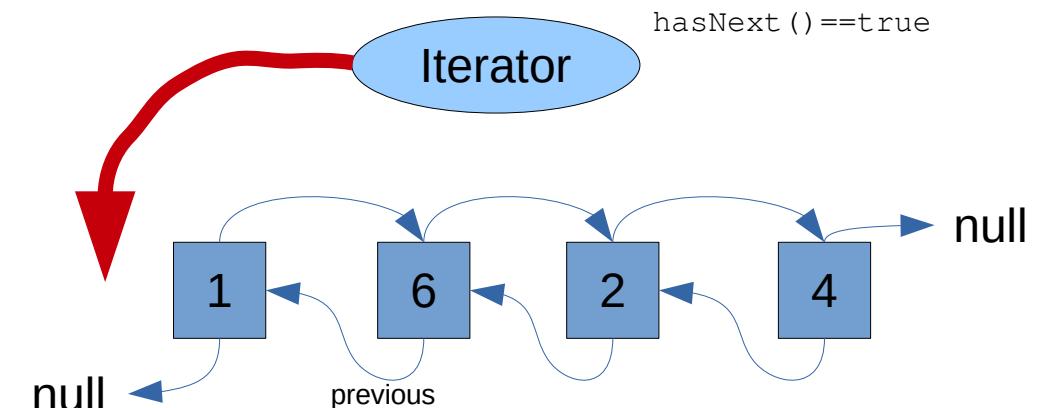
liefert Instanz einer Klasse zurück, die **Iterator<Integer>** implementiert

kann auf Daten in x zugreifen

Kap. 13.5

Iteration

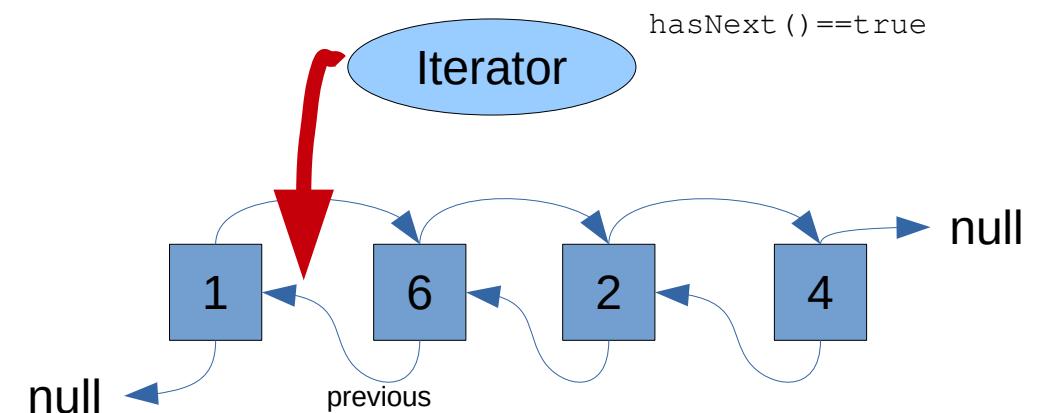
- Kommentare zu Iteratoren:
 - Zugriff über Iteratoren erfolgt immer sequentiell durch next()
 - jeder Iterator hat einen internen Zeiger auf seine Position in der Collection (am Anfang "vor dem ersten Element")



Kap. 13.5

Iteration

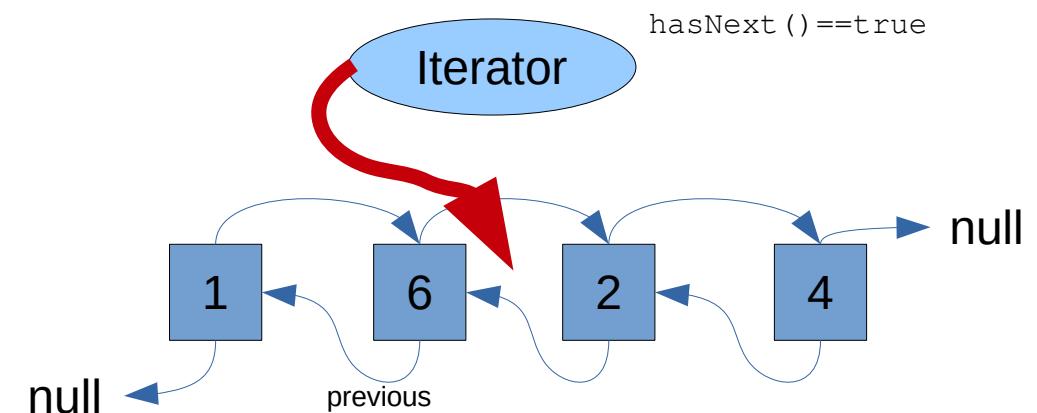
- Kommentare zu Iteratoren:
 - Zugriff über Iteratoren erfolgt immer sequentiell durch next()
 - jeder Iterator hat einen internen Zeiger auf seine Position in der Collection (am Anfang "vor dem ersten Element")
 - next() → Zeiger rückt ein Element weiter
 - Vorsicht mit next()



Kap. 13.5

Iteration

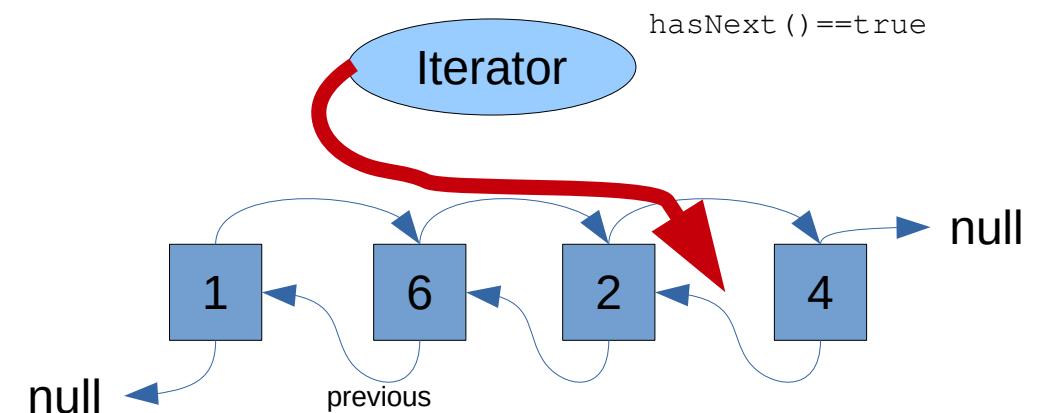
- Kommentare zu Iteratoren:
 - Zugriff über Iteratoren erfolgt immer sequentiell durch next()
 - jeder Iterator hat einen internen Zeiger auf seine Position in der Collection (am Anfang "vor dem ersten Element")
 - next() → Zeiger rückt ein Element weiter
 - Vorsicht mit next()



Kap. 13.5

Iteration

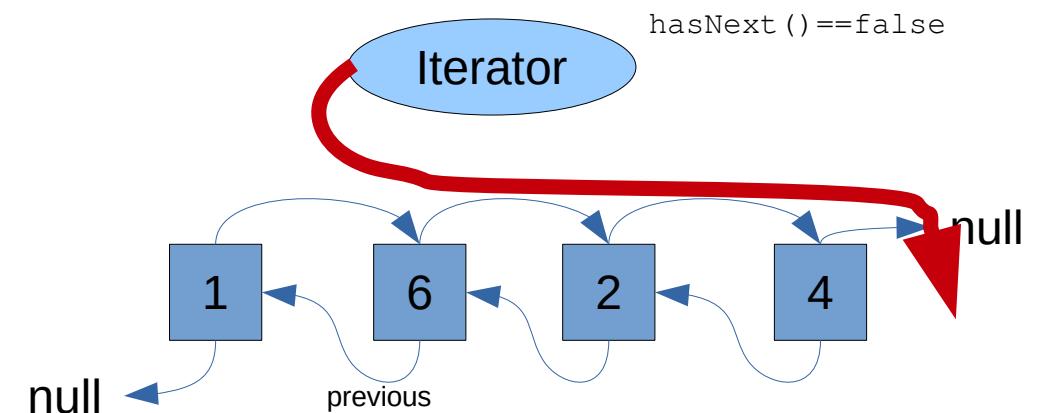
- Kommentare zu Iteratoren:
 - Zugriff über Iteratoren erfolgt immer sequentiell durch next()
 - jeder Iterator hat einen internen Zeiger auf seine Position in der Collection (am Anfang "vor dem ersten Element")
 - next() → Zeiger rückt ein Element weiter
 - Vorsicht mit next()



Kap. 13.5

Iteration

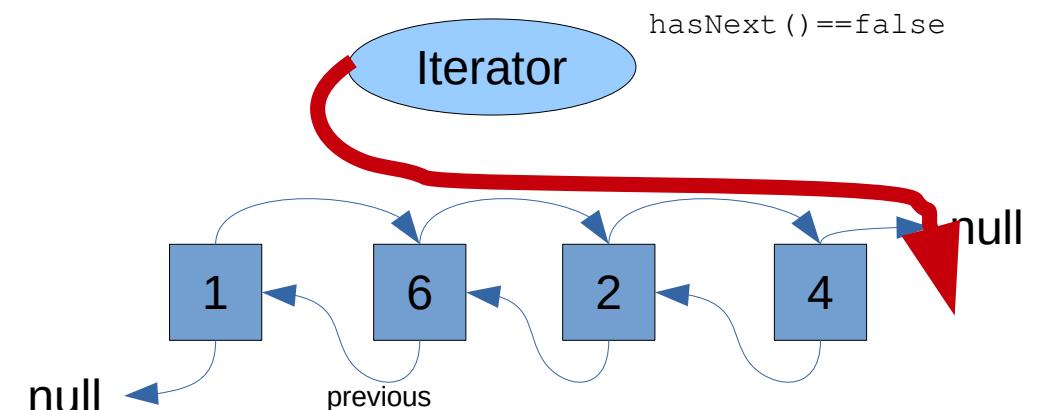
- Kommentare zu Iteratoren:
 - Zugriff über Iteratoren erfolgt immer sequentiell durch next()
 - jeder Iterator hat einen internen Zeiger auf seine Position in der Collection (am Anfang "vor dem ersten Element")
 - next() → Zeiger rückt ein Element weiter
 - Vorsicht mit next()



Kap. 13.5

Iteration

- Kommentare zu Iteratoren:
 - Zugriff über Iteratoren erfolgt immer sequentiell durch next()
 - jeder Iterator hat einen internen Zeiger auf seine Position in der Collection (am Anfang "vor dem ersten Element")
 - next() → Zeiger rückt ein Element weiter
 - Vorsicht mit next()
 - wenn hasNext()==false
 - Iterator ist "verbraucht"
 - neuer Durchlauf
 - neuer Iterator





Zusammenfassung

- Wichtige Datentypen und Konzepte:
 - Konzept der Iteration über Collections, in allen modernen Sprachen ebenfalls vorhanden
 - basiert in Java auf generischen Klassen!



