

# Einführung und Motivation

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Ziele der Vorlesung (1)

- Grundlagen der Programmierung vertiefen
- Anknüpfen an Wissen aus Einführung in die Programmierung
- Grundkonzepte der Objektorientierung verstehen und anwenden
  - Analyse
  - Design
  - Implementierung
- Erlernen einer modernen objektorientierten Programmiersprache
- Verstehen moderner Programmierprinzipien
- Best practices lernen

# Ziele der Vorlesung (2)

Nach dieser Vorlesung können Sie:

1. Die Grundkonzepte der objektorientierten Programmierung wiedergeben und anwenden
2. Bestehende Programme analysieren, deren Funktionsweise und die verwendeten Techniken identifizieren
3. Problemstellungen analysieren, beschreiben und daraus die Architektur eines Programms erarbeiten und umsetzen
4. Codequalität bewerten und damit „guten“ von „schlechtem“ Code unterscheiden und dies im eigenen Code auch umsetzen.

# Inhalt

- Grundelemente der Programmierung in Java
- Grundlagen der Objektorientierung
- Einführung in UML
- Vererbung und Polymorphie
- Ausnahmenbehandlung
- Unit-Tests
- Java Collections
- Generische Programmierung in Java
- Funktionale Programmierung in Java
- Streams in Java
- Refactoring
- GUI-Programmierung
- Einführung in die Java Virtual Machine
- Clean Code

# Allgemeine Literatur



Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)  
12. Auflage frei verfügbar: <http://openbook.rheinwerk-verlag.de/javainsel/>



Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021  
2. Auflage frei verfügbar: <http://openbook.rheinwerk-verlag.de/oop/>



Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Verlag, 8. Auflage, 2016  
7. Auflage frei verfügbar (Uni-Netz):  
<http://link.springer.com/book/10.1007%2F978-3-8348-2270-3>



Simon Harrer, Jörg Lenhard, Linus Dietz: **Java by Comparison. Become a Java Craftsman in 70 Examples**. The Pragmatic Bookshelf, 1st Edition, 2018  
Frei verfügbar (Uni-Netz): <https://bibsearch.uibk.ac.at/AC16131176>

# Lehrkonzept

Zeit  
↓

## Vorbereitung

- Vorlesungsfolien
- Vorlesungsvideos
- Quiz lösen
- Fragen stellen/notieren

## Vorlesungseinheit

- Fragen stellen
- Wiederholung von schwierigen Themen
- Aktive Lerntätigkeit (Programmieren)

## Nachbereitung

- Vertiefung mit Vorlesungsunterlagen

# Organisatorisches

- OLAT
  - Vorlesungsfolien, Vorlesungsvideos, Quiz, weitere Links, Ankündigungen
- GIT
  - Alle Beispiele auf <https://git.uibk.ac.at/c7031278/PMExamples>
- Vorlesung
  - Vorlesungsvideos werden für die entsprechende Woche via OLAT veröffentlicht
  - Fragestunde: Do. 08:15 - 10:00, HS A
- Prüfung
  - Schriftliche Klausur über alle Themen, welche in der Vorlesung besprochen wurden
  - Termine
    - 1. Klausur: 04.07.2022 09:00 – 12:00
    - 2. Klausur: 26.09.2022 09:00 – 12:00

# Kontakt & Fragen

- E-Mail: [lukas.kaltenrunner@uibk.ac.at](mailto:lukas.kaltenrunner@uibk.ac.at)
- [Chat Matrix/Element](#): @lukas.kaltenrunner:uibk.ac.at
- Sprechstunde: Di. 10:00 – 11:00 nach Vereinbarung
- ARSnova (<https://arsnova.uibk.ac.at/mobile/#id/13421235>)
- Q&A
- OLAT Forum



# Folien - Codebeispiele

```
private static List<Integer> getFlaggedCells(List<Integer> l) {  
    List<Integer> r = new ArrayList<Integer>();  
    for (Integer x : l) {  
        if (x == 4) {  
            r.add(x);  
        }  
    }  
    return r;  
}
```



Verbesserungswürdiger Code  
(schlechtes Beispiel)

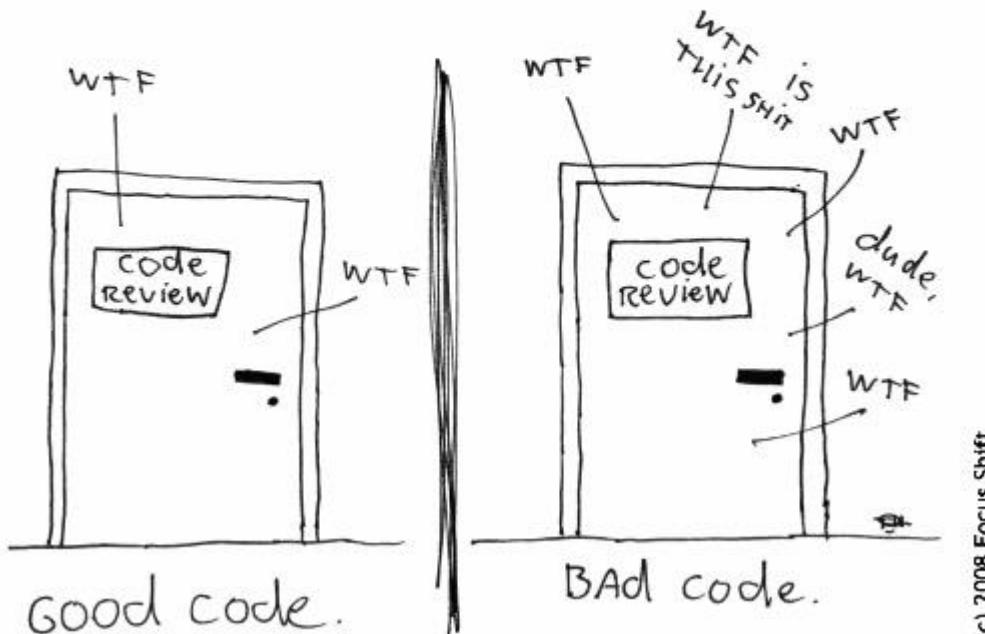
Pfad zu Sourcecode im Git-Repository  
(direkt verlinkt; hier nur Platzhalter)



<src/at/ac/uibk/pm/basics/HelloWorld.java>

# Clean Code

The ONLY VALID measurement  
OF code QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift



# Avoid Single-Letter Names

Clean Code Tipp

Bezeichnung

```
private static List<Integer> getFlaggedCells(List<Integer> gameBoard) {  
    List<Integer> flaggedCells = new ArrayList<Integer>();  
    for (Integer cell : gameBoard) {  
        if (cell == 4) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```



Verbesserter Code  
(gutes Beispiel)

- Vorher:
  - Leser muss Bedeutung (Semantik) der Variablen selbst herausfinden.
  - Variablen geben keinerlei Auskunft über Inhalt.
- Nachher: lesbarer Code

# Proseminar (1)

- Anwesenheitspflicht
- Aufgaben, Präsentation(en) und zwei Tests bestimmen die Note
  - 1. Midterm-Test: Mo. 02.05.2022, 17:15 – 19:00, für alle Gruppen vor Ort!
  - 2. Midterm-Test: Mo. 16.06.2022, 17:15 – 19:00, für alle Gruppen vor Ort!
- OLAT
  - Übungsaufgabe, Quiz, Abgaben, Ankündigungen

# Proseminar (2)

- 12 Übungsgruppen:
  - Gruppe 1: Benedikt Hupfauf (Mo, 08:15 – 10:00, rr15)
  - Gruppe 2: Lukas Kaltenbrunner (Mo, 12:15 – 12:00, rr15)
  - Gruppe 3: Lukas Kaltenbrunner (Mo, 14:15 – 16:00, rr15)
  - Gruppe 4: Eduard Frankford (Mo, 16:15 – 18:00, rr15)
  - Gruppe 5: Simon Priller (Mo, 08:15 – 10:00, rr26)
  - Gruppe 6: Simon Priller (Mo, 12:15 – 14:00, rr26)
  - Gruppe 7: Alexander Blaas (Mo, 14:15 – 16:00, rr26)
  - Gruppe 8: Alexander Blaas (Mo, 16:15 – 18:00, rr26)
  - Gruppe 9: Melanie Ernst (Mo, 12:15 – 14:00, rr25)
  - Gruppe 10: Umutcan Simsek (Mo, 14:15 – 16:00, rr25)
  - Gruppe 11: Manfred Moosleitner (Mo, 10:15 – 12:00, eLecture)[EWS]
  - Gruppe 12: Elwin Huaman Quispe (Mo, 17:15 – 19:00, eLecture)[EWS]
- Bitte Übungsgruppen-Zuordnung überprüfen

# Credits

- Foliensatz Programmiermethodik (Sommersemester 2012)  
Stefan Podlipnig
- Foliensatz Programmiermethodik (Sommersemester 2013)  
Rene Thiemann
- Foliensatz Programmiermethodik (Sommersemester 2014 – 2020)  
Eva Zangerle

# Einführung Java

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck



# Programmiersprachen

# Programmierparadigma

- Programmierparadigma = Sichtweise auf und Umgang mit den zu verarbeitenden Daten und Operationen.
- Beispiele
  - Imperatives Programmierparadigma (*LV Einf. in die Programmierung*)
    - Folge von Anweisungen die streng sequenziell abgearbeitet wird.
  - Objektorientiertes Programmierparadigma (*diese LV*)
    - Verfügt über weitergehende Konzepte wie Klassen, Vererbung und Polymorphie.
  - Funktionales Programmierparadigma (*LV Funktionale Programmierung*)
    - Rein funktionale (applikative) Programmierung kennt keine Wertzuweisung.
    - Reihe von Funktionsaufrufen, die eine Eingabe in eine Ausgabe transformiert.
  - Logisches Programmierparadigma
    - Verwendet Logik zur Darstellung und Lösung von Problemen.
    - Programm besteht aus einer Menge von Axiomen, aus denen der Interpreter eine Lösungsaussage berechnet.

# Abstraktionsgrad

- Der Abstraktionsgrad definiert, wie weit sich die Semantik der einzelnen Sprachkonstrukte von den Grundbefehlen des Prozessors unterscheidet.
- Abstraktionsgrade
  - Maschinensprache
    - Wird direkt vom Prozessor verstanden
  - Assembler
    - Einsatz symbolischer Namen, Sprungmarken etc.
  - Höhere Programmiersprachen
    - Kontrollstrukturen, Datentypen etc.
  - Objektorientierte Programmiersprachen
    - Daten und Methoden bilden eine Einheit.
  - Deklarative Programmiersprachen (Funktional/Logisch)
    - Kein Unterschied zwischen Daten und Operationen

Steigender Abstraktionsgrad



# Ausführungsschema

## Übersetzende Programmiersprachen (z.B. C, C++)

- Der Quelltext eines Programms wird vor der ersten Ausführung durch einen Übersetzer in eine Zielsprache (üblicherweise Maschinensprache) übersetzt.
- Hohe Ausführungsgeschwindigkeit
- Überprüfung auf Fehler

## Interpretierende Programmiersprachen

- Der Quelltext eines Programms wird zur Laufzeit von einem Interpreter eingelesen und Befehl für Befehl abgearbeitet.
- Langsamer
- Flexibler (bei kleinen Änderungen)

## Mischformen (z.B. Java)

- Programm wird in einen Zwischencode übersetzt.
- Übersetzter Code wird interpretiert (virtuelle Maschine).



# **Java – Eine kurze Einführung**

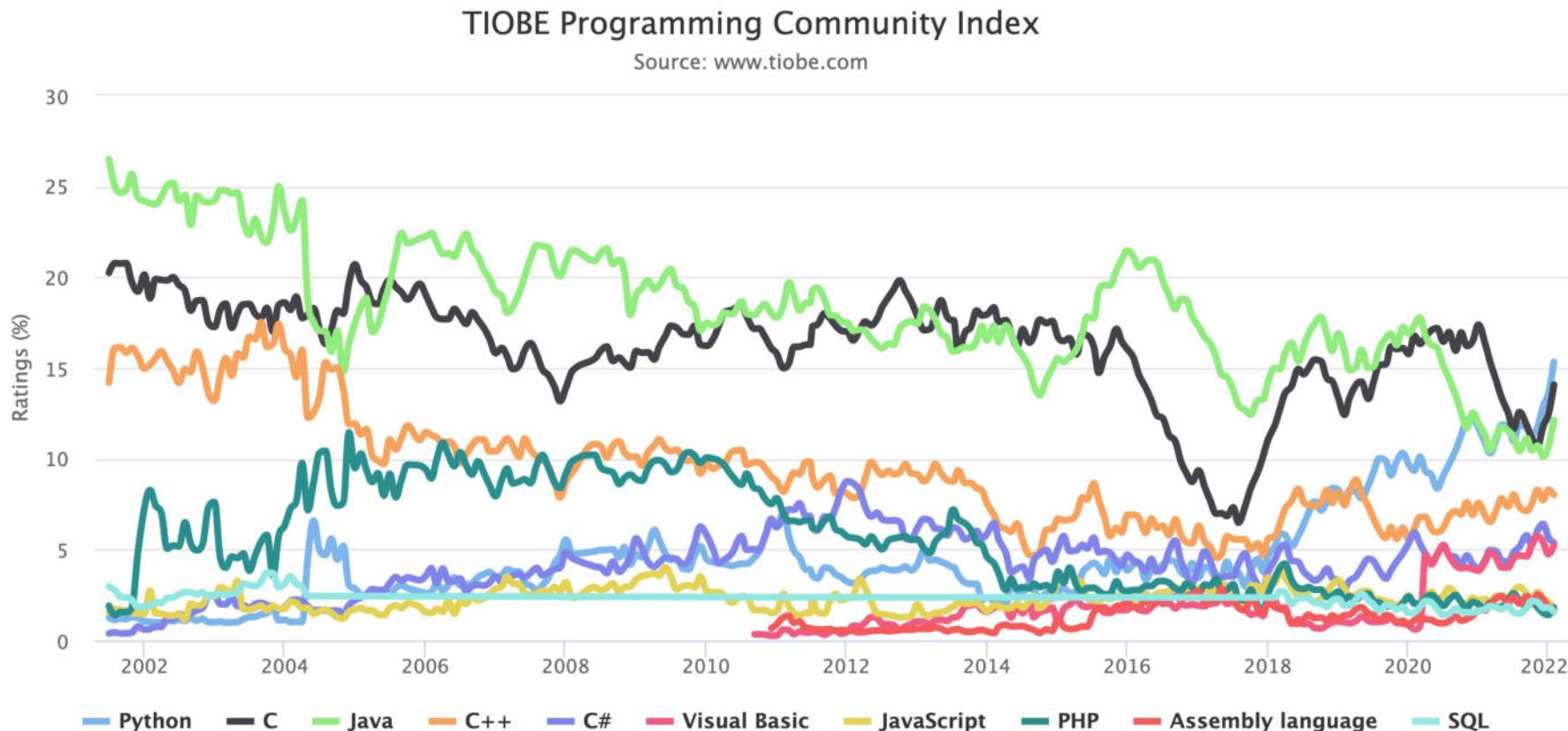
# Java (1)

- Start 1991
- Sun Microsystems (seit 2010 Tochterunternehmen der Oracle Corporation)
- Ursprünglicher Name “Oak” (Object Application Kernel)
  - Angelehnt an C++
  - Elemente aus Smalltalk (Bytecode, Garbage Collection)
- Ziel war die Entwicklung einer Hochsprache für hybride Systeme im Consumer-Electronic Bereich.
  - Kompakte Programme
  - Plattform-unabhängige Programme
- Neuausrichtung Projekt → auch für Internet-Programmierung geeignet
- Sun Demo Web-Browser HotJava in den 90-ern (Applets)
  - Kleine Programme als Applets in HTML-Seiten

# Java (2)

- Durchbruch 1995
  - Netscape Navigator 2.0 mit integrierter Java Virtual Machine (JVM)
- Aktuell
  - Java 13 Release September 2019
  - Java 14 Release März 2020
  - Java 15 Release September 2020
  - Java 16 Release März 2021
  - Java 17 Release September 2021
  - Java 18 geplanter Release März 2022
  - Release-Zyklus: 6 Monate (seit 2017)
- Java ist nicht nur reine Programmiersprache, bringt auch Plattformcharakter mit sich (über JVM).

# Java (3)



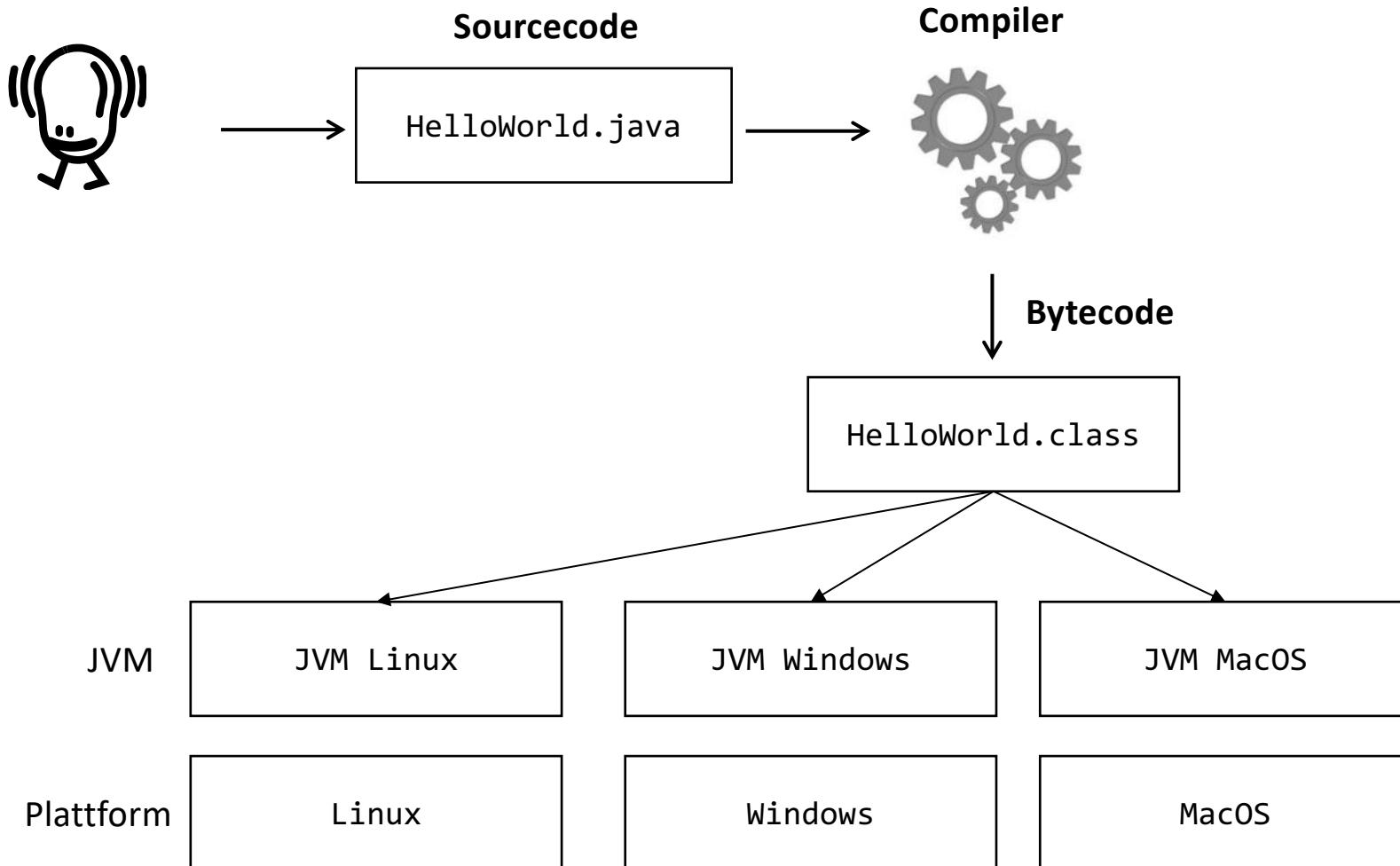
# Eigenschaften der Sprache Java

- Portierbar
- Objektorientiert
- Robust & Sicher
  - Strenges Typsystem
  - Behandlung von Ausnahmen (Exception Handling)
  - Automatische Speicherbereinigung (Garbage Collection)
- Nebenläufig (Threading)

# Erstellen/Ausführen von Programmen

- Java Compiler
  - Übersetzt den Programmtext in einen einfachen künstlichen Code (Bytecode)
  - Bytecode ist für alle Rechner/Betriebssysteme gleich.
- Java Virtual Machine (JVM)
  - "Virtueller, künstlicher" Computer
  - Dient als Schnittstelle zur Maschine und dem Betriebssystem.
  - Für jede Plattform existiert eine eigene JVM.
  - Liest den Bytecode und führt ihn aus (Interpretation!).
  - Mehr in der entsprechenden Vorlesung

# Hybrider Ansatz in Java



# Bytecode

- Plattform-unabhängig und meist sehr kompakt  
→ *write once, run anywhere*
- Enthält keine Anweisungen in Maschinensprache für den physischen Computer.
- Interpreter für Bytecode ist kleiner und schneller als ein Interpreter für den ursprünglichen Source-Code.
- Endbenutzer\*innen haben keinen Zugriff auf den Source-Code.
- Der Interpreter kann zusätzliche Überprüfungen durchführen (z.B. Arraygrenzen).

# Programm in Java

- In der Einführung in die Programmierung wurden verschiedene Grundelemente von Programmiersprachen behandelt, z.B.:
  - Datentypen, Variablen
  - Anweisungen
  - Zeiger
  - Modularisierung
- In Java, einer objektorientierten Sprache, sind alle Programmteile einer Klasse zugeordnet.
- Jedes Programm hat zumindest eine Klassendefinition mit mindestens einem Unterprogramm (Methode).
- Es muss eine „Start“-Methode geben - in Java ist dies die `main`-Methode.

# Hello World! (C und Java)

- HelloWorld.c

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

- HelloWorld.java