Programmierung 2

Exceptions

Alexander Gepperth, Juni 2022



Kap. 6

Heute

- Beschäftigung mit **Laufzeitfehlern**
 - Programm compiliert korrekt
 - aber nach der Ausführung wird es durch einen Fehler abgebrochen



Heute

- Beschäftigung mit **Laufzeitfehlern**
 - Programm compiliert korrekt
 - aber nach der Ausführung wird es durch einen Fehler abgebrochen



```
Exception in thread "main"  
java.lang.NullPointerException  
at Error.main(error.java:6)
```



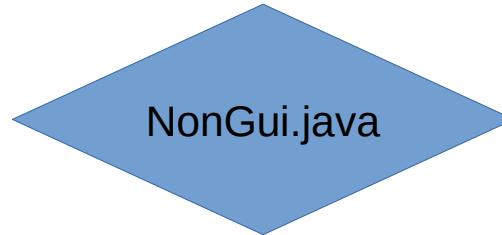


Demo: typische Laufzeitfehler



Problemstellung: Fehler zurückmelden

- Siehe Code-File: NonGui.java





Problemstellung: Fehler zurückmelden

NonGui.java

- Betrachte Methode aus NonGui.java:

```
static  
double firstValBelowX(ArrayList<Double> list, Double x)
```

- Ziele:

- finde erstes Element im Array, das kleiner ist als x
 - Fehler falls es solch ein Element nicht gibt

- Beispiele (für x = 5):

```
6 7 13 28 1 0 → ?  
15 0 3 8 -10 1 → ?  
7 8 20 → ?
```



Problemstellung: Fehler zurückmelden

NonGui.java

- Betrachte Methode aus NonGui.java:

```
static  
double firstValBelowX(ArrayList<Double> list, Double x)
```

- Ziele:
 - finde erstes Element im Array, das kleiner ist als x
 - Fehler falls es solch ein Element nicht gibt
- Beispiele (für x = 5):

```
6 7 13 28 1 0 → 1  
15 0 3 8 -10 1 → 0  
7 8 20 → FEHLER!
```



Problemstellung: Fehler zurückmelden

- Wie kann der Fehler signalisiert werden?

NonGui.java



Problemstellung: Fehler zurückmelden

NonGui.java

- Wie kann der Fehler signalisiert werden?
 - spezieller Rückgabewert, z.B. -1



Problemstellung: Fehler zurückmelden

NonGui.java

- Wie kann der Fehler signalisiert werden?
 - spezieller Rückgabewert, z.B. -1
 - Fehlermeldung ausgeben!



Problemstellung: Fehler zurückmelden

NonGui.java

- Wie kann der Fehler signalisiert werden?
 - spezieller Rückgabewert, z.B. -1
 - Fehlermeldung ausgeben!
 - Programmabbruch mit Fehlermeldung durch System.exit()



Problemstellung: Fehler zurückmelden

- Alle Vorschläge sind problematisch:
 - 1) Rückgabe von speziellem Wert:
kann mit normalem Rückgabewert verwechselt werden
 - 2) Fehlermeldung:
kann trotzdem ignoriert werden
 - 3) Programm abbrechen mit `System.exit()`
über Programmabbruch sollte die aufrufende Methode entscheiden

NonGui.java



Problemstellung: Fehler zurückmelden

- Gute Fehlerbehandlung sollte also:
 - nicht mit `return` verwechselt werden
 - die aufrufende Methode entscheiden lassen, wie mit dem Fehler umzugehen ist
 - nicht ignoriert werden können
 - Java besitzt einen von `return` unabhängigen “**Fehlerkanal**”, der diesen Anforderungen gerecht wird!
- Unabhängig von `return`
- Behandlung delegieren
- Behandlung erzwingen



CUT: Q&A



Kap. 6.5

Fehler zurückmelden in Java

- Fehler über **Fehlerkanal** zurückgeben:
 - Schlüsselwort `throw`
 - gefolgt von einer Instanz von `Throwable` (oder Unterklasse)
 - Beispiel: `throw new Exception("Fehler!")`
 - Instanz sollte den Fehler so genau wie möglich eingrenzen (eigene möglich!)
 - Verlässt die aktuelle Methode sofort!



Kap. 6.5

Deklaration von Exceptions

- Selbst verursachte Exceptions müssen (fast immer) deklariert werden

```
static
double firstValBelowX(ArrayList<Double> list,
    Double x) throws NoSmallerElementException {
```

- throws XXX im Methodenkopf bedeutet:
throw new XXX kann auftreten!
- Bezug zu @throws in JavaDoc, gefolgt von Typ
der Exception und unter welchen Umständen
sie auftritt!

NonGui.java



Kap. 6.5

Warum Deklaration von Exceptions?

- Der Compiler besteht (fast immer) auf der Deklaration einer Exception, falls eine Methode
 - einen `throw`-Befehl enthält
 - oder eine andere Methode aufruft, die eine Exception deklariert (mit `throws` im Methodenkopf)
- Warum müssen Exceptions deklariert werden?
 - damit der Compiler prüfen kann, ob sie **behandelt** werden!



CUT: Q&A



Kap. 6.1

Behandlung von Exceptions



Kap. 6.1

Behandlung von Exceptions

- Behandlung bedeutet:
 - Im Fehlerfall zurückgeliefertes Exception-Objekt empfangen
 - auswerten
 - entscheiden wie und ob das Programm weiterlaufen kann



Kap. 6.1

Behandlung von Exceptions

- try...catch...catch...finally...
 - im try-Block steht Code der eine Exception verursachen kann
 - catch-Blöcke **behandeln** Exceptions
 - der finally-Block wird auf jeden Fall ausgeführt!



Kap. 6.1

Beispiel: Exception wird behandelt

NonGui.java

```
public static void main(String[] args) {  
    double smallestVal = 0.0 ;  
    ArrayList<Double> list =  
        new ArrayList<Double>() ;  
    list.add (7.) ; list.add (8.) ; list.add(20.) ;  
try {  
    smallestVal = firstValBelowX(list, 5.) ;  
}  
catch (NoSmallerElementException e) {  
    System.out.println("Problem!!") ;  
}  
}
```



Kap. 6.1

Beispiel: Exception wird behandelt

NonGui.java

```
public static void main(String[] args) {  
    double smallestVal = 0.0 ;  
    ArrayList<Double> list =  
        new ArrayList<Double>() ;  
    list.add (7.) ; list.add (8.) ; list.add(20.) ;  
try {  
    smallestVal = firstValBelowX(list, 5.) ;  
}  
catch (NoSmallerElementException e) {  
    System.out.println(e.toString()) ;  
}  
}
```



Kap. 6.3

Behandlung von Exceptions

- catch-Blöcke überwachen den Rückgabekanal für Fehler auf Instanzen bestimmter Klassen
- **und ihrer Unterklassen!!**
- **d.h. catch(Throwable t) behandelt jeden beliebigen Fehler!**



Kap. 6.1

Das hätte also auch funktioniert!

```
public static void main(String[] args) {  
    double smallestVal = 0.0 ;  
    ArrayList<Double> list =  
        new ArrayList<Double>() ;  
    list.add (7.) ; list.add (8.) ; list.add(20.) ;  
try{  
    smallestVal = firstValBelowX(list, 5.) ;  
}  
catch (NoSmallerElementException e) {  
    System.out.println(e.toString()) ;  
}  
}
```

NonGui.java



Kap. 6.1

Das hätte also auch funktioniert!

```
public static void main(String[] args) {  
    double smallestVal = 0.0 ;  
    ArrayList<Double> list =  
        new ArrayList<Double>() ;  
    list.add (7.) ; list.add (8.) ; list.add(20.) ;  
try{  
    smallestVal = firstValBelowX(list, 5.) ;  
}  
catch (Throwable e) {  
    System.out.println(e.toString()) ;  
}  
}
```

NonGui.java



Kap. 6

Vergleich throw und return

return

verlässt **sofort** aktuelle Methode

gibt beliebige Werte zurück

zurückgebener Wert kann, muss aber nicht benutzt werden

Rückgabetyp muss deklariert sein

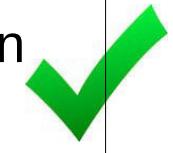
throw

verlässt **sofort** aktuelle Methode

gibt Unterklassen von Throwable zurück

zurückgegebene Exception muss behandelt werden (fast immer)

“geworfene” Exception muss deklariert sein (fast immer)





CUT: Q&A



Kap. 6.1

Exceptions & Programmabbruch

- Was passiert wenn eine Exception nicht behandelt wird??



Kap. 6.1

Exceptions & Programmabbruch

- Was passiert wenn eine Exception nicht behandelt wird??
 - Exception “steigt” durch die Aufrufhierarchie (“Call stack”) nach oben
 - Falls sie in `main` immer noch nicht behandelt wird
→ Programmabbruch



Kap. 6.1

Beispiel: String nach Zahlen konvertieren!

- `Integer.valueOf(String s, int radix)`
konvertiert einen String in eine Zahl
- Annahme: String enthält Zahl zur Basis `radix`
- Wie soll man hier einen Fehler signalisieren??
 - spezieller Wert?
 - Fehlermeldung?
 - Abbruch?

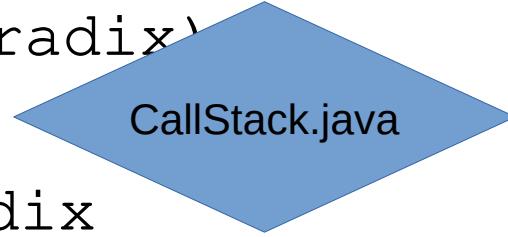
CallStack.java



Kap. 6.1

Beispiel: String nach Zahlen konvertieren!

- `Integer.valueOf(String s, int radix)`
konvertiert einen String in eine Zahl
- Annahme: String enthält Zahl zur Basis `radix`
- Wie soll man hier einen Fehler signalisieren??
 - spezieller Wert? **schlecht, kann verwechselt werden**
 - Fehlermeldung? **kann ignoriert werden**
 - Abbruch? **evtl zu krass!**



CallStack.java



Kap. 6.1

Beispiel: String nach Zahlen konvertieren!

- `Integer.valueOf(String s, int radix)` konvertiert einen String in eine Zahl
- Annahme: String enthält Zahl zur Basis `radix`
- Wie soll man hier einen Fehler signalisieren??
 - spezieller Wert? **schlecht, kann verwechselt werden**
 - Fehlermeldung? **kann ignoriert werden**
 - Abbruch? **evtl zu krass!**
- Fehler wird per `NumberFormatException` angezeigt!
- Was passiert ohne und mit Behandlung?

CallStack.java

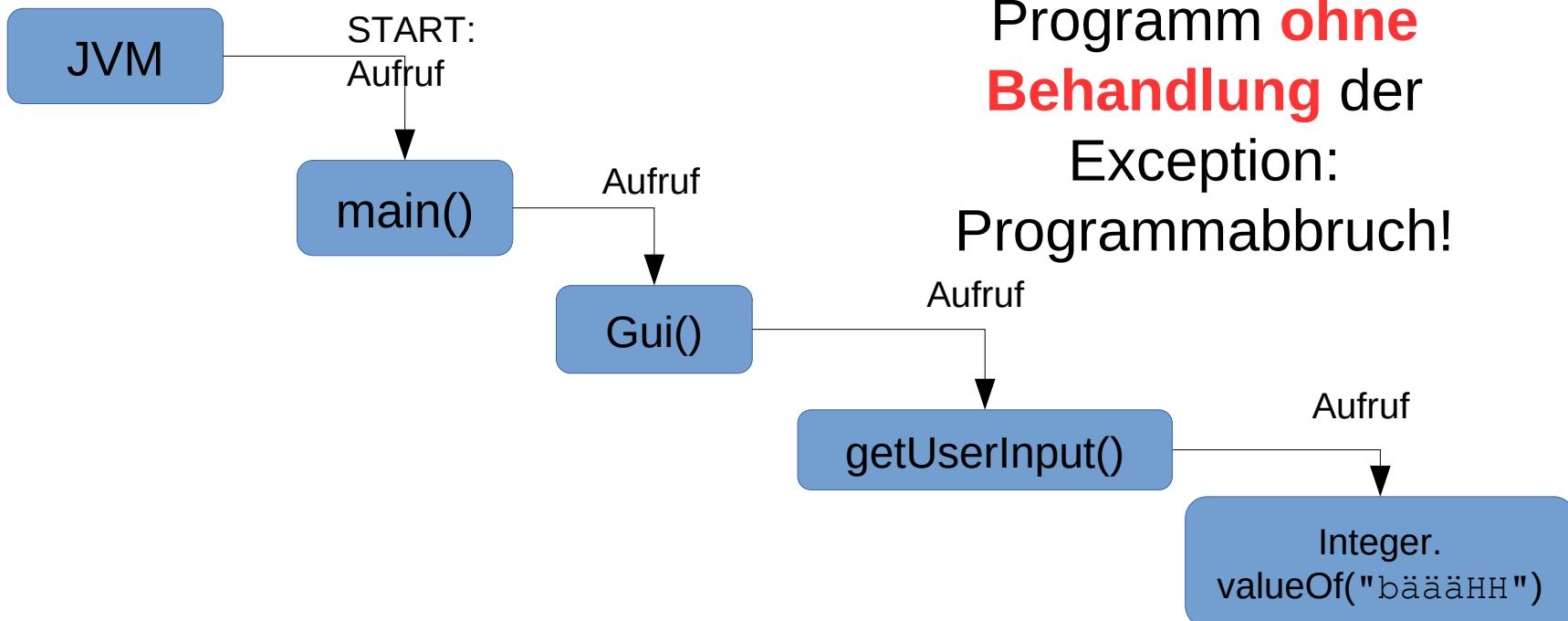


Kap. 6.1

Demo

CallStack.java

Integer.valueOf()
in einem komplexen
Programm **ohne**
Behandlung der
Exception:
Programmabbruch!





Kap. 6.1

Demo

CallStack.java

Integer.valueOf()
in einem komplexen
Programm **ohne**
Behandlung der
Exception:
Programmabbruch!