```
public class HelloWorld {

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

# Hello (Java) World!

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

- Eine Klasse wird durch das Schlüsselwort `class` vereinbart.
- Die Klasse erhält den Namen `HelloWorld`:
  - Der Code muss in einer Datei `HelloWorld.java` stehen.
  - Die durch das Kompilieren erhaltene Bytecode-Datei heißt `HelloWorld.class`.
- Die `main`-Methode ist der Startpunkt der Ausführung.
  - Die `main`-Methode muss immer `public` und `static` sein (wird später im Kapitel über Objektorientierung noch ausführlich erklärt).
  - Die `main`-Methode gibt nichts zurück, daher `void`.
  - Dem Programm (der `main`-Methode) können Parameter übergeben werden (wird im Kapitel über Arrays genauer erklärt).
- In der `main`-Methode können beliebige Anweisungen stehen.

# Übersetzen und Ausführen

- Auf der Kommandozeile (wenn entsprechende Programme installiert sind)
  - Übersetzen  
`javac HelloWorld.java`
  - Ausführen  
`java HelloWorld`
- Entwicklungsumgebungen (Beispiele)
  - Eclipse: <http://www.eclipse.org/>
  - IntelliJ: <https://www.jetbrains.com/idea/>
  - Visual Studio Code: <https://code.visualstudio.com/>

# Datentypen & Operatoren

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Datentypen

- Wiederholung
  - Alle Daten haben einen Typ (Datentyp)
  - Der Typ schränkt die Benutzung der Daten ein
  - Datentyp bestimmt:
    - Wertebereich
    - Operationen, die auf den Datentyp angewandt werden dürfen
- In Java existieren zwei Arten von Datentypen:
  - Primitive Datentypen
    - Nicht veränderbar oder ergänzbar
  - Referenztypen (Datentypdefinition durch Klassen)
    - Vorgegebene Klassen
    - Benutzerdefinierte Klassen
- Java ist streng typisiert → Datentyp und entsprechende Operationen sind während Programmlaufzeit unveränderlich

# Datentypen und Deklarationen

- Primitive Datentypen in Java
  - Ganze Zahlen (byte, short, char, int, long)
  - Gleitkommazahlen (float, double)
  - Logische Werte (boolean)
- Referenztypen
  - Zeichenketten (String)
  - Arrays
  - ...
- Für Variablen müssen Bezeichner und Datentyp angegeben werden
  - Variable mit Bezeichner counter, die ganze Zahlen aufnehmen kann

```
int counter;
```

# Ganze Zahlen (1)

- Wertebereich ist endlich
  - byte (n=1; 8 bit), short (n=2; 16 bit), int (n=4; 32 bit), long (n=8; 64 bit)
    - $[-2^{8n-1} \dots 2^{8n-1} - 1]$  → Zweierkomplement-Darstellung
    - Wertebereich ist asymmetrisch
  - char (16 bit; '\u0000' bis '\uffff' das entspricht 0 bis 65535)
- Operationen
  - Beliebige Typen: (Typ), ?:, ==, !=
  - Arithmetische Typen: +, -, \*, /, %, <, >, <=, >=, ++, --
  - Ganzzahl Typen: ~, <<, >>, >>>, &, ^, |

# Ganze Zahlen (2)

- Shift (<<, >>, >>>)
  - $n \ll s$ : Linksverschiebung der Bits von  $n$  um  $s$  Positionen
  - $n \gg s$ : Rechtsverschiebung der Bits von  $n$  um  $s$  Positionen mit Vorzeichen
  - $n \ggg s$ : Rechtsverschiebung der Bits von  $n$  um  $s$  Positionen ohne Vorzeichen
- Bitweises Exklusiv-ODER (^)
  - Korrespondierende Bits werden miteinander Exklusiv-ODER-verknüpft
- Division (/)
  - Division durch 0 nicht erlaubt
  - Ganzzahlige Division

```
int x = 5;           // x = 5
int y = (x / 3) * 3; // y = 3
```

# Ganze Zahlen – Überläufe

- Java zeigt Überläufe nicht an.
- Bei einem Überlauf wird mit fehlerhaften Werten weitergerechnet.
  - Bei additiven Operationen (+, -) kann über die Grenzen hinaus und wieder zurück gerechnet werden.
  - Bei den multiplikativen Operationen \* bzw. / und bei den Shift-Operationen (<<, >>, >>>) ist Vorsicht geboten!

Konstante MAX\_VALUE der  
Klasse Integer

```
int maxInt;
int overflow;
maxInt = Integer.MAX_VALUE; // maxInt = 2147483647

overflow = maxInt + 1;      // overflow = -2147483648 = Integer.MIN_VALUE
overflow = overflow - 1;    // overflow = 2147483647 = Integer.MAX_VALUE

overflow = maxInt * 2;      // overflow = -2
overflow = overflow / 2;   // overflow = -1
```

# Gleitkommazahlen (1)

- Gleitkommadarstellung
  - $x = \text{Mantisse} * \text{Basis}^{\text{Exponent}}$
  - Die Größe und die Genauigkeit sind endlich!
    - Nicht alle Zahlen können dargestellt werden (z.B. 0.1 → binär periodisch).
    - Vergleiche sind möglich – aber nicht immer unproblematisch!
- IEEE 754 Format:
  - float: 32 bit (1 bit Vorzeichen, 8 bit biased Exponent, 23 bit Mantisse)
  - double: 64 bit (1 bit Vorzeichen, 11 bit biased Exponent, 52 bit Mantisse)
- Operationen
  - Beliebige Typen: (Typ), ?:, ==, !=
  - Arithmetische Typen: +, -, \*, /, %, <, >, <=, >=, ++, --

# Gleitkommazahlen (2)

- Gleitkommazahlen können nicht nur positive und negative Zahlen darstellen.
  - Positive und negative Null
  - Positives und negatives Unendlich
  - Not-a-Number (NaN)
- Überläufe bzw. Unterläufe führen zu einem positiven bzw. negativen Unendlich.
- NaN-Werte werden verwendet, um Ergebnisse von ungültigen Operationen zu kennzeichnen.
- Abgesehen von NaN-Werten weisen Gleitkommazahlen eine totale Ordnung auf.

# Logische Werte

- Der Wertebereich von boolean umfasst 2 Werte
  - true und false
- Operationen
  - Beliebige Typen: (Typ), ? :, ==, !=
  - Logische Typen: !, &, ^, |, &&, ||
- Logische Ausdrücke bestimmen den Kontrollfluss von:
  - if-Anweisungen
  - while-Schleifen
  - do-while-Schleifen
  - for-Schleifen

# Logische Ausdrücke – Auswertung

- „*Short circuit evaluation*“ bei logischen Operatoren:
  - $x \&& y$ : ist  $x$  false, ist das Ergebnis false und  $y$  wird nicht ausgewertet
  - $x || y$ : ist  $x$  true, ist das Ergebnis true und  $y$  wird nicht ausgewertet
- Erhöht die Robustheit
  - Der Ausdruck  $y$  darf Code enthalten, der ungültig ist im Fall
    - $(x \&& y)$  bei  $x == \text{false}$
    - $(x || y)$  bei  $x == \text{true}$
  - Beispiel
    - $(x != 0 \&& y/x > 2)$  – keine Division durch 0 möglich!

# Zeichenketten (String)

- Strings werden zur Darstellung von Zeichenketten verwendet
- String ist kein primitiver Datentyp!
  - Ist eine Klasse
  - Wird noch ausführlich in der Vorlesung über Objektorientierung besprochen
- Deklaration (eine Möglichkeit)

```
String stringTest = "Text";
```
- Operationen
  - Beliebige Typen: (Typ), ?:, ==, !=
  - String Typen: +

```
String s1 = "Hello" + " " + "World!"; // "Hello World!"  
String s2 = "int: " + 7; // "int: 7"  
String s3 = "double: " + 3.14; // "double: 3.14"
```

# Ganzzahlenliterale

- Ganzzahlige Literale können im Dezimal-, Binären-, Oktal- oder Hexadezimalsystem angegeben werden.