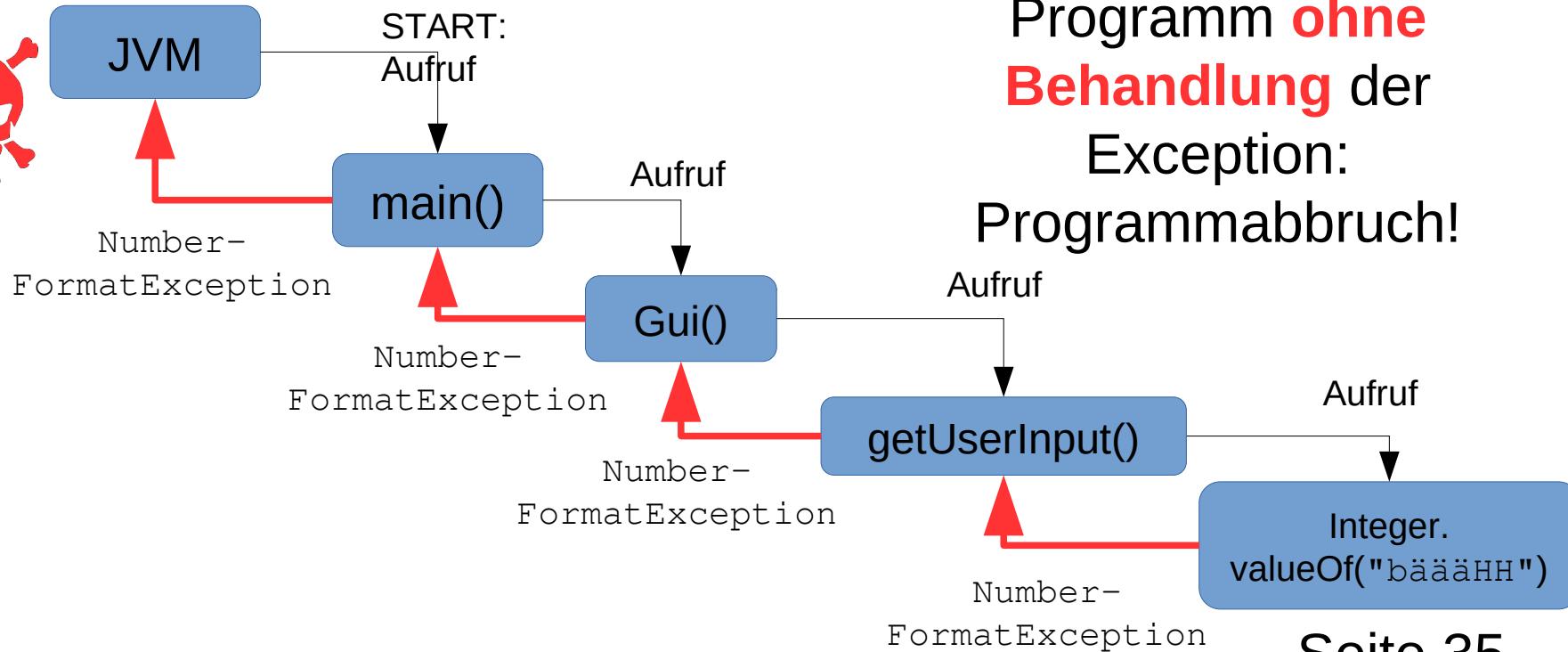
Aufruf

Aufruf

Aufruf

Aufruf

Seite 35



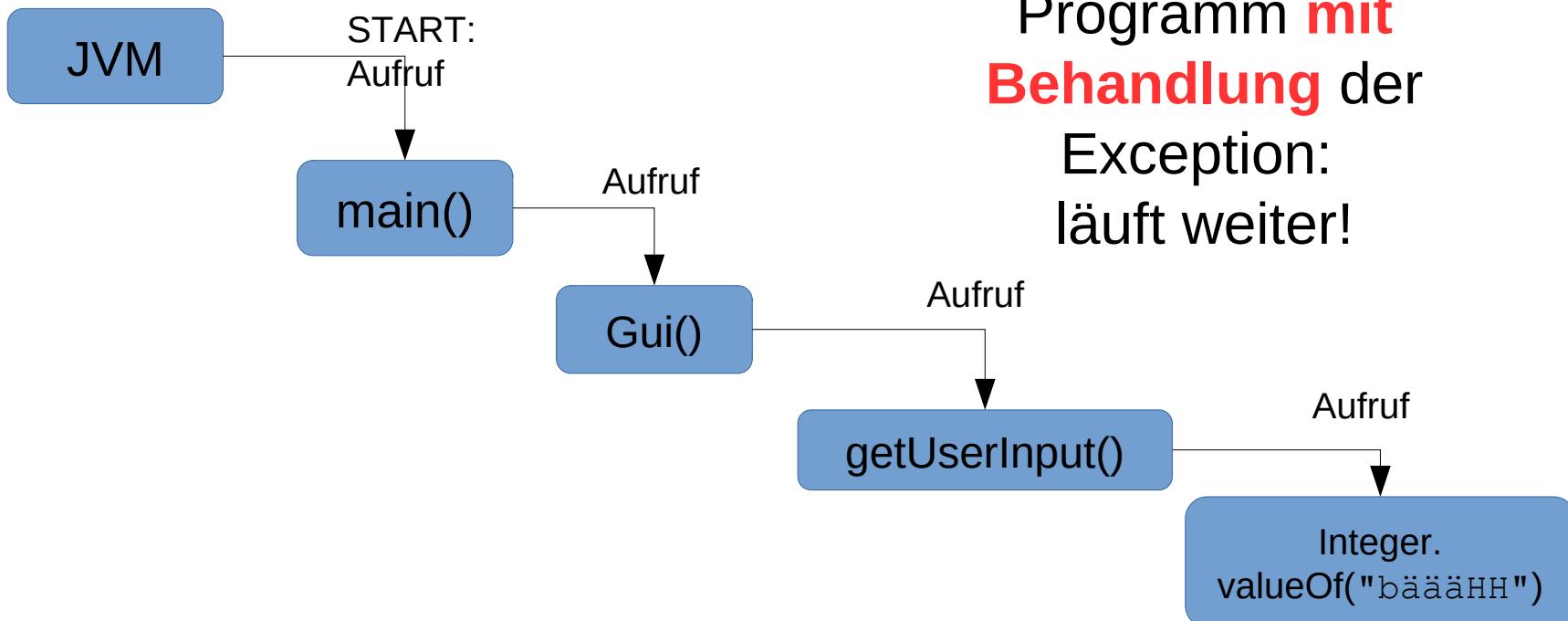


Kap. 6.1

Demo

CallStack.java

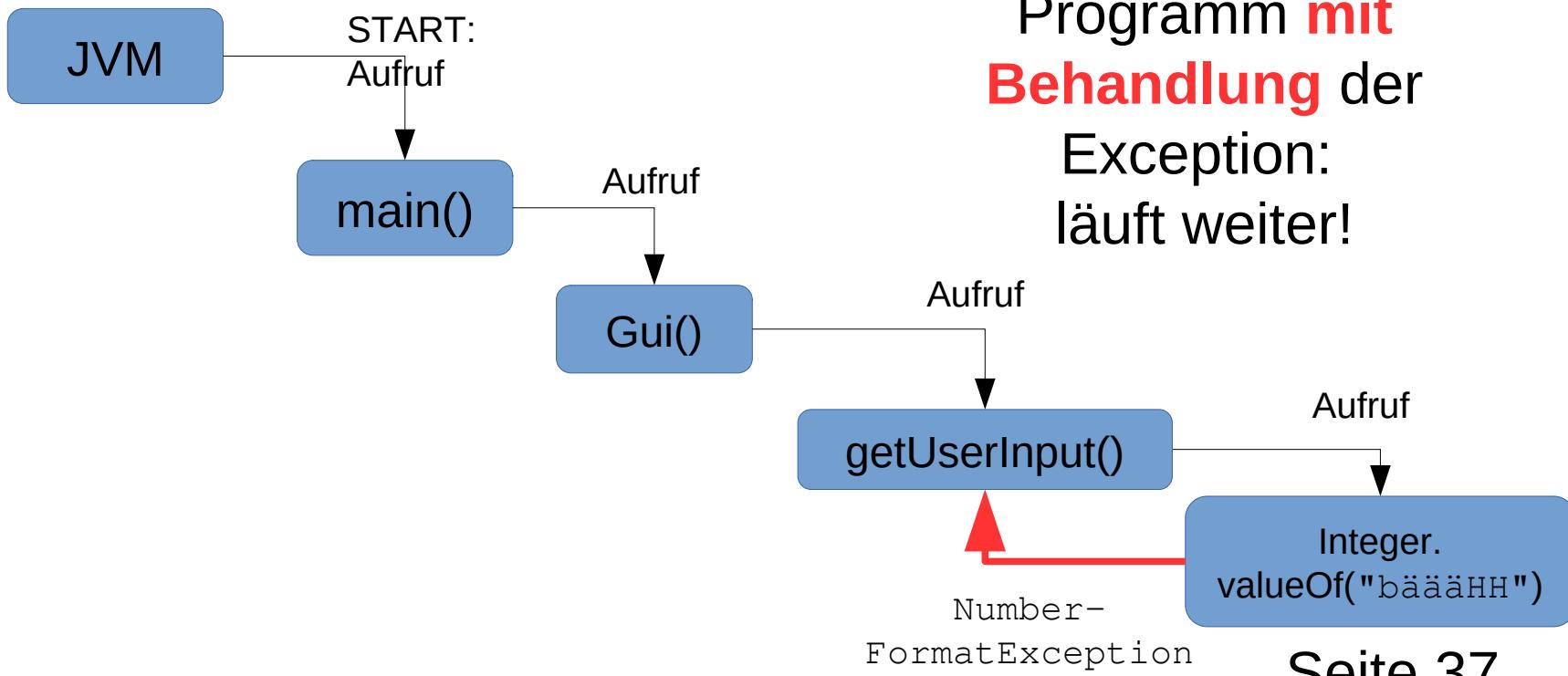
Integer.valueOf()
in einem komplexen
Programm **mit**
Behandlung der
Exception:
läuft weiter!





Kap. 6.1

Demo





Kap. 6

Fazit

- Ein Laufzeitfehler bedeutet nicht (immer) das Ende des Programms
- Bestimmte Fehler können behandelt/abgefangen werden
- Wird ein Fehler behandelt läuft das Programm weiter!
- Wird er nicht behandelt bricht das Programm ab
 - Ausgabe der Exception
 - Ausgabe des “Call stack”



Der Call Stack

Hex-Zahl her aber schnell: Bäääh

Exception in thread "main" java.lang.NumberFormatException: For input string: "Bäääh"
at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.valueOf(Integer.java:957)
at Gui.getUserInput(CallStack.java:11)
at Gui.<init>(CallStack.java:21)
at CallStack.main(CallStack.java:29)

Fehlermeldung
"Call Stack" mit Zeilennummern



CUT: Q&A



Beschreibung von Fehlern



Beschreibung von Fehlern

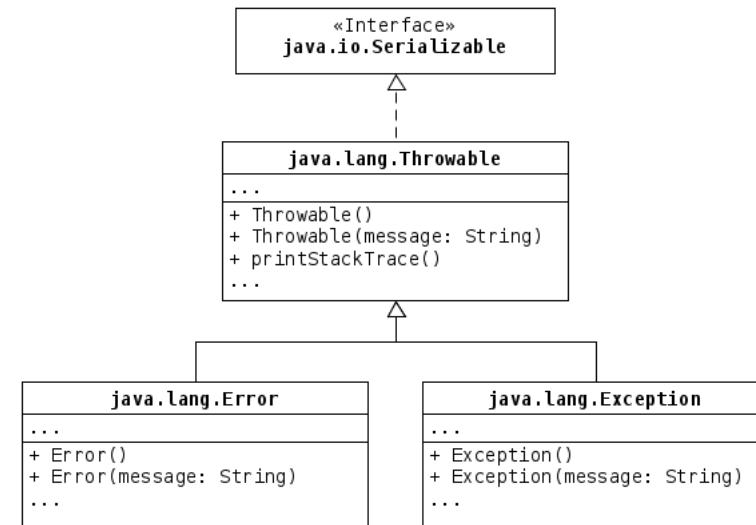
- Java beschreibt Laufzeitfehler durch Instanzen von Klassen, die von `Throwable` erben
 - tragen Informationen über Art und Ort des Fehlers
 - spezifische Fehlermeldung ebenfalls enthalten
- jede UnterkLASSE von `Throwable` repräsentiert einen bestimmten FehlerTyp
- salopp bezeichnet man alle Laufzeitfehler als “**Exceptions**”



Kap. 6.3

Beschreibung von Fehlern

- Alle möglichen Fehler bilden eine Vererbungshierarchie
- Basisklasse: Throwable
 - Error: sehr schwere Fehler, nur für JVM
 - Exception: leichtere Fehler, für Programmierer*innen

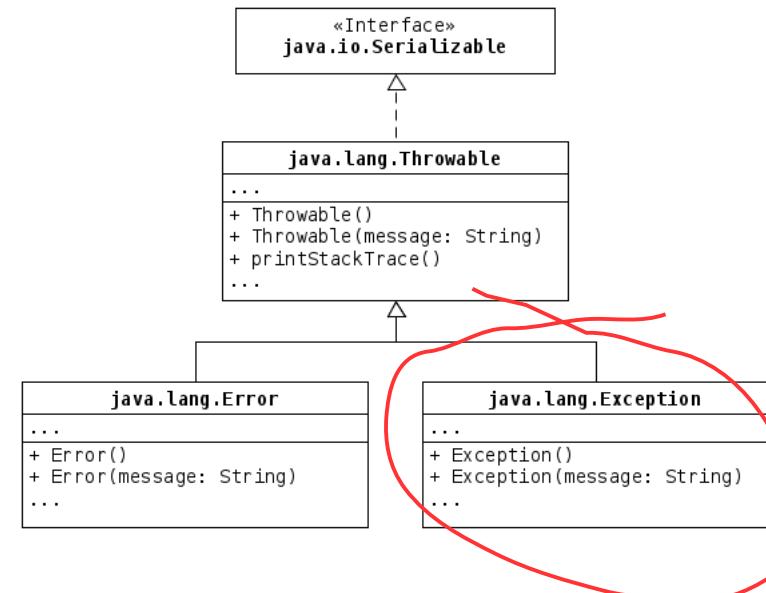




Kap. 6.3

Beschreibung von Fehlern

- Für uns relevant sind nur die Unterklasser von Exception

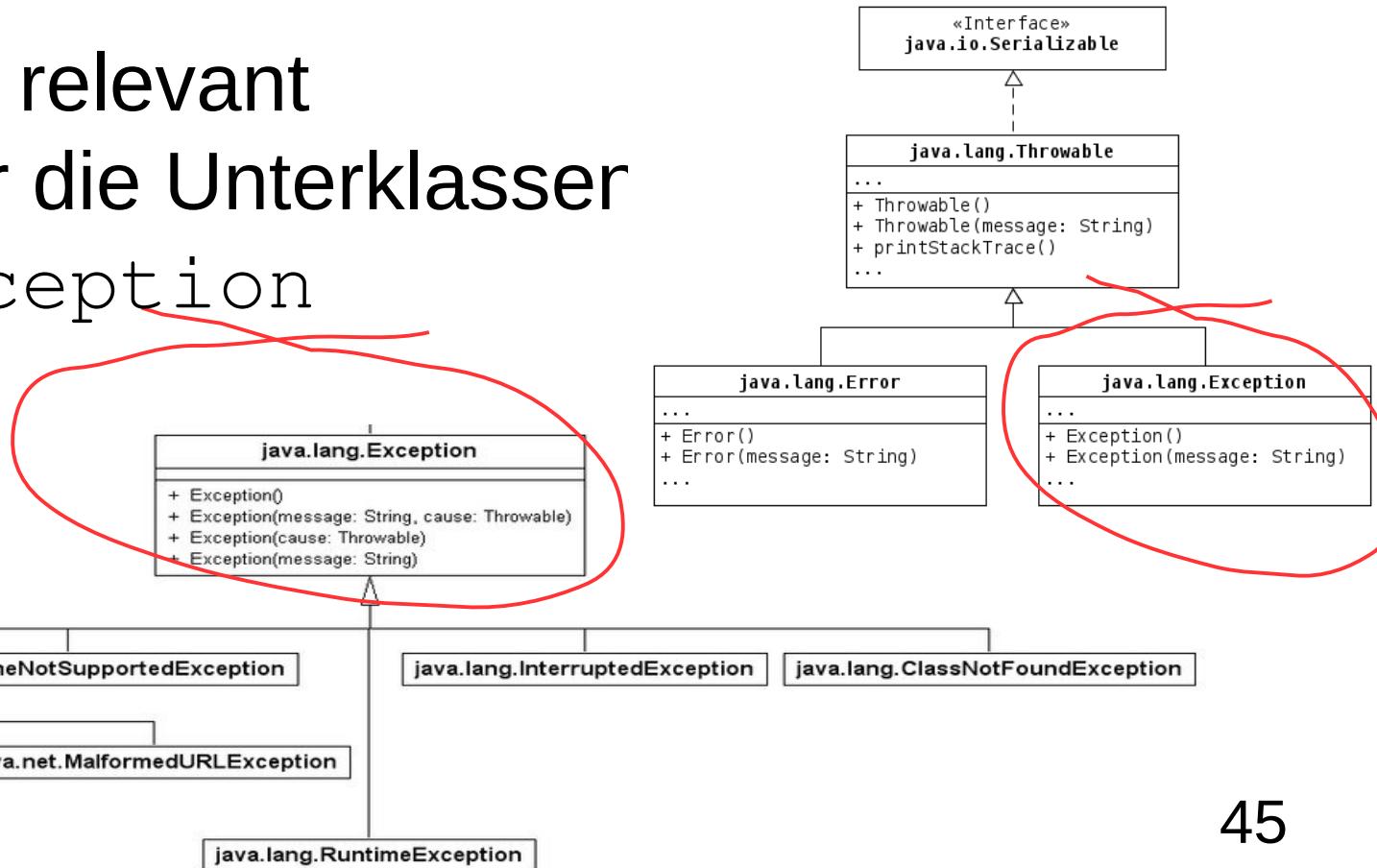




Kap. 6.3

Beschreibung von Fehlern

- Für uns relevant sind nur die Unterklasser von `Exception`

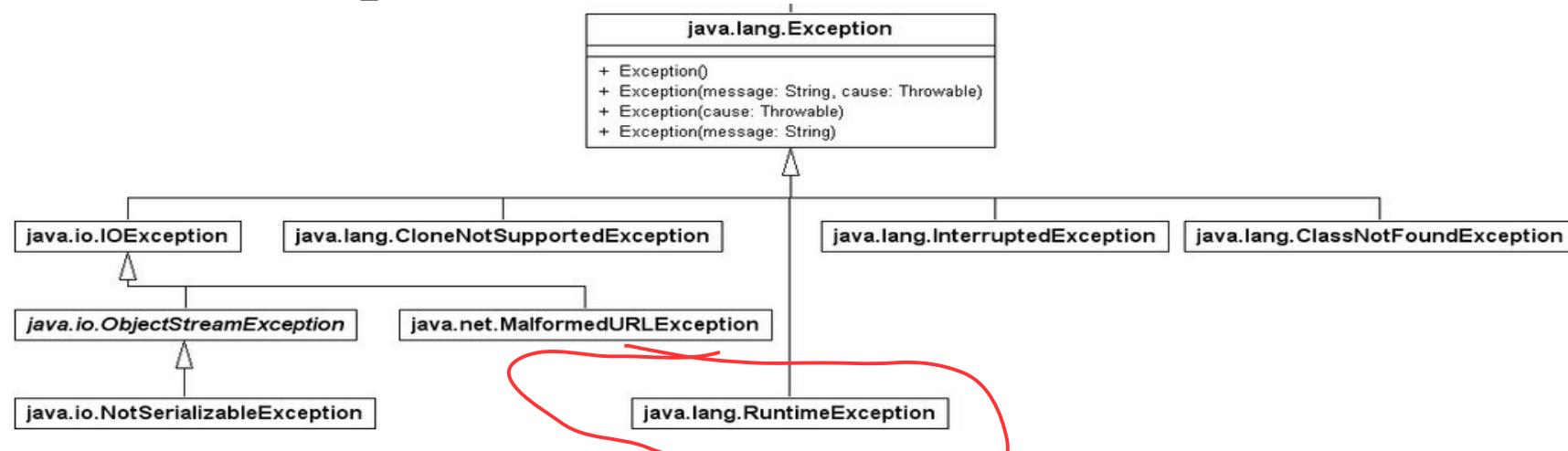




Kap. 6.3

Beschreibung von Fehlern

- Für uns relevant sind nur die Unterklassen von Exception . . .