Stellenwertsystem	Präfix	Ziffernmenge
Binärsystem	0b, 0B	0, 1
Oktalsystem	0	0 - 7
Dezimalsystem		0 - 9
Hexadezimalsystem	0x, 0X	0 - 9, a - f, A - F

- Ziffernfolgen können Unterstriche enthalten
- Ein Ganzzahlenliteral ist vom Typ long, sofern das Suffix L oder l verwendet wird, ansonsten ist der Typ int
- Beispiele:
  - int Literale: 0, 12, 0327, 0b101011, 0xCafe, 1669
  - long Literale: -5L, 2\_147\_483\_648L, 1669L

# Gleitkommaliterale

- Gleitkommaliterale können im Dezimal- oder Hexadezimalsystem angegeben werden.
  - Sie bestehen aus einem Vorkomma, einem Punkt, einem Nachkomma, einem Exponenten und einem Suffix.
  - Suffix f oder F für float, d oder D für double (default).
  - Um ein Gleitkommaliteral von einem integralen Literal unterscheiden zu können, muss mindestens der Dezimalpunkt, der Exponent oder das Suffix vorhanden sein.
- Ziffernfolgen können Unterstriche enthalten
- Beispiele:
  - float Literale: 2.5f, 2.f, .5f, 0f, 3e1f, 5.2546e+13f
  - double Literale: 2.5, 2., .5, 0.0, 0x278.b5p+4, 10\_123.312\_5, 1e55, 4d

# Zeichenliterale

- Ein Zeichenliteral umschließt mit einfachen Anführungszeichen ein Zeichen oder eine Escapesequenz.
- Ist immer vom Typ char.
- Ist UTF-16 kodiert.
- Beispiele:
  - 'A', 'a', '%', '!', '5', 'ä'
  - '\n', '\t', '\\', '\u0041', '\u0061', '\101', '\u00e4'

# String-Literale

- Ein String-Literal umschließt mit doppelten Anführungszeichen eine Folge von Zeichen und Escapesequenzen.
- Ist immer vom Typ String.
- Kann sich nicht über mehr als eine Code-Zeilen erstrecken.
- Ein langes String-Literal kann mithilfe der Stringkonkatenation (+) in kürzere String-Literale aufgeteilt werden.
- Beispiele:
  - "", "Hello World!", "Universit\u00e4t"

# Textblöcke

- Ein Textblock umschließt innerhalb von Begrenzern eine Folge von Zeichen und Escapesequenzen.
  - Start (opening delimiter): Drei Anführungszeichen, gefolgt von beliebig vielen Whitespace-Zeichen und einem Zeilenumbruch
  - Ende (closing delimiter): Drei doppelte Anführungszeichen
- Ist immer vom Typ String.
- Erstreckt sich mindestens über zwei Code-Zeilen.
- Der Inhalt eines Textblocks muss nicht ganz links beginnen. Gemeinsame Einrückungen (incidental whitespaces) werden im resultierenden String nicht berücksichtigt.

```
String string_literal = "This is the first line of a string.\n" +
    "This is the second line of a string.';

String text_block = """
    This is the first line of a string.
    This is the second line of a string.""";
```

# null-Literal

- Das null-Literal repräsentiert die null-Referenz.
- Die null-Referenz kann jedem Referenztyp zugewiesen werden.
- Die null-Referenz kann nicht in einen primitiven Typ umgewandelt werden.
- Wird versucht über null auf eine Variable oder Methode zuzugreifen, tritt ein Laufzeitfehler (`NullPointerException`) auf.

# Unveränderliche Werte

- Variablen können als `final` deklariert werden. Diesen Variablen kann nur einmal ein Wert zugewiesen werden.
- Finale lokale Variablen
  - Deklaration (danach keine Zuweisung möglich!)  
`final int x = 10;`
  - Aufgeschobene Initialisierung  
`final double y;`  
...  
`y = 20.2;`
- Konstanten
  - Ein statisches, finales Feld, welches einen primitiven Datentyp oder einen unveränderlichen Referenztyp hat.  
`static final int MINIMUM_LENGTH = 30;`

# Bezeichner (1)

- Für Variablen (und Konstanten), Methoden, Klassen etc. werden Bezeichner vergeben.
- Ein Bezeichner ist eine Folge von Buchstaben, Ziffern und den Symbolen \_ (Unterstrich) und \$ (Dollar).
- Die Buchstaben dürfen aus dem Unicode-Zeichensatz entstammen.
- Das erste Zeichen des Bezeichners darf keine Ziffer sein.
- Der Bezeichner darf kein reserviertes Schlüsselwort, Boolean-Literal oder das null-Literal sein.
- Empfehlung: *englische Bezeichner*

# Bezeichner (2)

- In Java gibt es viele etablierte Namenskonventionen.
- Diese Namenskonventionen sollten nur im Ausnahmefall mit einer entsprechenden Begründung nicht eingehalten werden.
- Felder
  - Bestehen aus einem Wort oder mehreren Wörtern.
  - Keine ungewöhnlichen Abkürzungen.
  - Bei Konstanten werden alle Buchstaben als Großbuchstaben geschrieben, wobei Wörter durch Unterstriche getrennt werden (SCREAMING\_SNAKE\_CASE).
  - Felder, die keine Konstanten sind, werden klein geschrieben, wobei der erste Buchstabe jedes Folgewort groß geschrieben wird (camelCase).
- Lokale Variablen
  - Werden in der camelCase Schreibweise geschrieben.
  - Dürfen Abkürzung enthalten und aus einzelnen Buchstaben bestehen.

# Bezeichner (3)

```
public static List<Integer> getFlaggedCells(List<Integer> l) {  
    List<Integer> r = new ArrayList<Integer>();  
    for (Integer x : l) {  
        if (x == 4) {  
            r.add(x);  
        }  
    }  
    return r;  
}
```





# Avoid Single-Letter Names

```
public static List<Integer> getFlaggedCells(List<Integer> gameBoard) {  
    List<Integer> flaggedCells = new ArrayList<Integer>();  
    for (Integer cell : gameBoard) {  
        if (cell == 4) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```



- Vorher:
  - Leser muss Bedeutung (Semantik) der Variablen selbst herausfinden.
  - Variablen geben keinerlei Auskunft über Inhalt.
- Nachher: lesbarer Code

# Bezeichner (4)

```
public class Bankaccount {  
    private double blnc;  
  
    public void modifycurrentBlnc(double Amnt) {  
        blnc += Amnt;  
    }  
  
    public double getCurrentBlnc() {  
        return blnc;  
    }  
}
```



# < > Use Java Naming Conventions

```
public class BankAccount {  
    private double balance;  
  
    public void modifyCurrentBalance(double amnt) {  
        balance += amnt;  
    }  
  
    public double getCurrentBalance() {  
        return balance;  
    }  
}
```



- Vorher:
  - Bezeichner-Schreibweise folgt keinem gemeinsamen Schema (Groß-Kleinschreibung, etc.)
- Nachher:
  - Format der Bezeichner für Methodennamen, Variablen, etc. vereinheitlicht
  - Folgt den [Java Naming Convention](#)



# Avoid Abbreviations

```
public class BankAccount {  
    private double balance;  
  
    public void modifyCurrentBalance(double amount) {  
        balance += amount;  
    }  
  
    public double getCurrentBalance() {  
        return balance;  
    }  
}
```



- Vorher:
  - Leser muss Bedeutung (Semantik) der Variablen selbst herausfinden.
  - Lesen dauert länger, Code schwerer verständlich.
- Nachher: lesbarer Code



# Avoid Meaningless Terms

```
public class BankAccount {  
    private double balance;  
  
    public void modifyBalance(double amount) {  
        balance += amount;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
}
```



- Vorher:
  - Bezeichner beinhalten nutzlose Information
  - Bezeichner sind dadurch länger, aber nicht aussagekräftiger
- Nachher: lesbarer Code
  - Kürzere, aber trotzdem aussagekräftige Bezeichner

# Reservierte Schlüsselwörter

- Reservierte Schlüsselwörter dürfen nicht als Bezeichner verwendet werden:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto (*)	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const (*)	float	native	super	while
_ (*)				

(\*) reserviert, aber nicht benutzt

# Operatoren

- Operatoren werden mit Operanden zu Ausdrücken verknüpft.
- Komplexe nicht geklammerte Ausdrücke werden in Teilausdrücke aufgelöst durch:
  - Präzedenz oder Bindungsstärke
    - Regelt die implizite Klammerung bei Operatoren auf verschiedenen Stufen.
    - Beispiel:  $1 + 2 \cdot 3 = 1 + (2 \cdot 3)$
  - Links-/Rechtsassoziativität
    - Regelt die implizite Klammerung bei Operatoren auf gleicher Stufe.
    - Linksassoziativ:  $a \cdot b \cdot c = (a \cdot b) \cdot c$
    - Rechtsassoziativ:  $a \cdot b \cdot c = a \cdot (b \cdot c)$

# Operatorpräzedenz

Stärke	Präzedenzgruppen	Assoziativität
14	<code>++ (postfix), -- (postfix)</code>	links
13	<code>++ (präfix), -- (präfix), +(unär), -(unär), ~, !, (type)</code>	rechts
12	<code>*, /, %</code>	links
11	<code>+, -, +(String-Konkatenation)</code>	links
10	<code>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</code>	links
9	<code>&lt;, &lt;=, &gt;, &gt;=, instanceof</code>	links
8	<code>==, !=, == (Referenz), != (Referenz)</code>	links
7	<code>&amp; (bitweises UND), &amp; (logisches UND)</code>	links
6	<code>^ (bitweises XOR), ^ (logisches XOR)</code>	links
5	<code>  (bitweises ODER),   (logisches ODER)</code>	links
4	<code>&amp;&amp; (logisches UND mit Short-Circuit-Evaluation)</code>	links
3	<code>   (logisches ODER mit Short-Circuit-Evaluation)</code>	links
2	<code>? : (bedingte Auswertung)</code>	rechts
1	<code>=, +=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=, -&gt;</code>	rechts

# Typkompatibilität

- Ein Ausdruck, welcher ein Ergebnis erzeugt, hat einen Typ, der zur Compile-Zeit ermittelt werden kann.
- Der Typ eines Ausdrucks muss in Umwandlungskontexten kompatibel mit dem erwarteten Typ (Zieltyp) sein.
- In Umwandlungskontexten kann durch den Compiler eine implizite Typumwandlung erfolgen.
- Falls in einem Umwandlungskontext kein passender Typ ermittelt werden kann, kommt es zu einem Kompilierfehler.
- Beispiel

```
int x = 2.5;      // Fehler! Die Datentypen sind nicht kompatibel
double y = 10;    // Implizite Typumwandlung von int nach double
byte z = 3;       // Implizite Typumwandlung von int nach byte
```

# Umwandlungskontexte

- Zuweisungskontext (assignment context)
  - Zuweisung des Werts eines Ausdrucks an eine Variable
- Parameterkontext (invocation context)
  - Zuweisung des Werts eines aktuellen Parameters an den formalen Parameter
- String-Kontext (string context)
  - Stringkonkatenation, wenn ein Operand nicht den Typ String aufweist
- Cast-Kontext (casting context)
  - Cast-Ausdrücke
- Numerischer Kontext (numeric context)
  - Operanden von arithmetische Operatoren
  - Bedingungsoperator
  - switch-Ausdrücke
  - Arrayerzeugung und Arrayzugriff

# Typkonvertierungen (primitive Datentypen)

- Identitätskonvertierung ( $\approx$ )
- Erweiternde Konvertierung ( $\omega$ )
- Einschränkende Konvertierung ( $\eta$ )
- Erweiternde und einschränkende Konvertierung ( $\omega\eta$ )

von↓/nach→	byte	short	char	int	long	float	double	boolean
byte	≈	ω	ωη	ω	ω	ω	ω	-
short	η	≈	η	ω	ω	ω	ω	-
char	η	η	≈	ω	ω	ω	ω	-
int	η	η	η	≈	ω	ω	ω	-
long	η	η	η	η	≈	ω	ω	-
float	η	η	η	η	η	≈	ω	-
double	η	η	η	η	η	η	≈	-
boolean	-	-	-	-	-	-	-	≈

# Implizite Typanpassung (primitive Datentypen)

- Zuweisungskontext
  - Erweiternde Konvertierung
  - Einschränkende Konvertierung für konstante Ausdrücke vom Typ byte, short, char oder int
- Parameterkontext
  - Erweiternde Konvertierung
- String-Kontext
  - String-Konvertierung
- Numerischer Kontext
  - Einschränkende Konvertierung für konstante Ausdrücke beim Bedingungsoperator und bei switch Ausdrücken
  - Erweiternde Konvertierung

# Explizite Typanpassung (primitive Datentypen)

- Bei der expliziten Typanpassung wird mit einem Cast der Typ eines Ausdrucks explizit verändert.
- Form
  - (Zieltyp) Ausdruck
- Beispiel

```
int x = (int) 2.5;
```
- Mögliche Konvertierungen
  - Identitätskonvertierung
  - Erweiternde Konvertierung
  - Einschränkende Konvertierung
  - Erweiternde und einschränkende Konvertierung
- Achtung: kann zu Verlusten der Größenordnung und Präzision führen

# Anweisungen

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck

# Elementare Anweisungen

- Leere Anweisung  
;
- Ausdrucksanweisung  
**Ausdruck;**
  - Nur zielführend, wenn der Ausdruck einen Nebeneffekt hat
    - Zuweisung
    - Inkrement und Dekrement
    - Methodenaufruf und Instanzerzeugung
- return-Anweisung
  - Rücksprung zur aufrufenden Methode aus einer Methode
- Block
  - Zusammenfassung von Anweisungen, die nacheinander ausgeführt werden

```
{  
    Anweisung1;  
    Anweisung2;  
    ...  
}
```

# if-Anweisung

```
if (Bedingung1) {  
    Anweisungsfolge1;  
} else if (Bedingung2) {  
    Anweisungsfolge2;  
    ...  
} else {  
    AnweisungsfolgeX;  
}
```

- Ausführung
  - Die Bedingung (boolescher Ausdruck!) wird ausgewertet und dann entsprechend verzweigt.
  - Klammern (Block) können weggelassen werden, wenn nur eine Anweisung folgt (Achtung: geringere Lesbarkeit!).
  - Der else-Zweig bzw. else if-Zweige können weggelassen werden.

# Beispiel (if-Anweisung)

```
public static int computeGrade(int pointsReached) {  
    int grade;  
    if (pointsReached < 50) {  
        grade = 5;  
    } else if (pointsReached < 65) {  
        grade = 4;  
    } else if (pointsReached < 75) {  
        grade = 3;  
    } else if (pointsReached < 85) {  
        grade = 2;  
    } else {  
        grade = 1;  
    }  
    return grade;  
}
```

# Dangling else

- Bekanntes Problem bei der Programmierung
- Ein else-Zweig bei zwei if-Anweisungen  
→ Wohin gehört der else-Zweig?

```
if (counter < 5)
    if (counter % 2 == 0)
        System.out.println("Position 1");
else
    System.out.println("Position 2");
```



- Auflösung durch Compiler
  - else-Zweig gehört zum textuell letzten freien if im selben Block!
  - counter mit Wert 7 führt zu keiner Ausgabe.
  - counter mit Wert 3 führt zur Ausgabe "Position 2".



# Always Use Brackets

```
if (counter < 5) {  
    if (counter % 2 == 0) {  
        System.out.println("Position 1");  
    } else {  
        System.out.println("Position 2");  
    }  
}
```



- Vorher:
  - Abarbeitungsabfolge nicht klar
- Nachher:
  - Abarbeitung durch Klammerung klar
  - Gilt nicht nur für if-Blöcke, sondern für jegliche Anweisungen

# Bedingungsoperator

- Erlaubt den Wert eines Ausdrucks von einer Bedingung abhängig zu machen, ohne eine if-Anweisung zu verwenden.

- Form

Bedingung ? Ausdruck1 : Ausdruck2;

- Beispiel

```
int a, b;  
...  
int max = (a > b) ? a : b;
```

```
int a, b;  
...  
int max;  
if (a > b) {  
    max = a;  
} else {  
    max = b;  
}
```

# switch-Anweisung (1)

- Verzweigt den Kontrollfluss anhand des Wertes einer Bedingung zu einem oder mehreren Fällen (cases).
- Den Körper einer switch-Anweisung gibt es in den zwei Formen Anweisungsgruppen und switch-Regeln.

```
switch (Bedingung) {  
    case Wert1: {Anweisungenfolge1; break;}  
    case Wert2, Wert3: {Anweisungenfolge2; break;}  
    ...  
    default: {AnweisungenfolgeX; break;}  
}
```

```
switch (Bedingung) {  
    case Wert1 -> Ausdruck1;  
    case Wert2, Wert3 -> {Anweisungenfolge2;}  
    ...  
    default -> AusdruckX;  
}
```

# switch-Anweisung (2)

- Der Typ der Bedingung muss eine Ganzzahl (short, byte, int, char), ein Wrapper-Typ einer Ganzzahl, ein Aufzählungstyp (enum) oder ein String sein.
- Kann beliebig viele case-Lables enthalten.
- break kann beim switch mit Anweisungsgruppen verwendet werden und sorgt dafür, dass die Ausführung **nicht** in das folgende case-Label weiterspringt.
- Das default-Label ist optional und wird ausgeführt, wenn kein case-Label ausgeführt werden kann.
- Ein case-Label hat eine oder mehrere case-Konstanten, welche durch ein Komma getrennt werden.
- Jeder Wert eines case-Labels muss ein konstanter Ausdruck oder ein Bezeichner einer enum-Konstante sein.

# Beispiel (switch-Anweisung)

```
public static String getGradeDescription(int grade) {  
    String gradeString;  
    switch (grade) {  
        case 1:  
            gradeString = "Sehr Gut";  
        case 2:  
            gradeString = "Gut";  
            break;  
        case 3:  
            gradeString = "Befriedigend";  
            break;  
        case 4:  
            gradeString = "Genügend";  
            break;  
        default:  
            gradeString = "Nicht Genügend";  
            break;  
    }  
    return gradeString;  
}
```





# Avoid Switch Fallthrough

```
...  
case 1:  
    gradeString = "Sehr Gut";  
break;  
...
```



- Vorher:
  - Abarbeitungsabfolge für switch wird ohne break nicht eingehalten
- Nachher:
  - Abarbeitung wie gedacht

# switch-Ausdruck (1)

- Verzweigt den Kontrollfluss anhand des Wertes einer Bedingung zu einem Fall.
- Den Körper eines switch-Ausdrucks gibt es in den zwei Formen Anweisungsgruppen und switch-Regeln.

```
switch (Bedingung) {  
    case Wert1: {Anweisungenfolge1; yield Resultat1;}  
    case Wert2, Wert3: {Anweisungenfolge2; yield Resultat2;}  
    ...  
    default: {AnweisungenfolgeX; yield ResultatX;}  
}
```

```
switch (Bedingung) {  
    case Wert1 -> Ausdruck1;  
    case Wert2, Wert3 -> {Anweisungenfolge2; yield Resultat2;}  
    ...  
    default -> AusdruckX;  
}
```

# switch-Ausdruck (2)

- Jeder Fall muss ein Ergebnis für das Resultat des switch-Ausdrucks liefern.
  - Switch-Regel:
    - Sofern ein Block verwendet wird, kann mit dem yield-Ausdruck ein Ergebnis bereitgestellt werden.
    - Wird eine Ausdruck verwendet, ist der Ausdruck das Ergebnis.
  - Anweisungsgruppen:
    - Das Ergebnis kann mit der yield-Anweisung bereitgestellt werden.
- Alle möglichen Werte der Bedingung müssen behandelt werden.

# Beispiel (switch-Ausdruck)

```
public static String getGradeDescription(int grade) {  
    return switch (grade) {  
        case 1 -> "Sehr Gut";  
        case 2 -> "Gut";  
        case 3 -> "Befriedigend";  
        case 4 -> "Genügend";  
        default -> "Nicht Genügend";  
    };  
}
```

# Schleifen

- Anweisungsfolgen beliebig oft wiederholen
- while-, do-while- und for-Schleifen bestehen aus
  - Schleifenkörper
  - Abbruchbedingung

## **while**

- Zuerst Abbruchbedingung abfragen
- Dann Schleifenkörper ausführen

## **do-while**

- Zuerst Schleifenkörper ausführen
- Dann Abbruchbedingung abfragen

## **for**

- Geschlossene Schleife
- Vorbereitung (Initialisierung) und Fortschalten ist Teil der Schleifenkonstruktion

# while-Schleife

- Form

```
while (Bedingung) {  
    Anweisungsfolge;  
}
```

- Ablauf

1. Die Bedingung (boolescher Ausdruck) wird ausgewertet.
2. Ist sie wahr:
  - Der Schleifenkörper (Anweisungsfolge) wird ausgeführt.
  - Nach der Ausführung geht es wieder bei 1. weiter.
3. Wenn die Bedingung falsch ist, dann wird die nächste Anweisung nach der Schleife ausgeführt.

# Beispiel (while-Schleife)

```
int[] toSum = {1, 2, 3, 6, 10, 12};  
int sum = 0;  
int i = 0;  
while (i < toSum.length) {  
    sum += toSum[i];  
    ++i;  
}
```

Die Länge eines Arrays kann über das Datenelement `length`, welches jedes Array besitzt, ermittelt werden.

# do-while-Schleife

- Form

```
do {  
    Anweisungsfolge;  
} while (Bedingung);
```

- Ablauf

1. Der Schleifenkörper (Anweisungsfolge) wird ausgeführt.
2. Die Bedingung (boolescher Ausdruck) wird ausgewertet.
3. Ist sie wahr, dann Sprung nach 1.
4. Ist sie falsch, dann wird die nächste Anweisung ausgeführt

# for-Schleife

- Allgemeine Form

```
for (Initialisierung; Bedingung; Inkrementierung) {  
    Anweisungsfolge;  
}
```

- Aufbau

- Initialisierung: Wird vor dem Betreten der Schleife ausgeführt.
- Bedingung: Wird immer vor der Ausführung der Schleife überprüft.
- Inkrementierung: Wird am Ende jedes Schleifendurchlaufs ausgeführt.

- Ablauf

1. Initialisierung wird ausgeführt.
2. Die Bedingung wird ausgewertet:
  1. Bedingung erfüllt (wahr)
    - Schleifenkörper (Anweisung bzw. Block) wird ausgeführt.
    - Inkrementierung wird ausgeführt.
    - Es geht wieder bei der Bedingung weiter.
  2. Bedingung nicht erfüllt: Die nächste Anweisung wird ausgeführt.

# Beispiel (for-Schleife)

```
int[] toSum = {1, 2, 3, 6, 10, 12};  
int sum = 0;  
for (int i = 0; i < toSum.length; ++i) {  
    sum += toSum[i];  
}
```

# Erweiterte for-Schleife

- Form

```
for (Datentyp Bezeichner: Ausdruck) {  
    Anweisungsfolge;  
}
```

- Erweiterte for-Schleife (foreach-Schleife)
  - Erleichtert den Umgang mit Arrays und listenartigen Datenstrukturen (z.B. Collections).
  - Wird in den entsprechenden Kapiteln über Arrays bzw. Collections noch genau besprochen.

# Beispiel (erweiterte for-Schleife)

```
int[] toSum = {1, 2, 3, 6, 10, 12};  
int sum = 0;  
for (int value : toSum) {  
    sum += value;  
}
```

# break und continue

- **break**
  - Führt zum sofortigen Ausstieg aus dem aktuellen Schleifendurchlauf (oder aus der switch-Anweisung).
  - Wird benutzt in: while-, do-while- und for-Schleifen sowie switch-Anweisungen
- **continue**
  - Überspringt die restlichen Anweisungen im aktuellen Schleifendurchlauf.
  - Wird benutzt in: while-, do-while- und for-Schleifen
- **Verwendung**
  - Bei switch-Anweisungen mit Anweisungsgruppen meist notwendig (break)!
  - In Schleifen nur sparsam verwenden!
    - Kann übermäßige Verschachtelung vermeiden!
    - Kann durch entsprechende Formulierung der Schleifenbedingung umgangen werden!

# Ausblick - Anweisungen

- assert
  - Mit der assert-Anweisung kann eine Zusicherung überprüft werden.
- throw
  - Mit einer throw-Anweisung kann eine Ausnahme geworfen werden.
- try
  - Mit einer try-Anweisung können Ausnahmen gefangen werden (Exception Handling).

# Einführung Methoden

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Unterprogramme

- Die Teile eines Programms können auch an einer anderen Stelle als Unterprogramm formuliert werden.
- Unterprogramme können in einer Anweisungsfolge aufgerufen werden.
  - Verzweigung zum Unterprogramm
  - Abarbeitung des Unterprogramms
  - Nach der Abarbeitung Rückkehr zur Aufrufstelle
  - Eventuell Ergebnis zurückgeben

# Unterprogramme in C bzw. Java

- In C spricht man nur von Funktionen.
- In Java spricht man nur von Methoden.
  - Statische Methoden
    - Gehören zur Klasse
    - Gekennzeichnet mit dem Schlüsselwort `static`
  - Objektbezogene Methoden
    - Werden bei der Einheit zu Objektorientierung besprochen!

# Methoden (1)

## Methode

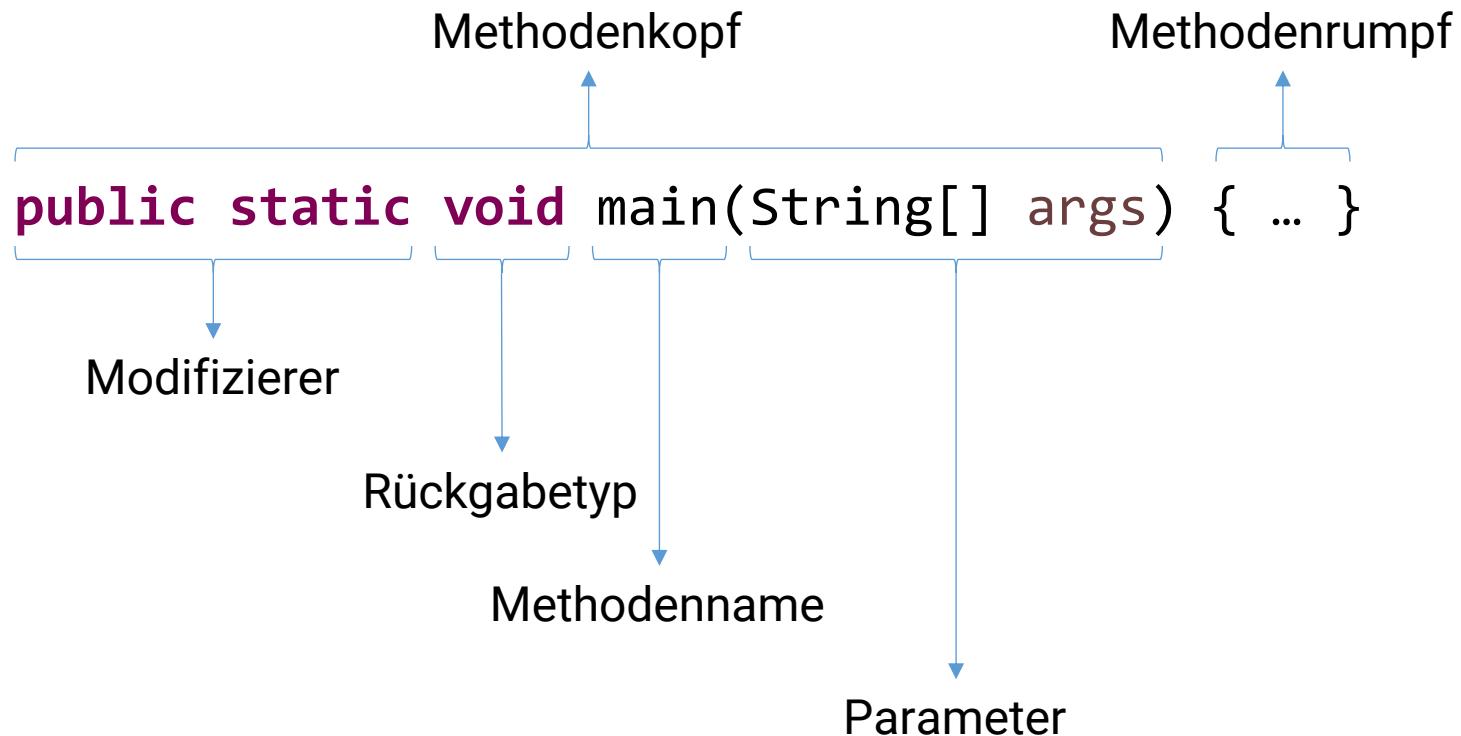
### Methodenkopf

- Rückgabetyp, Methodename und eventuell Parameter

### Methodenrumpf

- ist ein Block
- Enthält Variablendeklarationen und Anweisungen
- Kann auch leer sein (wird noch erklärt)

# Methoden (2)



# Rückgabewert

- Typ des Rückgabewerts steht vor dem Methodennamen.
- Methoden mit Rückgabewert
  - Rückgabewert wird durch eine return-Anweisung zurückgegeben.
  - Eine return-Anweisung muss in jedem Programmfpfad in der Methode vorhanden sein.
  - Der Typ des Ausdrucks, welcher in der return-Anweisung verwendet wird, muss zuweisungskompatibel mit dem Typ des Rückgabewerts sein.
- Methoden ohne Rückgabewert (Rückgabetyp void)
  - Benötigen keine return-Anweisung.
  - Durch eine leere return-Anweisung ( return; ) kann die Methode vorzeitig verlassen werden.

# Methodenname

- Der Name einer Methode sollte ausdrücken, was die Methode leistet.
- Er sollte mit einem Verb beginnen und klein geschrieben werden.
  - Methoden, die den Wert eines Felds abfragen bzw. setzen, sollten mit dem Wort get bzw. set beginnen.
  - Methoden, die einen boolean-Wert zurückgeben, sollten mit dem Wort is beginnen.
  - Methoden, die ein Objekt in einen anderen Typ bzw. ein anderes Format F umwandeln, sollten als toF bezeichnet werden.
- Falls der Name aus mehreren Worten besteht, sollte der Anfangsbuchstabe jedes Folgewortes groß geschrieben werden.
- Beispiele
  - add, append
  - getArea, setArea
  - isEmpty, isBlank
  - toString, toArray, toLowerCase

# Parameter

## Formale Parameter

- Werden im Methodenkopf deklariert
- Werden innerhalb der Methode wie lokale Variablen behandelt

## Aktuelle Parameter

- Beim Aufruf der Methode wird jeder formale Parameter durch je einen aktuellen Parameter initialisiert

# Parameterübergabe-Mechanismen

- Der Parameterübergabe-Mechanismus bestimmt wie die Parameter vom Methodenaufruf an die aufgerufene Methode übergeben werden.
- Beispiele für Parameterübergabe-Mechanismen
  - Call-by-value
    - Formaler Parameter wird mit dem Wert des aktuellen Parameter initialisiert.
    - Änderungen am formalen Parameter ändern den aktuellen Parameter nicht.
  - Call-by-reference
    - Eine Referenz wird implizit übergeben.
    - Änderungen am formalen Parameter ändern den aktuellen Parameter.
    - Z.B in C simuliert durch das Übergeben von Zeigern.

# Parameterübergabe in Java

- In Java wird nur call-by-value verwendet:
  - Für primitive Datentypen wird eine Kopie des Wertes übergeben.
  - Für Referenztypen wird eine Kopie der Referenz auf das Objekt übergeben.
- Beim Aufruf einer Methode werden die aktuellen an die formalen Parameter durch folgende Aktionen übergeben:
  - Die Ausdrücke der aktuellen Parameter werden berechnet.
  - Der Wert dieser Ausdrücke wird den entsprechenden formalen Parametern zugewiesen.
    - Die Typkompatibilität für den Parameterkontext muss eingehalten werden!
  - Durch die Zuweisung wird eine **Kopie** erzeugt!

# Beispiel (Primitiver Typ)