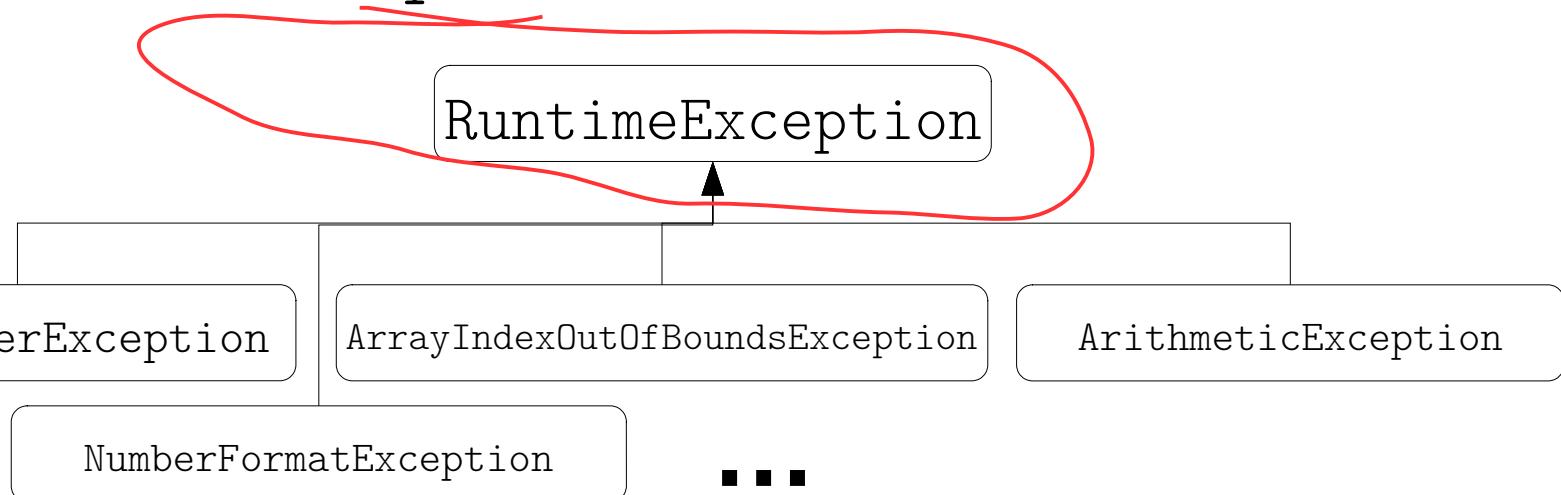




Kap. 6.3

Beschreibung von Fehlern

- ... und insbesondere die Unterklassen von RuntimeException!

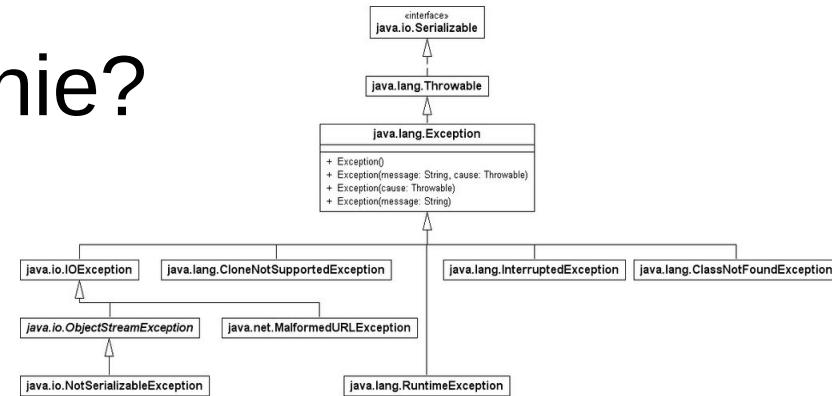




Kap. 6.3

Beschreibung von Fehlern

- Warum Klassenhierarchie?
 - erlaubt es Fehler sehr allgemein oder sehr präzise anzugeben
 - erlaubt es Fehler einfach zu behandeln (später!!)
 - erlaubt es, eigene Fehlerklassen zu erstellen





Geprüfte und ungeprüfte Exceptions



Kap. 6.2

Geprüfte und ungeprüfte Exceptions

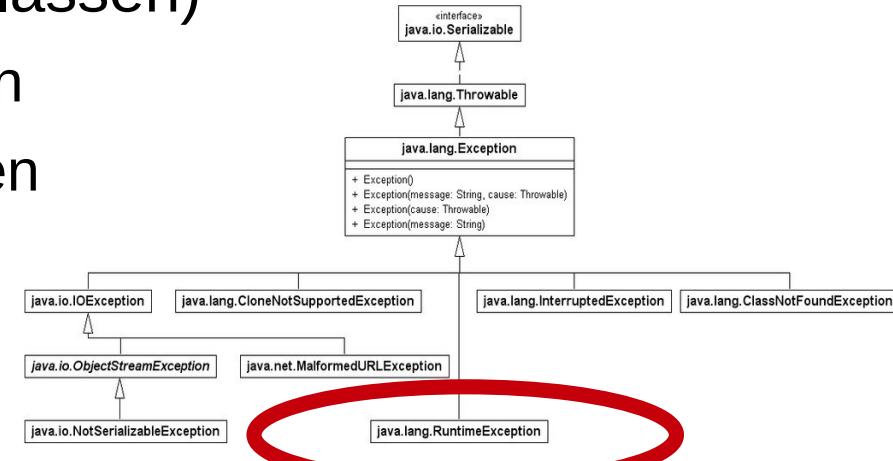
- Bestimmte Ausnahmen werden nicht aktiv von der Programmierer*in verursacht
- weisen eher auf Programmierfehler hin
 - NullPointerException
 - ArrayIndexOutOfBoundsException
 - Oberklasse: RuntimeException
- sollten nicht behandelt werden sondern Programm sollte korrigiert werden
- (aber: können behandelt werden)



Kap. 6.2

Geprüfte und ungeprüfte Exceptions

- Bei allen Ausnahmen außer RuntimeException (+Unterklassen) prüft der Compiler, ob sie behandelt werden (**“checked exception”**)
- RuntimeException (+Unterklassen)
 - muss nicht deklariert werden
 - muss nicht behandelt werden
 - “unchecked exception”**





Eigene Fehlerklassen definieren

- Eigentlich nur nötig, eine UnterkLASSE von Throwable (bzw. Exception) zu erzeugen
- Kein Überschreiben von Methoden nötig!
- Muss Fehler behandelt werden?
 - Ja: UnterkLASSE von Exception
 - Nein: UnterkLASSE von RuntimeException



CUT: Q&A



Zusammenfassung

- Neue Sprachelemente:
 - `throw` zur Auslösung eines Fehlers und Rückgabe der Fehlerinformation über separaten Rückgabekanal
 - `throws` zur Deklaration eines Fehlers in der Methodensignatur
 - `try..catch..finally` zur Behandlung von Fehlern
 - Klassenhierarchie von Fehlern basierend auf `Throwable`



Programmierung 2

GUI-Programmierung mit Swing

Alexander Gepperth, Juni 2022



Kap. 14

GUIs programmieren mit Swing

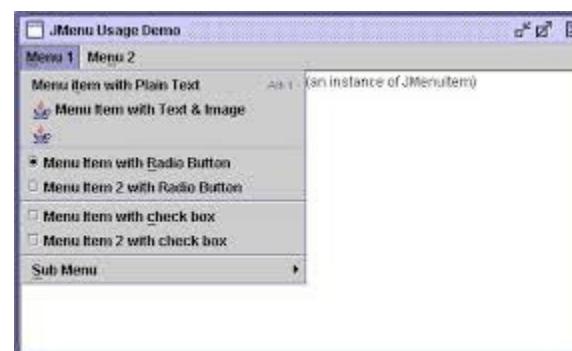
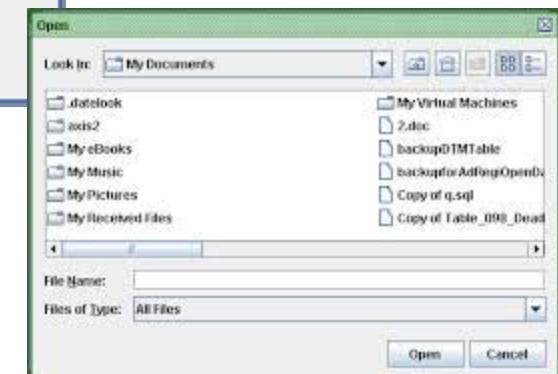
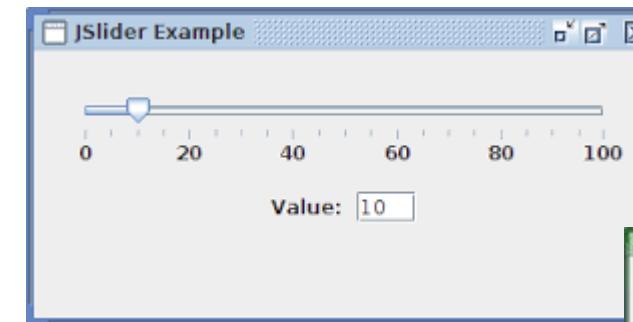
- Swing: Standardbibliothek zur Erstellung grafischer Oberflächen (GUIs)
- In Java (JRE) standardmäßig enthalten
 - läuft überall wo Java auch läuft
 - d.h. (quasi) überall!
- Funktioniert in anderen Programmiersprachen (mit anderen Bibliotheken) ähnlich!



Kap. 14

GUIs programmieren mit Swing

- Verfügbare Bedienelemente:
 - Buttons
 - Slider
 - Dateiauswahl
 - Menüleisten

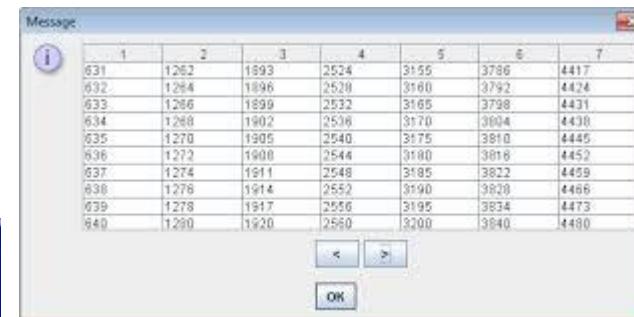
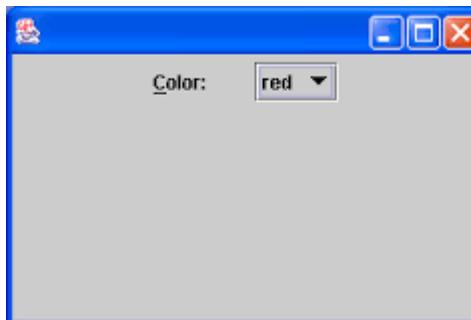




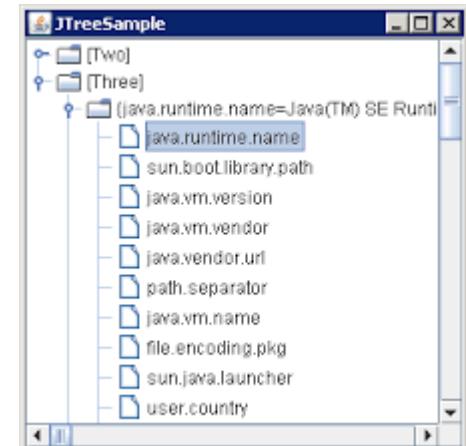
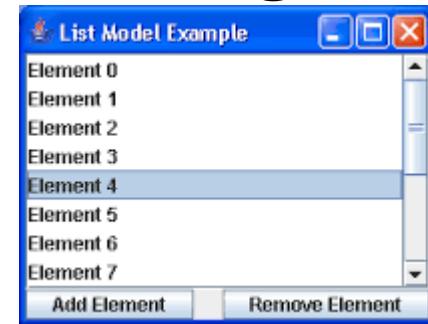
Kap. 14

GUIs programmieren mit Swing

- Verfügbare Bedienelemente:
 - Auswahllisten
 - Ausklappbare Auswahllisten
 - Tabellen
 - Comboboxen



1	2	3	4	5	6	7
631	1262	1893	2524	3155	3786	4417
632	1264	1896	2528	3160	3792	4424
633	1266	1899	2532	3165	3798	4431
634	1268	1902	2536	3170	3804	4438
635	1270	1905	2540	3175	3810	4445
636	1272	1908	2544	3180	3816	4452
637	1274	1911	2548	3185	3822	4459
638	1276	1914	2552	3190	3828	4466
639	1278	1917	2556	3195	3834	4473
640	1280	1920	2560	3200	3840	4480

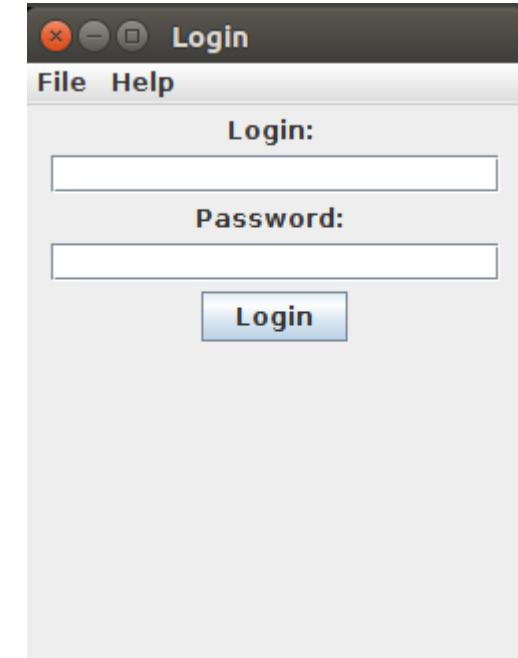




Kap. 14

Swing: Grundprinzipien

- Grundprinzipien:
 - Jedes GUI-Element ist eine Instanz einer spezialisierten Klasse
 - GUI-Elemente können andere enthalten → Hierarchie!
 - Wir können die Reaktionen jedes GUI-Elements auf bestimmte Ereignisse über **Callbacks/Listener** definieren

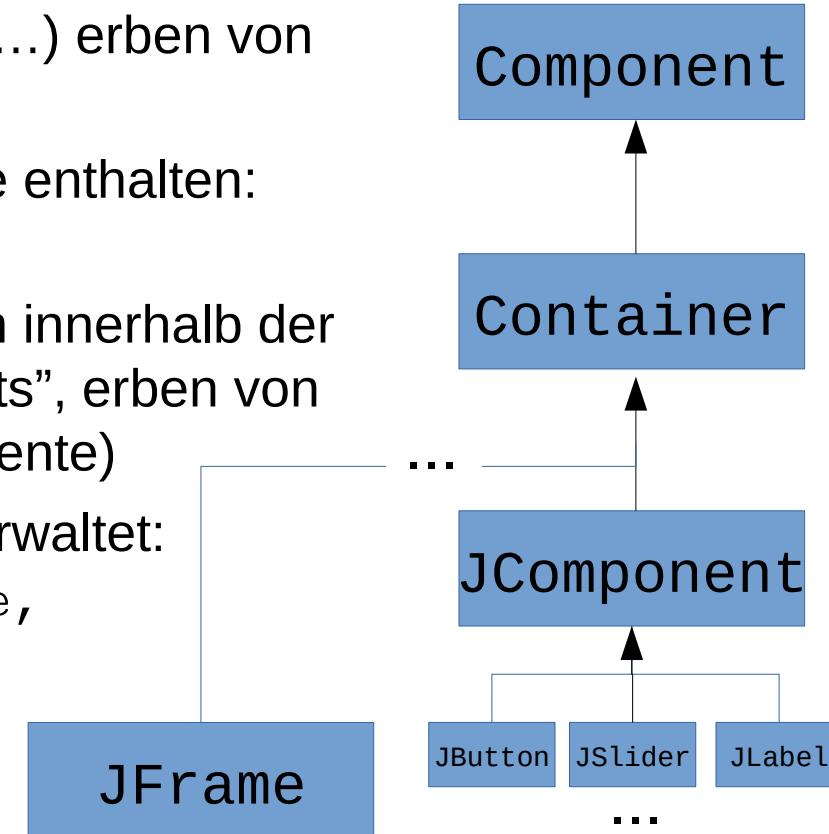




Kap. 14

Swing: Vererbungshierarchie

- Alle GUI-Elemente (Fenster, Buttons, ...) erben von Component: "Komponenten"
- manche GUI-Elemente können andere enthalten: "Container", erben von Container
- die meisten Komponenten werden rein innerhalb der JVM verwaltet: "lightweight components", erben von JComponent (alle wichtigen GUI-Elemente)
- einige werden vom Betriebssystem verwaltet: "heavyweight components" wie JFrame, JDialog (erben nur von Container)





Wichtige Swing-Klassen

- Alle Swing-Klassen sind im Package `javax.swing` enthalten
- sehr viele Klassen: der Einfachheit halber sollten Sie immer benutzen:
`import javax.swing.*;`
- Manche Hilfsklassen sind im älteren Package `java.awt` enthalten (wird gesondert erwähnt)



Kap. 14

Wichtige Swing-Klassen

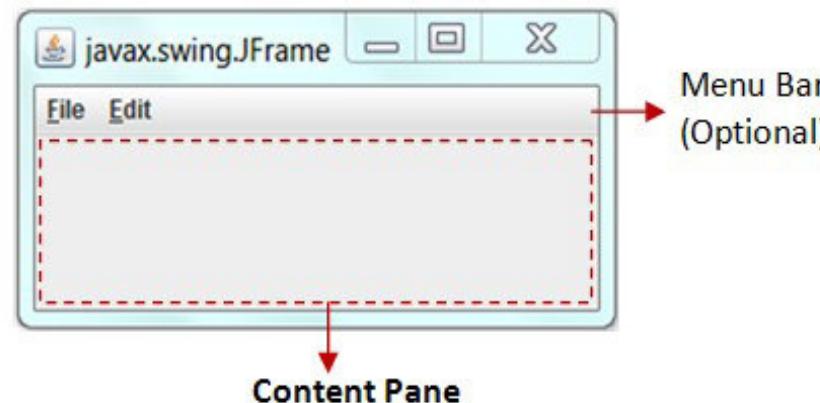
- Alle Swing-Klassen sind im Package `javax.swing` enthalten
- sehr viele Klassen: der Einfachheit halber sollten Sie immer benutzen:
`import javax.swing.*;`
- Manche Hilfsklassen sind im älteren Package `java.awt` enthalten (wird gesondert erwähnt)
- Welche Klasse aus `java.awt` kennen wir??



Kap. 14

Wichtige HeavyWeight-Klassen

- Hauptfenster: JFrame
 - Hauptfenster einer Anwendung (heavyweight component)
 - muss von jeder GUI-Anwendung erzeugt werden
 - Top-Level Container: enthält alle anderen GUI-Elemente
 - nicht direkt, sondern in separatem Container: “contentPane”





Kap. 14

Wichtige HeavyWeight-Klassen

- Hauptfenster: JFrame
 - Hauptfenster sichtbar machen und GUI starten:
`void setVisible(boolean b)`
 - Hauptfenstergröße festlegen:
`void setSize(int width, int height)`
 - Menüleiste hinzufügen:
`void setJMenuBar(JMenuBar menubar)`
 - contentPane setzen (meist JPanel-Instanzen):
`void setContentPane (Container contentPane)`
 - neues GUI-Element hinzufügen:
`void add(Component comp)`



Kap. 14

Wichtige HeavyWeight-Klassen

- Hauptfenster: JFrame
 - wird meist im Konstruktor der Programm-Hauptklasse instanziert

```
JFrame gui = new JFrame() ;  
JPanel contentPane = new JPanel() ;  
gui.setContentPane(contentPane) ;  
// GUI-Elemente erzeugen, zu panel hinzufuegen  
gui.setSize(100, 100) ;  
gui.setVisible(true) ;
```

- **Demo!!**



Kap. 14

Wichtige LightWeight-Klassen

- Rechteckiger Bereich: JPanel
 - enthält andere Komponenten
 - unsichtbar, nur die enthaltenen Elemente sind sichtbar
 - enthaltene Elemente **verwaltet durch Layout-Manager**
 - wichtige öffentliche Methoden:
 - Component add(Component comp)
fügt eine Komponente hinzu



Kap. 14

Einschub: Demo (typisches Programm mit Swing-GUI)