```
public class PrimitiveTypesMethodApplication {  
    public static int pow2(int base) {  
        base = base * base;  
        return base;  
    }  
  
    public static void main(String[] args) {  
        int value = 4;  
        System.out.println(value);  
        System.out.println(pow2(value));  
        System.out.println(value);  
    }  
}
```

formaler Parameter base

aktueller Parameter value

Ausgabe:

4  
16  
4

# Beispiel (Referenztyp)

```
public class ReferenceTypeMethodApplication {  
    public static void pow2(int[] bases) {  
        for (int i = 0; i < bases.length; ++i) {  
            bases[i] = bases[i] * bases[i];  
        }  
    }  
  
    public static void main(String[] args) {  
        int[] values = {2, 4, 10};  
        System.out.println(values[1]);  
        pow2(values);  
        System.out.println(values[1]);  
    }  
}
```

Ausgabe:

4  
16

# Lokale Variablen

- Methoden können neben Anweisungen auch lokale Deklarationen enthalten.

```
public static void method() {  
    int x;  
    short y;  
    ...  
}
```

- Deklariert die zwei lokalen Variablen x und y
  - Dürfen nur in dieser Methode verwendet werden
  - Beim Aufruf wird der Speicherplatz angelegt
  - Am Ende der Methode wird der Speicherplatz wieder freigegeben.
- Jede lokale Variable muss initialisiert werden, bevor auf den Wert der Variable zugegriffen wird (definite assignment).

# Klassenvariablen

- Variablen können auch außerhalb von Methoden deklariert werden.
- Werden diese Variablen mit dem Keyword `static` gekennzeichnet, wird von Klassenvariablen gesprochen.
- Grundregel: Variablen immer möglichst lokal deklarieren!

```
public class ClassVariables {  
    static int a;  
    static double b;  
    public static void method() {...}  
    ...  
}
```

- a und b sind Klassenvariablen
  - Können in allen Methoden der Klasse `ClassVariables` benutzt werden
  - Werte existieren über Methodenaufrufe hinweg.
  - Näheres dazu in VO zu Objektorientierung.

# Lebensdauer

- Zeitintervall, in dem die Variable zur Laufzeit existiert.
- Lokale Variablen
  - werden angelegt, wenn ihr Block betreten wird
  - werden freigegeben, wenn ihr Block verlassen wird
- Klassenvariablen
  - werden angelegt, wenn die Klasse geladen wird
  - werden freigegeben, wenn die Klasse entladen wird

# Sichtbarkeit

- Programmstück, in dem auf die Bezeichner zugegriffen werden kann.
- Bezeichner in einer Klasse (z.B. Klassenvariablen, Methoden)
  - Ist in der ganzen Klasse gültig
  - Variablen können in Methoden neu deklariert werden. Durch die neue Deklaration wird die ursprüngliche Variable überdeckt.
- Bezeichner in einem Block (z.B. lokale Variablen)
  - Gültig von der Deklaration bis zum Ende des Blocks, in dem er deklariert wurde.
  - Blöcke können verschachtelt werden
  - In einem geschachtelten Block kann aber keine erneute Deklaration erfolgen!

# Überladen von Methoden

- In Java können Methoden überladen (engl. overloading) werden.
- Eine Methode ist überladen, wenn Methoden einer Klasse mit unterschiedlichen Parameterlisten den selben Namen haben.
  - Unterschiedliche Typen
  - Unterschiedliche Reihenfolge der Typen
  - Unterschiedliche Anzahl der Typen
- Basierend auf den Typen der Parameter wird beim Aufruf einer überladenen Methode vom Compiler die richtige Methode ausgewählt.
- Überladen gibt es auch bei Operatoren:
  - + hat unterschiedliche Bedeutung für int, double, String ....

# Signatur einer Methode in Java



- Alle Methoden einer Klasse müssen unterschiedliche Signaturen haben.
- Methodename und Liste der Typen der formalen Parameter (wenn vorhanden) bilden die **Signatur** einer Methode!
  - **Rückgabetyp gehört nicht dazu!**
  - Die Namen der Parameter spielen keine Rolle, nur die Typen und ihre Reihenfolge!
  - Beispiele:

```
public static int method1(int param1) {...}
```

Signatur: method1(int)

```
public static void method2(double param1, char param2) {...}
```

Signatur: method2(double, char)

# Beispiel (Überladen)

```
public class OverloadingApplication {  
    public static void myPrint(int i) {  
        System.out.println("myPrint(int): " + i);  
    }  
    public static void myPrint(double d) {  
        System.out.println("myPrint(double): " + d);  
    }  
    public static void main(String[] args) {  
        myPrint(10);  
        myPrint(10.0);  
    }  
}
```

Ausgabe:

```
myPrint(int): 10  
myPrint(double): 10.0
```

# Auflösen eines Methodenaufrufs

- Für das Bestimmen der Methode, welche durch einen Methodenaufruf aufgerufen wird, werden durch den Compiler drei Schritte durchgeführt.
  1. Basierend auf dem Methodenaufruf muss bestimmt werden in welchem Typ nach dieser Methode gesucht wird.
  2. Auswahl der Methode, welche verfügbar und anwendbar ist.
    - Sofern mehrere überladene Methoden anwendbar sind, wird durch den Compiler die spezifischste Methode ausgewählt.
    - Spezifischste Methode
      - Intuition: Eine Methode m1 ist spezifischer als die Methode m2, wenn jeder Aufruf der Methode m1 durch m2 behandelt werden könnte.
      - Wird durch den Compiler keine spezifischste Methode gefunden, ist der Aufruf mehrdeutig und es kommt zu einem Kompilierfehler.
  3. Überprüfung, ob die ausgewählte Methode passend ist.

# Beispiel (Auswahl überladener Methoden)

```
public class MostSpecificMethodApplication {  
    public static void myPrint(short s) {  
        System.out.println("myPrint(short): " + s);  
    }  
    public static void myPrint(int i) {  
        System.out.println("myPrint(int): " + i);  
    }  
    public static void myPrint(double d) {  
        System.out.println("myPrint(double): " + d);  
    }  
    public static void main(String[] args) {  
        byte b = 5;  
        myPrint(b);  
        myPrint('A');  
    }  
}
```

Ausgabe:  
myPrint(short): 5  
myPrint(int): 65

# Beispiel (Mehrdeutiger Methodenaufruf)

```
public class AmbiguousMethodsApplication {  
    public static void myPrint(float f, double d) {  
        System.out.println("myPrint(float, double): " + f + ", " + d);  
    }  
    public static void myPrint(double d, float f) {  
        System.out.println("myPrint(double, float): " + d + ", " + f);  
    }  
    public static void main(String[] args) {  
        myPrint(4.2f, 5.5f);  
    }  
}
```



## Ausgabe des Compilers:

```
java: reference to myPrint is ambiguous  
both method myPrint(float,double) in AmbiguousMethodsApplication and  
method myPrint(double,float) in AmbiguousMethodsApplication match
```

# Beispiel (Rückgabetyp)

```
public class IncompatibleTypesApplication {  
    public static String method(int i) {  
        return i + "";  
    }  
    public static int method(long l) {  
        return l;  
    }  
    public static void main(String[] args) {  
        int result = method(5);  
    }  
}
```



Ausgabe des Compilers:

java: incompatible types: java.lang.String cannot be converted to int

# Vordefinierte statische Methoden

- Java bietet etliche statische Methoden (Klassenmethoden) in speziellen Klassen an.
- Typischer Aufruf
  - `ClassName.MethodName(Parameter)`
  - Beispiel  
`Math.sqrt(10.0);`
- Manchmal auch in der Form
  - `ClassName.FieldName.MethodName(Parameter)`
  - Beispiel  
`System.out.println(x);`

# Beispiele für Klassen

- **System**

- `System.out.println()` etc.

- **Math**

- `Math.random()`, `Math.sqrt(10.0)` etc.

- **Arrays**

- Hilfsklasse für Arrays

- Nützliche überladene Methoden (hier für int):

- `binarySearch(int[] a, int key)`
- `equals(int[] a, int[] a2)`
- `deepEquals(Object[] a1, Object[] a2)`
- `fill(int[] a, int val)`
- `sort(int[] a)`

# Verwendung von Methoden anderer Klassen

- Bestimmte Klassen bzw. Pakete werden automatisch geladen.
  - Die Klassen aus `java.lang` werden automatisch importiert (z.B. Klassen `Math` und `String`)
- Andere Klassen bzw. Pakete müssen importiert oder mit dem voll qualifizierten Namen angesprochen werden.
  - Voll qualifizierter Name (bei Verwendung)  
`java.util.Arrays.sort(x);`
  - Import-Deklaration (am Anfang der Datei)  
`import java.util.Arrays;  
import java.util.*; // all classes from java.util  
import static java.lang.Math.min; // static import  
import static java.lang.Math.*; // all methods`

# Dokumentation der Klassen

- Startpunkt (Java 17)
  - <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>
- Bereitgestellte Informationen (Beispiele)
  - In welchem Paket befindet sich eine Klasse?
    - Die einzelnen Klassen eines Pakets können ausgewählt werden.
  - Für eine Klasse
    - Beschreibung der Klasse
    - Liste der Methoden
  - Für jede Methode
    - Kurzbeschreibung
    - Längere Beschreibung (Parameter, Ausnahmen etc.)

# Arrays

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Arrays

- Zusammenfassung gleichartiger Elemente
- Im Array können primitive Datentypen oder Referenztypen verwaltet werden.
- Arrayvariablen sind in Java **Referenzvariablen** (Referenz auf Speicherplatz eines Objektes).
- Arrays haben eine fixe Länge.
  - Nach dem Anlegen kann die Länge nicht mehr verändert werden.
  - Für Container mit veränderbarer Größe existieren in Java spezielle Klassen!
  - Die Arraylänge kann durch die finale Objektvariable `length` ermittelt werden.
- Jedem Element ist ein Index vom Typ `int` zugewiesen.
  - Indexzählung beginnt bei 0!
  - Indexzählung geht bis Länge-1!
- Eindimensionale Arrays enthalten Elemente vom Elementtyp.
- Mehrdimensionale Arrays enthalten weitere Arrays!

# Deklaration von Arrays

```
int array[], x, y, testArray[]; // arrays and variables  
int[] array, testArray; // arrays
```

- Deklaration
  - Es werden nur die Referenzvariablen angelegt.
  - Es wird noch kein Speicherplatz reserviert.
- Best practice
  - Deklarationen aufteilen
  - Möglichst nur eine Deklaration pro Zeile (Lesbarkeit)

# Anlegen von Arrays – ohne Initialisierung

```
// Anlegen bei Deklaration  
int[] array = new int[10];
```

```
// Späteres Anlegen  
int[] array;  
...  
array = new int[10];
```

- Die Größe des Arrays kann mit `array.length` abgefragt werden.
- Werte der Elemente werden zunächst mit Standard-Werten belegt (z.B. 0 bei Integer-Array).

# Anlegen von Arrays – mit Initialisierung

```
int[] array = {3, 4, 5, 6, 7};  
  
// or  
  
int[] array;  
...  
array = new int[]{3, 4, 5, 6, 7};
```

Das Array hat die Länge 5 und enthält die Elemente 3, 4, 5, 6, 7.

# Arrayindex

- Arrayelemente werden über einen Index angesprochen.
- Der Index muss nicht unbedingt eine Konstante sein, er kann auch ein beliebiger Ausdruck sein.
- Der Typ des Indexausdrucks muss aber int sein (und kann daher auch short, byte oder char sein).
- Der Indexbereich ist 0 .. Länge des Arrays-1.  
`int[] array = new int[10];`
- Zugriff
  - Indexwert in eckigen Klammern (z.B. array[3])
- Indexwert muss gültig sein!
  - **Sonst gibt es einen Laufzeitfehler, d.h. das Programm wird sofort abgebrochen!**

# Beispiel (main-Argumente)

```
public class PrintArgumentsApplication {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; ++i) {  
            System.out.println(i + ": " + args[i]);  
        }  
    }  
}
```

```
java PrintArgumentsApplication hello again
```

Ausgabe:  
0: hello  
1: again

# Zuweisung von Arrays

- Arrayvariable ist eine Referenz auf das eigentliche Array.
- Einer Arrayvariable vom Typ x kann immer nur ein Array vom Typ x zugewiesen werden.
- Die Länge kann aber variieren.
- Bei der Zuweisung wird nur die Referenz (Adresse) kopiert, **nicht** der Inhalt!

```
int[] x = new int[10];
int[] y = x;
y[1] = 7;
System.out.println(y[0] + " " + x[0]);
System.out.println(y[1] + " " + x[1]);
```

Ausgabe:

```
0 0
7 7
```

# Freigabe des Arrayspeichers

- Freigabe des Speicherplatzes erfolgt durch das System.
  - Speicher muss nicht explizit freigegeben werden.
  - Nicht benutzter Speicher wird zur Laufzeit automatisch eingesammelt und wieder freigegeben.
  - Garbage-Collector (siehe Vorlesung über JVM).
- **Voraussetzung: Array wird nicht mehr referenziert!**
- Näher erklärt in Vorlesung über JVM

# Mehrdimensionale Arrays

- Arrays können beliebig viele Dimensionen haben.

```
// Example: 2-dimensional array - matrix
int[][] matrix1 = new int[3][3];
int[][] matrix2 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
// access
int x = matrix2[1][2];
```

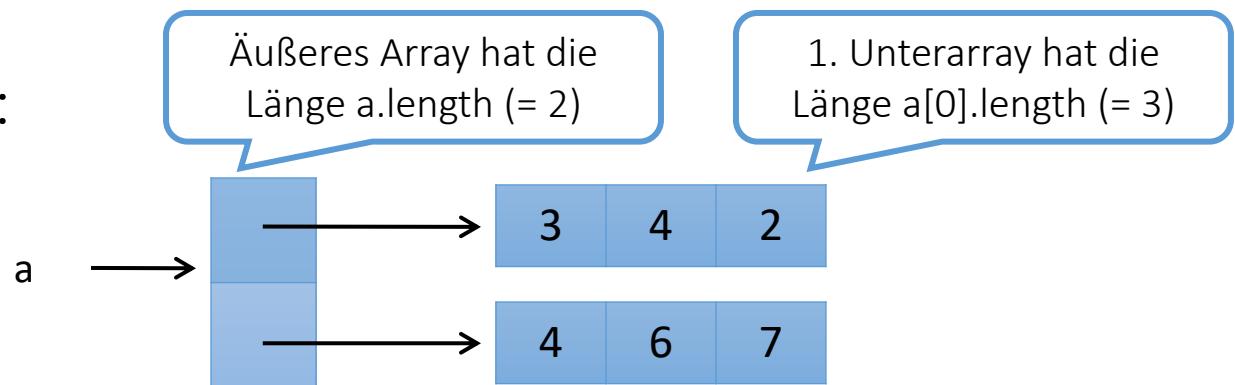
- Konzeptionelle Sicht:

```
int[][] a = {{3, 4, 2}, {4, 6, 7}};
```

a →

3	4	2
4	6	7

- „Realität“ in Java:



# Weitere Aspekte (1)

- Einzelne Dimensionen können offen bleiben (nur am Ende) und später initialisiert werden.

```
int[][] gameBoard = new int[4][];  
for (int i = 0; i < gameBoard.length; ++i) {  
    gameBoard[i] = new int[4];  
}
```

- Es können auch nicht rechteckige Arrays (jagged arrays) erzeugt werden:

```
int[][] irregularGameBoard = new int[2][];  
for (int i = 0; i < irregularGameBoard.length; ++i) {  
    irregularGameBoard[i] = new int[i + 3];  
}
```

# Weitere Aspekte (2)

- Object ist die direkte Oberklasse von jedem Array-Typ.
- Mit den Operatoren == und != wird überprüft, ob zwei Variablen auf das selbe Array-Objekt verweisen.

```
int[] values1 = {1, 5, 3};  
int[] values2 = {1, 5, 3};  
System.out.println(values1 == values2); // false
```

- Die erweiterte for-Schleife kann für das Iterieren über die Komponenten eines Arrays verwendet werden.

```
public class PrintArgumentsApplicationForeach {  
    public static void main(String[] args) {  
        for (String arg : args) {  
            System.out.println(arg);  
        }  
    }  
}
```



# Favor For-Each over For

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



```
for (String arg : args) {  
    System.out.println(arg);  
}
```



- Vorher:
  - Indexvariable wird nur für den Zugriff auf das nächste Element verwendet, sonst keine Funktion.
  - Indexvariable kann jederzeit verändert werden
  - Abbruchbedingung könnte falsch gewählt werden (IndexOutOfBoundsException)
- Nachher:
  - Jedes Element in der Datenstruktur wird verarbeitet.
  - Keinerlei Manipulationsmöglichkeiten

# Erweiterte for-Schleife

- Der Compiler setzt die erweiterte for-Schleife durch eine for-Schleife um.
- Im Vergleich zur klassischen for-Schleife gibt es beim Verwenden der erweiterten for-Schleife für Arrays einige Einschränkungen:
  - Ein Start- oder Endindex kann nicht gesetzt werden.
  - Das Array wird immer vom ersten bis zum letzten Index durchlaufen.
  - Der Index wird in jeder Iteration um 1 erhöht.
  - Der Index ist nicht sichtbar und kann nicht verwendet werden.
  - Die Komponenten im Array können gelesen werden, allerdings kann der Wert nicht ersetzt werden.
- Sofern für den Anwendungsfall diese Einschränkungen zu tragen kommen, kann beispielsweise die for-Schleife Abhilfe bieten.

# Kopieren von Arrays

- `System.arraycopy(src, srcPos, dest, destPos, length)`
  - Kopiert length-Elemente des Arrays src ab der Stelle srcPos in das Array dest ab der Position destPos.
  - Kann verwendet werden, um Elemente eines Arrays zu verschieben.
- `Arrays.copyOf(original, newLength)`
  - Erstellt ein neues Array und kopiert die ersten newLength-Elemente des Arrays original in das neue Array.
- `clone()`
  - Arrays bieten die Methode `clone()` an, um eine flache Kopie des Arrays zu erzeugen.

```
int[] original = {1, 5, 3};  
int[] copy1 = original.clone();  
int[] copy2 = Arrays.copyOf(original, original.length);  
int[] copy3 = new int[original.length];  
System.arraycopy(original, 0, copy3, 0, original.length);
```

# Arrays-Klasse

- Die Klasse Arrays ist Teil des Java Collections Framework.
- Stellt statische Methoden für die Manipulation von Arrays bereit.
- Beispiele:
  - `binarySearch` – zum Suchen in einem sortierten Array
  - `equals` – zum Überprüfen ob zwei Arrays gleich sind
  - `fill` – zum Füllen von Arrays
  - `hashCode` – zum Berechnen eines Hash
  - `sort` – zum Sortieren der Array-Elemente
  - `toString` – zum Erzeugen einer String-Präsentation des Array-Inhaltes
- Bei mehrdimensionalen Arrays ist bei den Methoden `equals`, `hashCode` und `toString` Vorsicht geboten.
- Weitere Details siehe [Dokumentation](#)

# Beispiel Arrays.toString

```
String[][] array1 = {{ "a", "b"}, {"c", "d"}};  
String[][] array2 = {{ "a", "b"}, {"c", "d"}};
```

```
System.out.println(Arrays.toString(array1));  
System.out.println(Arrays.toString(array2));
```

```
System.out.println(Arrays.deepToString(array1));  
System.out.println(Arrays.deepToString(array2));
```

Ausgabe:

```
[[Ljava.lang.String;@a3db808, [Ljava.lang.String;@4c56ea0f]  
[[Ljava.lang.String;@87f99898, [Ljava.lang.String;@356f015d]  
[[a, b], [c, d]]  
[[a, b], [c, d]]
```

# Beispiel Arrays.equals

```
String[][] array1 = {{ "a", "b"}, {"c", "d"}};
String[][] array2 = {{ "a", "b"}, {"c", "d"}};

System.out.println(Arrays.equals(array1, array2));
System.out.println(Arrays.deepEquals(array1, array2));
```

Ausgabe:

false

true

# Arrays & Strings

- Weder Strings noch Arrays sind nullterminiert!
- Strings sind keine Arrays!
- Die Methode `toCharArray` der Klasse `String` kann verwendet werden, um ein `String` in ein char-Array umzuwandeln.