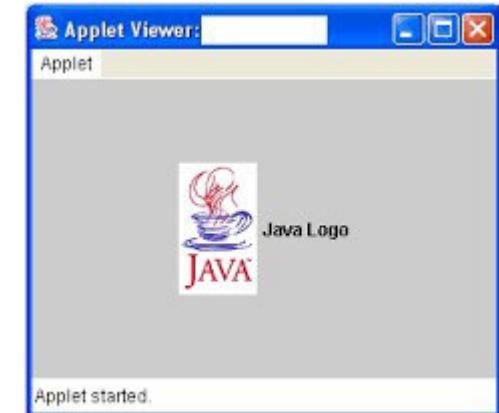
- Aufgaben der Programmiererin:
 - Hauptfenster: Instanz von `JFrame` erzeugen
 - Instanz von `JPanel` erzeugen, als `contentPane` definieren
 - GUI-Elemente erzeugen, konfigurieren und zur `contentPane` hinzufügen
 - Callbacks für Benutzerinteraktion registrieren
 - optional: Größe des Hauptfenster setzen
 - Hauptfenster sichtbar machen (`setVisible`)



Kap. 14

Wichtige LightWeight-Klassen

- Statischer Text: JLabel
 - Anzeigen von Text und/oder Grafiken
 - keine Interaktion vorgesehen
- Wichtige Methoden:
 - JLabel(String arg0)
 - JLabel(Icon arg0)

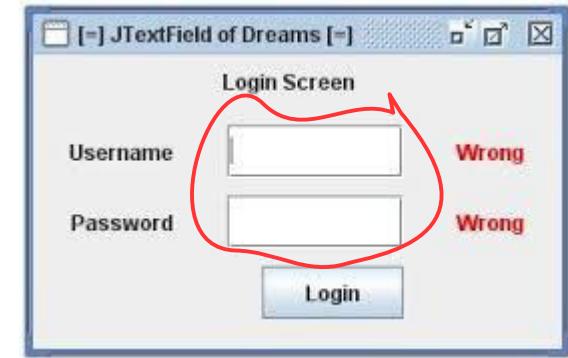




Kap. 14

Wichtige LightWeight-Klassen

- Texteingabe: JTextField
 - Texteingabefeld für Benutzer
 - Text kann vorausgefüllt werden
- Wichtige Methoden:
 - Konstruktor mit Text:
`JTextField(String text, int columns)`
 - Konstruktor ohne Text: `JTextField()`
 - Callback/Listener registrieren:
`addActionListener(ActionListener l)`

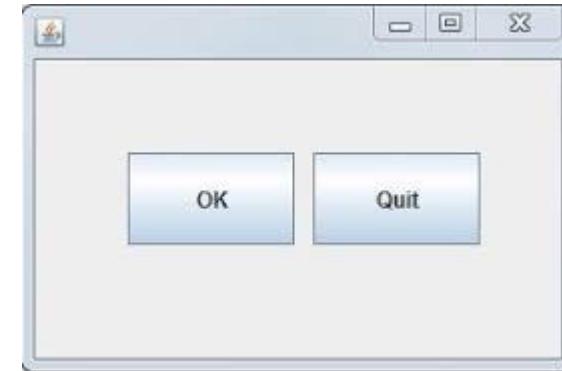




Kap. 14

Wichtige LightWeight-Klassen

- Schaltfläche: JButton
 - Button auf den geklickt werden kann
 - Text oder Bild frei wählbar
- Wichtige Methoden:
 - Konstruktor mit Text: JButton(String text)
 - Konstruktor mit Bild: JButton(Icon icon)
 - Callback/Listener registrieren:
`addActionListener(ActionListener l)`





Kap. 14

Wichtige LightWeight-Klassen

- Wichtige Klassen: `JMenu`
 - repräsentiert ein Menü oder ein Untermenü
 - Menüs haben Titel (Menüleiste/Obermenü)
 - enthält Instanzen von `JMenuItem` oder Separatoren
 - wichtige Methoden:
 - Erzeuge Menü mit Titel: `JMenu(String s)`
 - Erzeuge Menü ohne Titel: `JMenu()`
 - Eintrag hinzufügen: `JMenuItem add(JMenuItem menuItem)`
 - Trennlinie hinzufügen: `void addSeparator()`

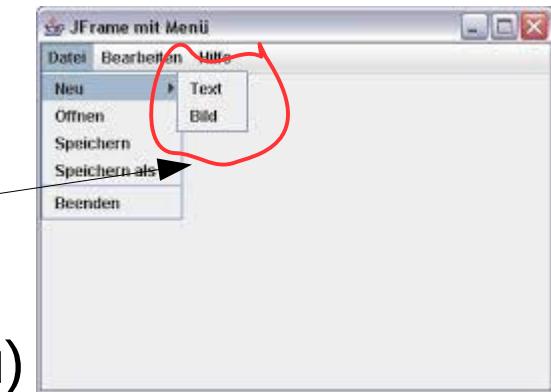




Kap. 14

Wichtige LightWeight-Klassen

- Wichtige Klassen: `JMenu`
 - repräsentiert ein Menü oder ein Untermenü
 - Menüs haben Titel (Menüleiste/Obermenü)
 - enthält Instanzen von `JMenuItem` oder Separatoren
 - wichtige Methoden:
 - Erzeuge Menü mit Titel: `JMenu(String s)`
 - Erzeuge Menü ohne Titel: `JMenu()`
 - Eintrag hinzufügen: `JMenuItem add(JMenuItem menuItem)`
 - Trennlinie hinzufügen: `void addSeparator()`

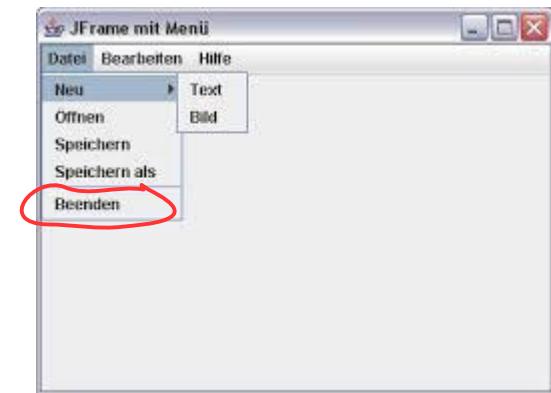




Kap. 14

Wichtige LightWeight-Klassen

- Menüeintrag: JMenuItem
- wichtige Methoden:
 - erzeuge Eintrag mit Titel: JMenuItem(String arg0)
 - Listener registrieren für den Fall dass der Menüeintrag gewählt wird:
`void addActionListener(ActionListener al)`

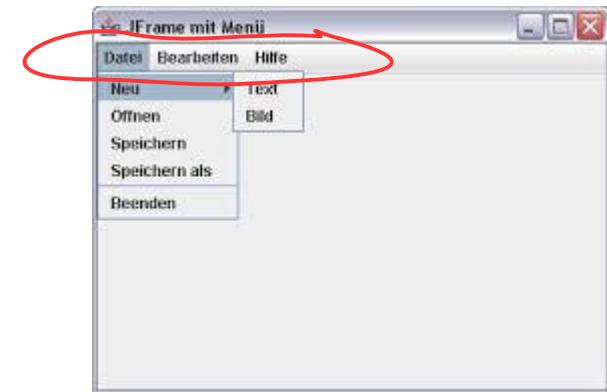




Kap. 14

Wichtige LightWeight-Klassen

- Wichtige Klassen: JMenuBar
 - repräsentiert eine Menüleiste
 - wichtige Methoden:
 - erzeuge Menüleiste:
`JMenuBar()`
 - Menü hinzufügen:
`JMenu add(JMenu c)`

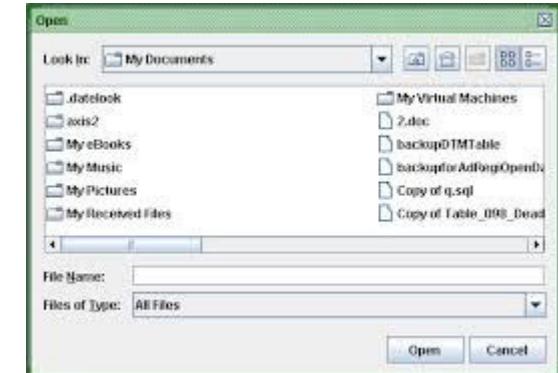




Kap. 14

Wichtige LightWeight-Klassen

- Wichtige Klassen: JFileChooser
 - Eingabedialog für Dateinamen oder Pfade
- wichtige Methoden:
 - Konstruktor: JFileChooser()
 - Fenster öffnen und Benutzer wählen lassen:
`int showOpenDialog(Component parent)`
Liefert:
 - JFileChooser.APPROVE_OPTION
 - JFileChooser.CANCEL_OPTION
 - JFileChooser.ERROR_OPTION
 - liefere gewählten Dateinamen:
`File getFileSelected()`





Kap. 14

Wichtige LightWeight-Klassen

- Alle “lightweight”-Komponenten erben von JComponent
 - eigene Größe und Position setzen:
`void setBounds(int x, int y, int w, int h)`
 - Layout-Manager setzen, später:
`void setLayout(LayoutManager m)`
 - Ausrichtung setzen (für LayoutManager, später):
`void setAlignmentX(float f)`
 - vordefinierte Konstanten für f:
`Component.CENTER_ALIGNMENT`, `Component.LEFT_ALIGNMENT`,
`Component.RIGHT_ALIGNMENT`, `Component.TOP_ALIGNMENT`,
`Component.BOTTOM_ALIGNMENT`
 - Jede JComponent-Instanz kann andere enthalten:
`Component add(Component comp)`



CUT: Q&A



Kap. 14

Automatisches Layout für GUIs LayoutManager

- Bisher: GUI-Elemente wurden einem Container hinzugefügt ohne auf Position zu achten
- Warum war die Angabe der Position nicht nötig?
→ wird automatisch vom LayoutManager festgelegt
- Warum ist das sinnvoll? → Größenänderungen des Hauptfensters lassen sich nicht vorhersagen, GUI sollte trotzdem sinnvoll angepasst werden

package: `java.awt`



Kap. 14

Automatisches Layout für GUIs LayoutManager

- Für jeden Swing-Container lässt sich ein LayoutManager festlegen:
`void setLayout(LayoutManager mgr)`
- Übergebene Instanzen müssen das Interface LayoutManager implementieren
- LayoutManager regeln die Anordnung von GUI-Elementen innerhalb eines Containers
- Eigene LayoutManager-Unterklassen möglich!