```
String text = "hello";
char[] letters = text.toCharArray();
Arrays.sort(letters);
System.out.println(letters);
```

Ausgabe:  
ehllo

# Objektorientierung

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck

# Motivation

- Was wurde bisher besprochen?
  - Datentypen, Anweisungen, Arrays, Grundlagen der Methoden
- Was kann man damit machen?
  - Algorithmen implementieren
  - Kleinere Programme schreiben
- Große Programme?
  - Ja, aber nicht sinnvoll
  - Problem der Komplexität
- Komplexität
  - Schwierigkeiten
    - Komplizierte Algorithmen
    - Komplexe Software, d.h. viele aber meist einfache Funktionen
    - Umfangreicher Sachverhalt, der abgebildet werden muss
  - Probleme bei der Softwareentwicklung
    - Viele Funktionen müssen organisiert werden
    - (Großes) Team muss organisiert werden
    - Anforderungen an die Software können sich ändern

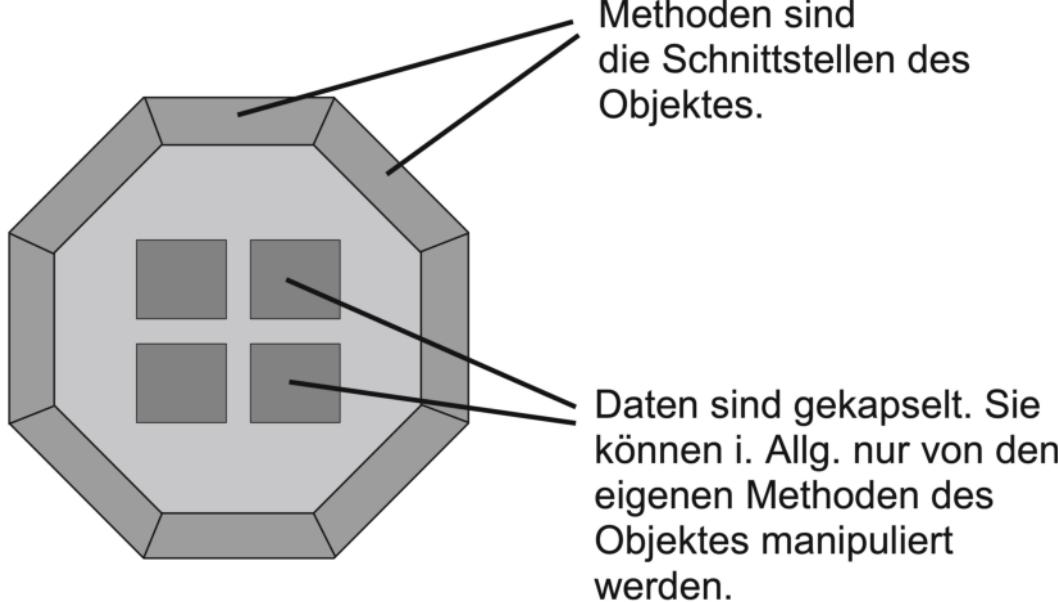
# Komplexität in den Griff bekommen

- Beachten von verschiedenen Prinzipien
  - Ein abgeschlossener Codeteil (Modul) sollte nur eine Verantwortung (Aufgabe) haben.
  - Wiederholungen vermeiden → DRY-Prinzip!
  - Trennung der Schnittstelle von der Implementierung!
  - Testbarkeit
  - etc.
- Techniken der Objektorientierung unterstützen diese Prinzipien.
- **ABER:**
  - Objektorientierung führt nicht automatisch zu besserer Software.
  - Objektorientierung unterstützt auch nicht alle Prinzipien vollständig!

# Objekt (1)

- Eigenschaften von Objekten:
  - Identität
    - Jedes Objekt hat eine unveränderliche Identität.
    - Nachrichten an Objekte werden über ihre Identität geschickt.
  - Zustand
    - Der Zustand eines Objekts setzt sich aus der aktuellen Belegung der Datenfelder zusammen.
  - Verhalten
    - Das Verhalten eines Objekts gibt an, wie sich ein Objekt beim Empfangen einer Nachricht verhält.
- Ein Objekt ist ein Verbund von Operationen die sich einen Zustand teilen.
  - Operationen bestimmen auf welche Nachrichten ein Objekt reagieren kann.
  - Direkter Zugriff auf den Zustand ist von außerhalb nicht möglich.
  - Der Zustand wird durch Variablen repräsentiert (= Objektvariablen).
  - Der Zustand kann nur von den Operationen des Objekts verändert werden.
  - Durch die Operationen wird das Verhalten des Objekts beschrieben.

# Objekt (2)



- Objekte gehen einen Kontrakt ein, der die Rahmenbedingungen beim Aufruf einer Operation regelt.
- Die Rahmenbedingungen sind:
  - Vorbedingungen
  - Nachbedingungen
  - Invarianten

# Objektorientierung

- Objektorientierung
  - Ist ein Programmierparadigma für die Analyse, den Entwurf und die Implementierung von objektorientierten Systemen.
  - Objektorientierte Programmierung erlaubt eine natürliche Modellierung vieler Problemstellungen.
  - Die objektorientierten Prinzipien können durch eine objektorientierte Implementierung in unterschiedlichem Maß eingehalten werden.
- Grundprinzipien der Objektorientierung
  - Kapselung
  - Polymorphie
  - Vererbung

# Objektorientierte Konzepte

- Klassenkonzept:
  - Deklaration von Klassen
  - Klassen beschreiben Eigenschaften von Objekten dieser Klasse
- Prototypkonzept:
  - Beschreibung einzelner Objekte
  - Neue Objekte werden durch Klonen und Abändern bestehender Objekte erzeugt

# Klassenkonzept

# Klassen

- Ein Objekt ist bei objektorientierten Programmiersprachen, welche auf dem Klassenkonzept basieren, immer zumindest einer konkreten Klasse zugeordnet.
- Eine Klasse:
  - Beschreibt Gemeinsamkeiten (Eigenschaften und Operationen) von Objekten
  - Definiert einen Datentyp
  - Dient als Konstruktionsplan für Objekte des Typs der Klasse
  - Ist ein elementares Modellierungswerkzeug
- Konzepte objektorientierter Programmiersprachen mit Klassenkonzept:
  - Abstraktion
  - Kapselung
  - Beziehungen
  - Polymorphie

# Abstraktion

- Der Prozess der Abstraktion hilft essenzielle Details herauszuheben, unwichtige Details zu ignorieren und die Komplexität der resultierenden Programme zu reduzieren.
- Trennung zwischen Konzept und Umsetzung
- Realisiert durch Klassen und Objekte
- Objekte sind tatsächlich existierende Dinge aus der Domäne des Programms
- Klassen sind Beschreibungen eines oder mehrerer ähnlicher Objekte. Sie beschreiben mindestens:
  - Wie ist ein Objekt der Klasse zu bedienen?
  - Welche Eigenschaften hat ein Objekt der Klasse?
  - Wie verhält sich ein Objekt der Klasse?
  - Wie wird ein Objekt der Klasse hergestellt?

# Kapselung

- Variablen und Methoden werden in einer logischen Einheit, dem Objekt, gekapselt.
- Daten und Methoden gehören zum Objekt
  - Daten gehören explizit einem Objekt
  - Methoden repräsentieren das Verhalten des Objekts
  - Methoden sollten die einzige Möglichkeit sein um mit dem Objekt interagieren zu können
  - Direkter Zugriff auf Daten ist nicht erlaubt.
- Kapselung hilft dabei
  - den Aufwand bei Änderungen der zugrundeliegenden Datenstruktur eines Objekts gering zu halten.
  - die Daten konsistent zu halten, da ein direkter Zugriff auf die Daten nicht erlaubt ist.

# Beziehungen

- Vererbung („is-a“-Beziehung)
  - Beziehung zwischen ähnlichen Klassen
    - Abbildung von Beziehungen zwischen Klassen durch Hierarchien aus Klassen und Unterklassen.
    - Objekte der Hierarchie teilen die Spezifikation bzw. Spezifikation und Implementierung gewisser Operationen.
- Aggregation und Komposition („part-of“-Beziehung)
  - Zusammensetzung eines Objekts aus anderen Objekten.
  - Komposition bezeichnet die strenge Form der Aggregation auf Grund einer existenziellen Abhängigkeit.
- Verwendungs- und Aufrufbeziehungen
  - Verwendung anderer Klassen bzw. Objekte als
    - Temporäre Variable
    - Typ eines formalen Parameters

# Polymorphie

- Polymorphie = Vielgestaltigkeit
- Die Vielgestaltigkeit bezieht sich bei Programmiersprachen auf Variablen und Methoden.
  - Eine Variable kann Objekte unterschiedlicher Klassen aufnehmen.
  - Eine Methode kann mit aktuellen Parametern, die unterschiedliche Typen aufweisen, aufgerufen werden.
- Verschiedene Formen der Polymorphie
  - Universelle Polymorphie
    - Generizität
    - Untertypen
  - Ad-hoc Polymorphie
    - Überladen
    - Typumwandlung



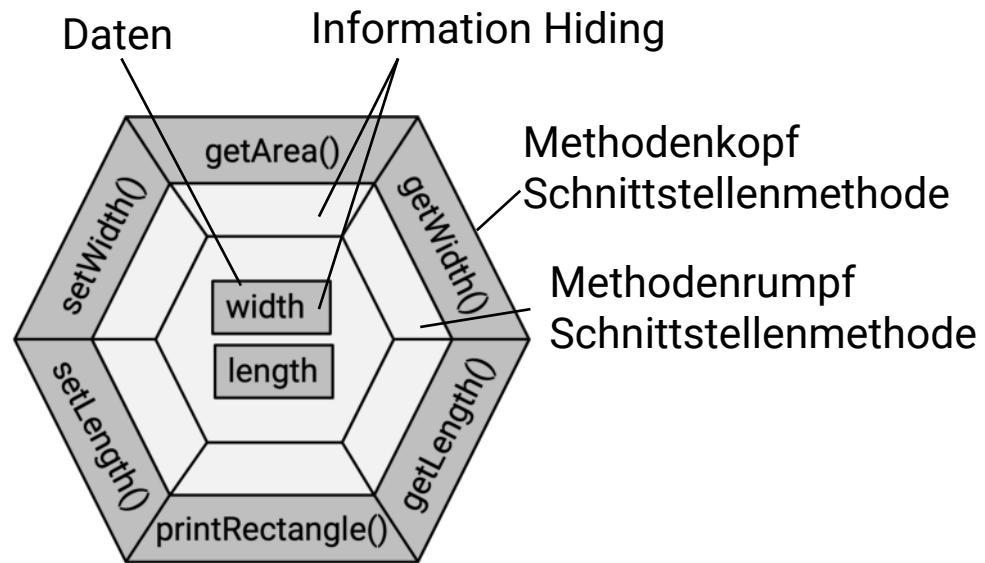
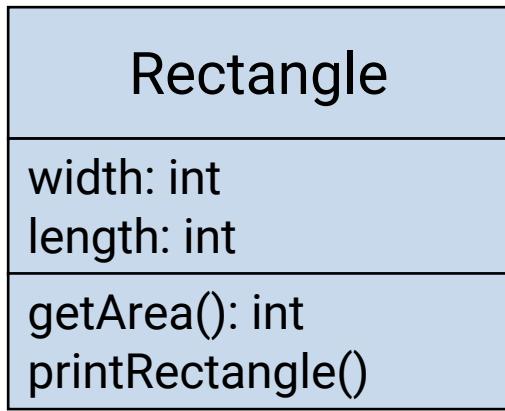
# Klassen & Objekte

# Klassen und Objekte (1)

- Objekte sind Exemplare der Klassen zu denen sie gehören.
- Es können mehrere Exemplare derselben Klasse existieren.
- Objekte können Exemplare mehrerer Klassen sein.
- In den meisten Programmiersprachen sind Objekte nur Exemplare einer Klasse.
- Jedes Exemplar hat einen eigenen Zustand.
- Exemplare teilen die Implementierung der Operationen.
- Das Verhalten von Objekten derselben Klasse kann sich aufgrund des internen Zustands unterscheiden.
- Interaktion mit einem Objekt kann nur über die bereitgestellten Operationen erfolgen (=Kapselung, engl. encapsulation)

# Klassen und Objekte (2)

- Beispiel: Rectangle-Objekt

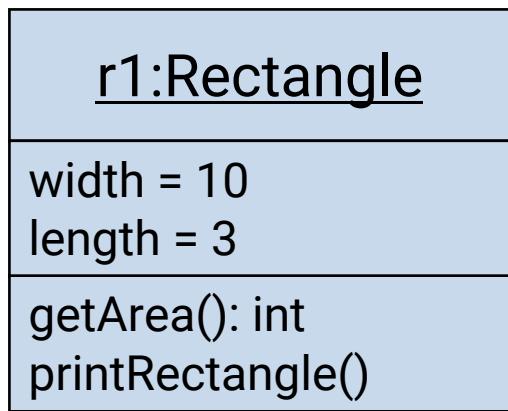


- Geheimnisprinzip (information hiding)

- Die Daten einer Kapsel, die Rümpfe der Schnittstellenmethoden und die Hilfsmethoden sollen nach außen nicht direkt sichtbar sein.

Abbildung angelehnt an „Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik“, Goll

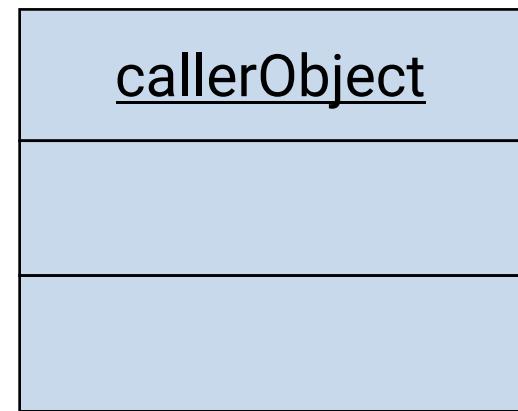
# Kollaboration von Objekten



← getLength()

← getArea()

Implementierung der  
Berechnung verborgen



Durch die Kapselung verschmelzen die Daten  
und Methoden eines Objektes (externe Sicht).



# Typsysteme

# Typsysteme

- Klassen legen den Typ für ihre Exemplare fest.
  - Das Typsystem kann dadurch Eigenschaften von Objekten und auf sie anwendbare Operationen erkennen.
- Das Typsystem ist Bestandteil der Umsetzung einer Programmiersprache oder ihres Laufzeitsystems.
- Das Typsystem ordnet jedem Ausdruck einen Typ zu.
- Überprüfung von Ausdrücken auf Verträglichkeit mit dem Typsystem. Beispielsweise:
  - Ist die vorliegende Operation auf dem Objekt erlaubt?
  - Kann das Objekt der vorliegenden Variable zugewiesen werden?
- Unterscheidung bei Typsystemen
  - Statisches vs. dynamisches Typsystem
  - Starkes vs. schwaches Typsystem
  - Namensbasiertes vs. strukturbasiertes Typsystem
  - ...

# Statisches Typsystem

- Eigenschaften
  - Typ von Variablen und Parametern wird im Quelltext deklariert.
  - Schränkt ein, welche Objekte einer Variable zugewiesen werden können.
  - Compiler erkennt schon eine unpassende Zuweisung oder unpassenden Aufruf einer Operation.
- Vorteile
  - Programmstruktur ist übersichtlicher.
  - Entwicklungsumgebungen können mehr Unterstützung anbieten.
  - Fehler können früher erkannt werden.
  - Kompilierte Anwendungen können besser optimiert werden.
- Beispiele: Java, C++, C#

# Dynamisches Typsystem

- Eigenschaften
  - Variablen sind keinem deklarierten Typ zugeordnet.
  - Variablen können beliebige Objekte referenzieren.
  - Ob eine Operation auf ein Objekt erlaubt ist, wird zur Laufzeit überprüft.
- Vorteile
  - Flexibilität
  - Keine Notwendigkeit einer expliziten Typumwandlung!
- Klassen und Typen sind entkoppelt.
- Beispiele: Smalltalk, Ruby, Python, PHP, JavaScript

# Stark und schwach typisierte Programmiersprachen

- Stark typisiert
  - Überwacht Zugriff auf Objekte
  - Variable kann nur auf Objekt verweisen, welches die Spezifikation ihres Typs erfüllt.
- Schwach typisiert
  - Keine Überwachung
  - Variable kann auch auf Objekt verweisen, welches die Spezifikation ihres Typs nicht erfüllt.

# Quellen

- Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Guido Krüger, Heiko Hansen: **Handbuch der Java-Programmierung**, Addison Wesley, 7. Auflage, 2011
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022
- Joachim Goll: **Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik**, Springer Vieweg, 2018

# Klassen & Objekte in Java

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Klassen

- Klassendefinitionen beginnen mit dem Schlüsselwort `class` und dem Namen der Klasse.
- Anschließend folgen in geschweiften Klammern eine beliebige Anzahl von:
  - Feldern (Objekt- und Klassenvariablen)
  - Methoden
  - Konstruktoren
  - Klassen- sowie Exemplarinitialisierer
  - Geschachtelte Klassen, Schnittstellen und Aufzählungen

```
public class Rectangle { ← Klassendeklaration  
...  
}
```



# Felder

- Felder werden auch als Attribute oder Membervariablen bezeichnet.
- Statische Felder werden auch Klassenvariablen genannt.
- Objektbezogene Felder werden auch Objektvariablen genannt.
- Objekt- und Klassenvariablen können bei der Deklaration initialisiert werden.

```
public class Rectangle {  
    private int width;  
    private int length; }  
    private static int instanceCounter;  
    ...  
}
```

Objektvariablen

Klassenvariable



# Methoden

```
public class Rectangle {  
    ...  
    public int getWidth() {  
        return width;  
    }  
  
    public void setWidth(int width) {  
        this.width = width;  
    }  
  
    public int getLength() {  
        return length;  
    }  
  
    public void setLength(int length) {  
        this.length = length;  
    }  
  
    public int getArea() {  
        return width * length;  
    }  
  
    public void printRectangle() {  
        System.out.println("Rectangle width: " + width + ", length: " + length);  
    }  
}
```



# this-Referenz

- Jedes Objekt hat eine this-Referenz.
- this ist in jeder nicht-statischen Methode automatisch definiert.
- Mit this referenziert ein Objekt auf sich selbst.
- Mithilfe von this können Objektvariablen von lokalen Variablen unterschieden werden.
  - Objektvariablen und lokale Variablen können den gleichen Bezeichner haben.
- this kann als Rückgabewert oder Parameter verwendet werden.

```
public class Rectangle {  
  
    ...  
    private int width;  
    ...  
  
    public void setWidth(int width) {  
        this.width = width;  
    }  
    ...  
}
```



# Von der Klasse zum Objekt

- Typ Rechteck ist durch die Klasse Rectangle definiert.
  - Nächster Schritt: Erzeugung eines Rectangle-Exemplars

```
Rectangle r1 = new Rectangle(10, 3);
```

Datentyp Bezeichner neues Rectangle-Objekt

```
...  
public Rectangle(int width, int length) {  
    this.width = width;  
    this.length = length;  
}  
  
public Rectangle() {  
}  
...  
  
Konstruktoren
```



# Konstruktoren

- Konstruktoren werden bei der Erzeugung von Exemplaren einer Klasse verwendet.
  - Objektvariablen können damit mit sinnvollen Werten belegt oder initialisiert werden.
- In Kombination mit dem Schlüsselwort new wird durch einen Konstruktor ein Exemplar erzeugt und eine Referenz darauf zurückgegeben.
- Konstruktor
  - Hat den gleichen Namen wie die Klasse.
  - Wird wie eine Methode ohne die Angabe eines Rückgabetyps deklariert.
- Eine Klasse kann überladene Konstruktoren haben.

# Konstruktoren und Parameter

- Konstruktoren mit Parametern

- Übergabe von Werten (für die Initialisierung)

```
public Rectangle(int width, int length) {...}
```

```
public Rectangle(int sideLength) {...}
```

- Parameterlose Konstruktoren

```
public Rectangle() {...}
```

- Können beliebigen Code enthalten.

# Default-Konstruktor

- Java erzeugt einen Default-Konstruktor automatisch, falls in einer Klasse keine Konstruktoren deklariert sind.
- Sobald explizit ein Konstruktor implementiert wurde, wird der Default-Konstruktor nicht erzeugt.
- Soll eine Klasse zusätzlich einen parameterlosen-Konstruktor haben, muss dieser explizit ausprogrammiert werden.
- Beispiel: Beide Implementierungen bieten dieselbe Funktionalität.

```
public class Point {  
    int x;  
    int y;  
}
```

```
public class Point {  
    int x;  
    int y;  
  
    public Point() {} ← Parameterloser Konstruktor  
}
```

# Konstruktorenverkettung mit this()

- Ein Konstruktor kann mit `this()` (bzw. `this(par1, ...)`) einen anderen Konstruktor derselben Klasse aufrufen.
- Folgende Einschränkungen existieren:
  - Der `this`-Aufruf darf nur einmal vorkommen.
  - Der `this`-Aufruf muss als erste Anweisung auftreten.