package: `java.awt`



Kap. 14

Automatisches Layout für GUIs LayoutManager

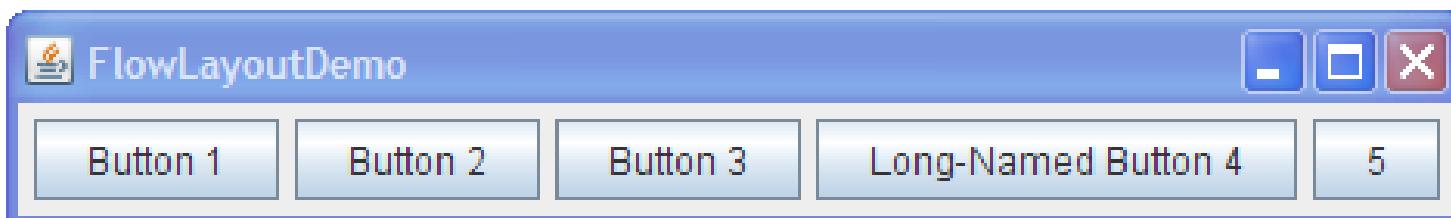
- Layout-Management abschalten:
`setLayout(null) ;
aufrufen!`
- dann: absolute Positionierung nötig mit
`setBounds(...);`
- kann hässlich sein...

package: `java.awt`



LayoutManager: FlowLayout

- alles in einer Zeile
- neue Zeile falls kein Platz mehr ist
- Konstruktor:
`FlowLayout()`



package: `java.awt`

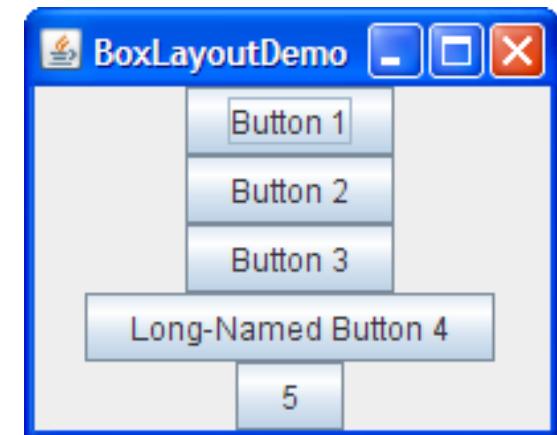


Kap. 14

LayoutManager: BoxLayout

- alles in einer Zeile oder Spalte
- Konstruktor:
`BoxLayout(Container target, int axis)`
- axis-Parameter:
 - `BoxLayout.X_AXIS`
 - `BoxLayout.Y_AXIS`
 - `BoxLayout.LINE_AXIS`
 - `BoxLayout.PAGE_AXIS`

package: javax.swing

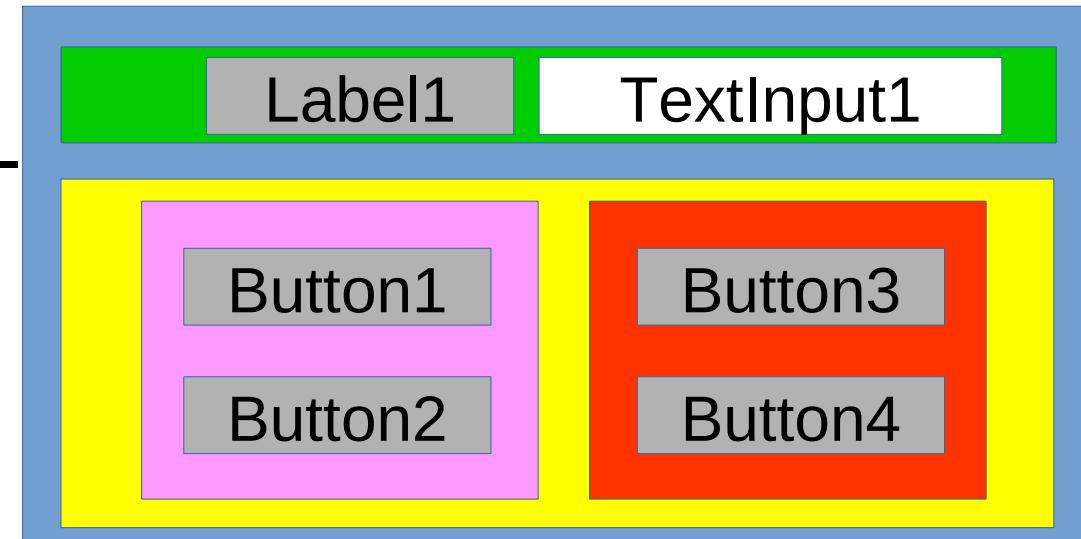




Kap. 14

Automatisches Layout für GUIs LayoutManager

- Eigene Layout-Manager für Bereiche der GUI möglich
- Bereiche: JPanel-Instanzen
- Welche L.-Manager sind für dieses Layout nötig?

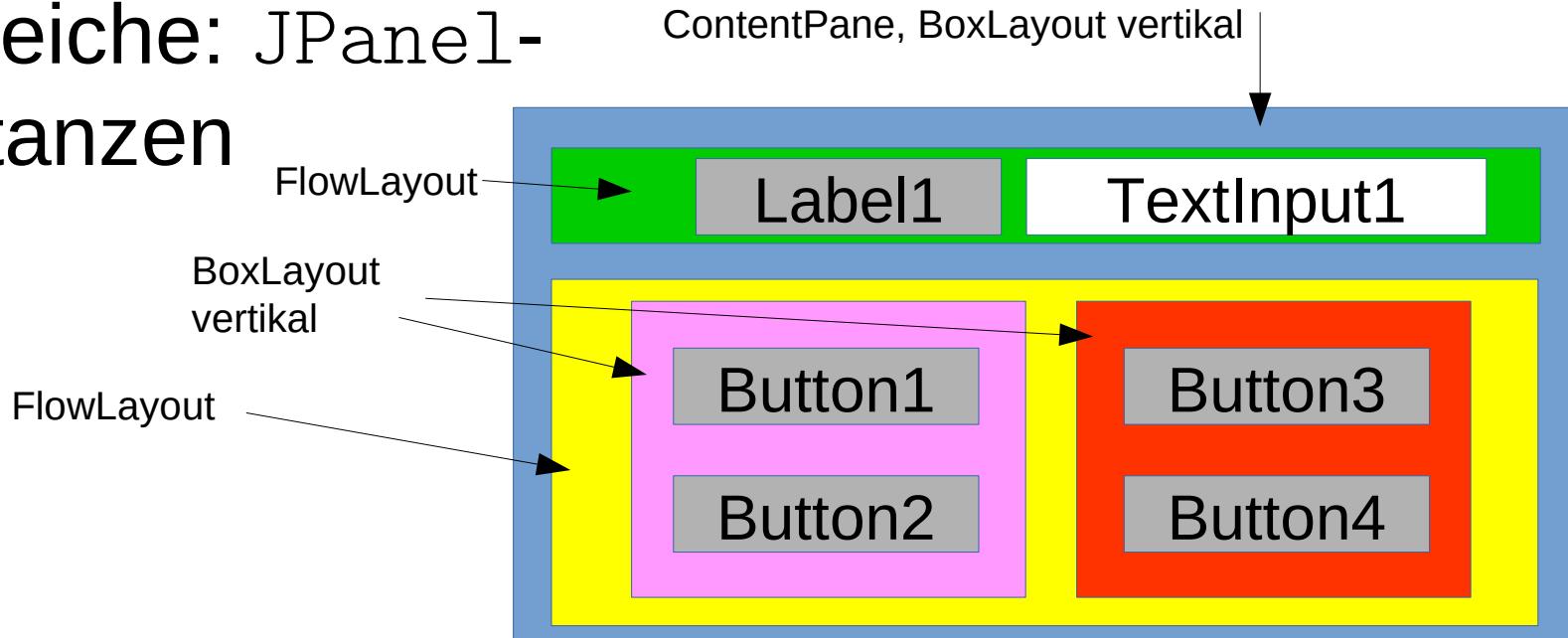




Kap. 14

Automatisches Layout für GUIs LayoutManager

- Eigene Layout-Manager für Bereiche der GUI möglich
- Bereiche: JPanel-Instanzen





CUT: Q&A



Kap. 14

Ereignisse behandeln

- GUI muss auf Benutzereingaben reagieren
 - Klicks auf Buttons
 - Eingabe in Textfelder
 - Menüpunkt gewählt
 - Taste gedrückt, Mausklick
 - Slider verändert
 - Checkbox angeklickt
 - ...



Kap. 14

Ereignisse behandeln

- Behandeln von Ereignissen mit **Listeners/Callbacks**
 - ideal wäre: eine Methode angeben, die bei einem bestimmten Ereignis aufgerufen wird



Kap. 14

Ereignisse behandeln

- Behandeln von Ereignissen mit **Listeners/Callbacks**
 - ideal wäre: eine Methode angeben, die bei einem bestimmten Ereignis aufgerufen wird
 - unmöglich, aber: wir können Objekt angeben, welches das Interface ActionListener implementiert
 - Methode actionPerformed(ActionEvent ae) muss existieren !
 - und muss vom Programmierer überschrieben werden!
 - Fast alle GUI-Elemente besitzen die Methode addActionListener(ActionListener a)



Kap. 14

Ereignisse behandeln

- Das Interface ActionListener
 - Schreibt eine einzige Methode vor:
 - void actionPerformed(ActionEvent ae)
 - addActionListener kann natürlich keine Instanzen von ActionListener übergeben bekommen..



Kap. 14

Ereignisse behandeln

- Das Interface ActionListener
 - Schreibt eine einzige Methode vor:
 - void actionPerformed(ActionEvent ae)
 - addActionListener kann natürlich keine Instanzen von ActionListener übergeben bekommen..
 - ... aber von Klassen, die dieses Interface implementieren und actionPerformed überschreiben
 - Diese Methoden können dann den ActionEvent auswerten und entsprechend reagieren!
- package:
java.awt.event



Kap. 14

Ereignisse behandeln

- Generell gute Praxis: die Hauptklasse des Programms implementiert ActionListener
- → nur eine einzige actionPerformed-Methode
- diese Methode muss per getSource() testen, von welchem GUI-Element das Ereignis kommt
- Deshalb ist es wichtig, Referenzen auf alle GUI-Elemente als Attribute der Hauptklasse zu deklarieren
- actionPerformed erhält eine ActionEvent-Instanz um Art des Events anzuzeigen



Kap. 14

Ereignisse behandeln

- Die Klasse ActionEvent
 - Welche Komponente hat Eingabe bekommen:
`JComponent getSource()`
liefert eine Referenz auf das betroffene GUI-Element,
das kann per == verglichen werden
 - was ist passiert: `String getActionCommand()`
liefert eine Beschreibung der Aktion, Bedeutung hängt vom
GUI-Element ab!



Kap. 14

Ereignisse behandeln

- Die Klasse ActionEvent
 - Welche Komponente hat Eingabe bekommen:
`JComponent getSource()`
liefert eine Referenz auf das betroffene GUI-Element,
das kann per `==` verglichen werden
 - was ist passiert: `String getActionCommand()`
liefert eine Beschreibung der Aktion, Bedeutung hängt vom
GUI-Element ab!

Warum `==` ?



CUT: Q&A



Zusammenfassung

- Neue Sprachelemente: keine!
- Standardbibliothek innerhalb der JRE: Swing
 - einfache Erstellung von GUIs (grafische Benutzeroberflächen)
 - wichtigste Klassen der Bibliothek und rudimentäre Benutzung
 - wichtig: online-Dokumentation benutzen können!!
 - mögliche Klausuraufgabe: programmieren Sie eine GUI die so aussieht!