```
public class Rectangle {  
  
    ...  
  
    public Rectangle(int width, int length) {  
        this.width = width;  
        this.length = length;  
    }  
  
    public Rectangle(int sideLength) {  
        this(sideLength, sideLength);  
    }  
  
    ...  
}
```



# Exemplarinitialisierer

- Die Exemplarinitialisierer einer Klasse werden ausgeführt, wenn ein Exemplar erzeugt wird.
- Eine Klasse kann mehrere Exemplarinitialisierer haben.
- Exemplarinitialisierer eignen sich um Code, welcher am Beginn jedes Konstruktors stehen müsste zu bündeln und so Code-Duplikate zu vermeiden.
- Exemplarinitialisierer werden leicht übersehen, da sie nicht explizit in den Konstruktoren aufgerufen werden und auch nicht explizit in der API-Dokumentation angeführt werden.
- Beide Nachteile können durch den Einsatz einer Initialisierungsmethode, welche in allen Konstruktoren aufgerufen wird, umgangen werden.

# Initialisieren des Objektzustandes

- Feldinitialisierung

```
public class Point {  
    int x = 1;  
    int y = 1;  
}
```

- Initialisierung durch einen Exemplarinitialisierer

```
public class Point {  
    int x;  
    int y;  
    {  
        x = 1;  
        y = 1;  
    }  
}
```

- Initialisierung im Konstruktor
- Automatische Initialisierung

# Automatische Initialisierung

- Klassenvariablen und Objektvariablen, welche nicht `final` sind, werden **automatisch** initialisiert:

Datentyp	Initialisierung
<code>boolean</code>	<code>false</code>
<code>byte</code>	<code>(byte) 0</code>
<code>short</code>	<code>(short) 0</code>
<code>int</code>	<code>0</code>
<code>long</code>	<code>0L</code>
<code>float</code>	<code>0.0f</code>
<code>double</code>	<code>0.0d</code>
<code>char</code>	<code>'\u0000'</code>
Referenztypen (z.B. String)	<code>null</code>

- Lokale Variablen werden nicht automatisch initialisiert.

# Verwendung von Objekten (1)

- Nach der Erzeugung eines Objekts kann dieses verwendet werden.
- Auf die Objektvariablen und die Methoden kann durch Qualifizierung zugegriffen werden:  
`objekt.Objektvariablenname`  
`objekt.Methodename(...)`
- Es wird zwischen einfachen und qualifizierten Bezeichnern unterschieden.
  - Einfache Bezeichner bestehen nur aus einem Namen.
  - Qualifizierte Bezeichner bestehen aus einer Folge von Namen, die jeweils durch einen Punkt getrennt sind.

# Verwendung von Objekten (2)

```
public class RectangleApplication {  
  
    public static void main(String[] args) {  
  
        Rectangle rectangle1 = new Rectangle(20, 3);  
        Rectangle rectangle2 = new Rectangle(10, 5);  
  
        rectangle1.printRectangle();  
        rectangle2.printRectangle();  
  
        System.out.println("Area rectangle 1: " + rectangle1.getArea());  
        System.out.println("Area rectangle 2: " + rectangle2.getArea());  
  
    }  
}
```

Ausgabe:

```
Rectangle width: 20, length: 3  
Rectangle width: 10, length: 5  
Area rectangle 1: 60  
Area rectangle 2: 50
```



# Zugriffsmodifikatoren in Java

- **public**

- Zugriff aus **beliebigen Klassen** erlaubt.

- **private**

- Zugriff nur aus **derselben Klasse** erlaubt.

- **protected**

- Zugriff aus **beliebigen Klassen desselben Pakets** und aus **Unterklassen** erlaubt.

- Kein Attribut (Default)

- Zugriff aus **beliebigen Klassen desselben Pakets** erlaubt.

# private (Unterscheidung)

## Klassenbasierte Sichtbarkeit

- Auf private Daten und Methoden eines Objekts kann nur aus Methoden der Klasse zugegriffen werden, in der diese privaten Elemente deklariert wurden.
- Auf private Elemente anderer Exemplare derselben Klasse kann zugegriffen werden (Klasse bestimmt Sichtbarkeit).
- Wird in Java verwendet.

## Objektbasierte Sichtbarkeit

- Auf private Daten und Methoden eines Objekts kann nur innerhalb von Methoden zugegriffen werden, die auf dem Objekt selbst ausgeführt werden.
- Es kann nicht auf private Elemente anderer Exemplare der Klasse zugegriffen werden.

# Getter- und Setter-Methoden (1)

- Direkter Zugriff auf Objektvariablen sollte nur in seltenen Fällen ermöglicht werden → Kapselung!
- Eine private Objektvariable ist von außen nicht erreichbar.
- Sollen Werte gelesen bzw. geändert werden können:
  - Getter- bzw. Setter-Methode (Accessors, Mutators)
  - Beispiel

```
private type identifier;  
public type getIdentifier() {...}  
public void setIdentifier(type identifier) {...}
```
- Vorteile durch Methoden
  - Kontrolle der übergebenen Werte
  - Hilfsmittel zur Fehlersuche
  - Setter/Getter für scheinbare Datenelemente
  - Implementierungsdetails verbergen

# Getter- und Setter-Methoden (2)

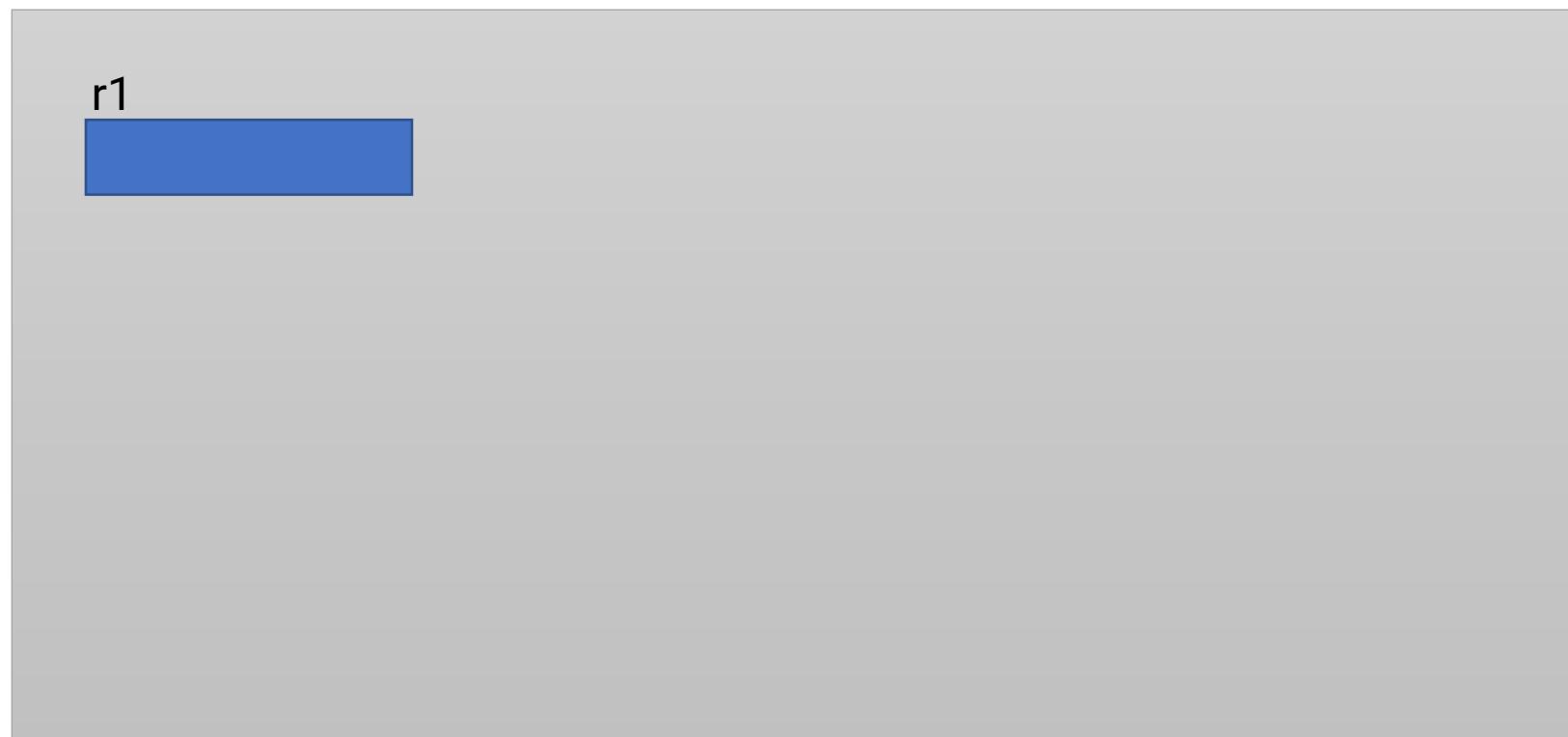
- Es muss nicht für jedes Feld einer Klasse ein Getter und Setter angeboten werden.
- Ein intensiver Gebrauch von Getter- und Setter-Methoden ist kein Zeichen von guter Objektorientierung.
  - Deutet eher auf einen fragwürdigen OO-Entwurf hin.
    - Objekt verkommt zu einem Datencontainer – das ist nicht Objektorientierung!
    - Das Verhalten des Objekts kann zu sehr von anderen Klassen gesteuert werden.

```
public class Rectangle {  
    private int width;  
    ...  
    public int getWidth() {  
        return width;  
    }  
  
    public void setWidth(int width) {  
        this.width = width;  
    }  
    ...  
}
```



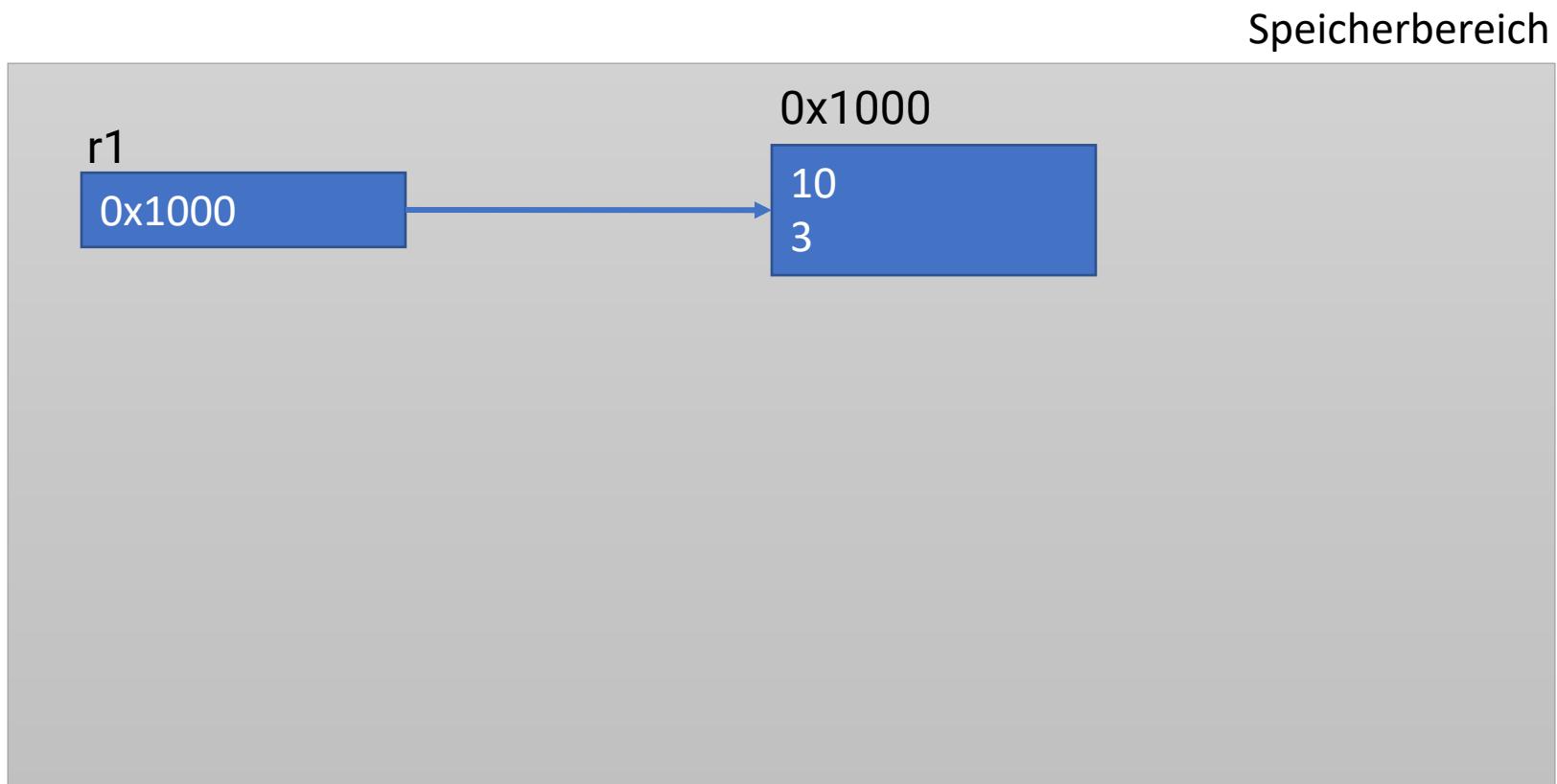
# Objekte und Referenzen (1)

Rectangle r1;



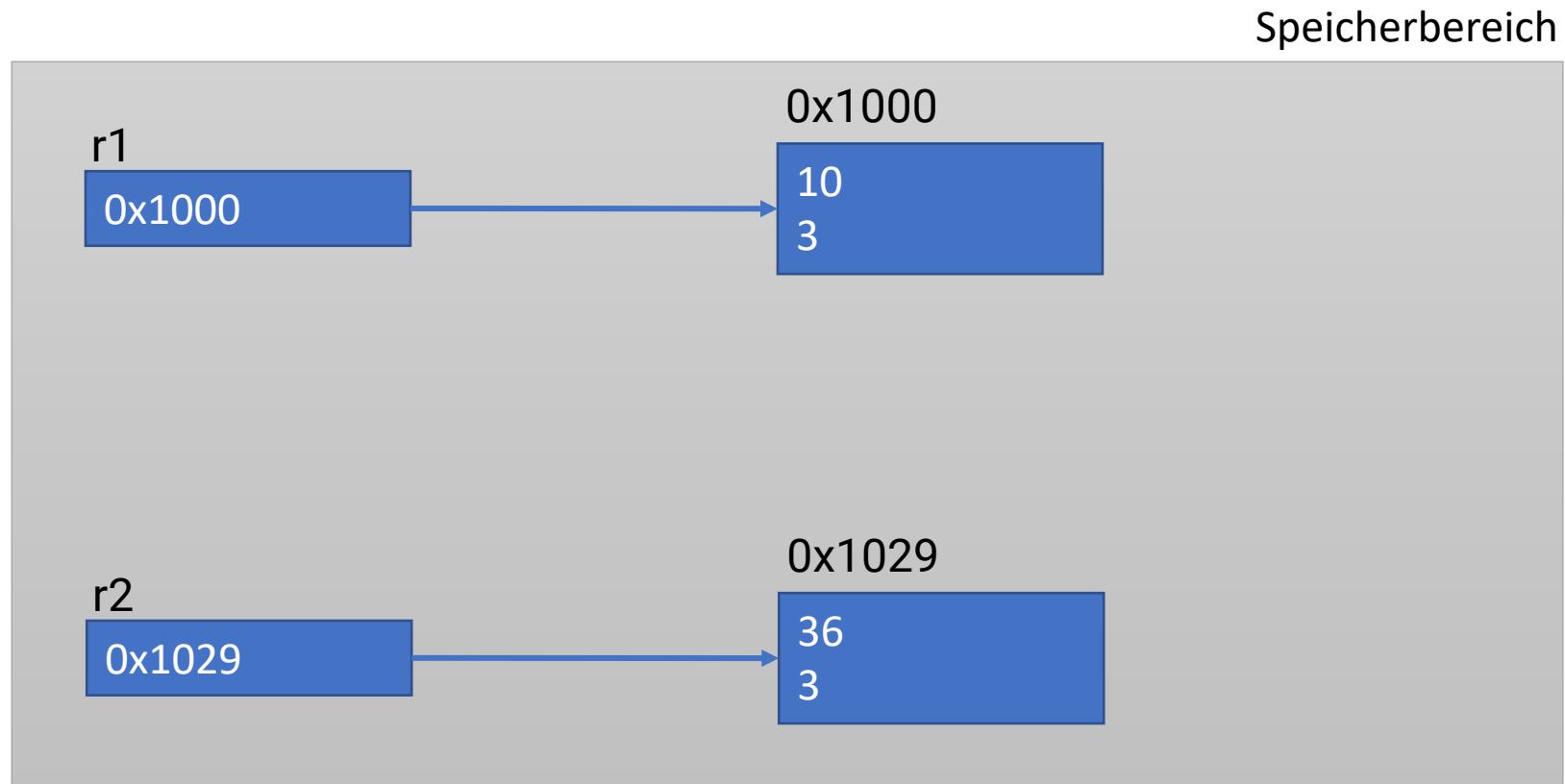
# Objekte und Referenzen (2)

```
Rectangle r1;  
r1 = new Rectangle(10, 3);
```



# Objekte und Referenzen (3)

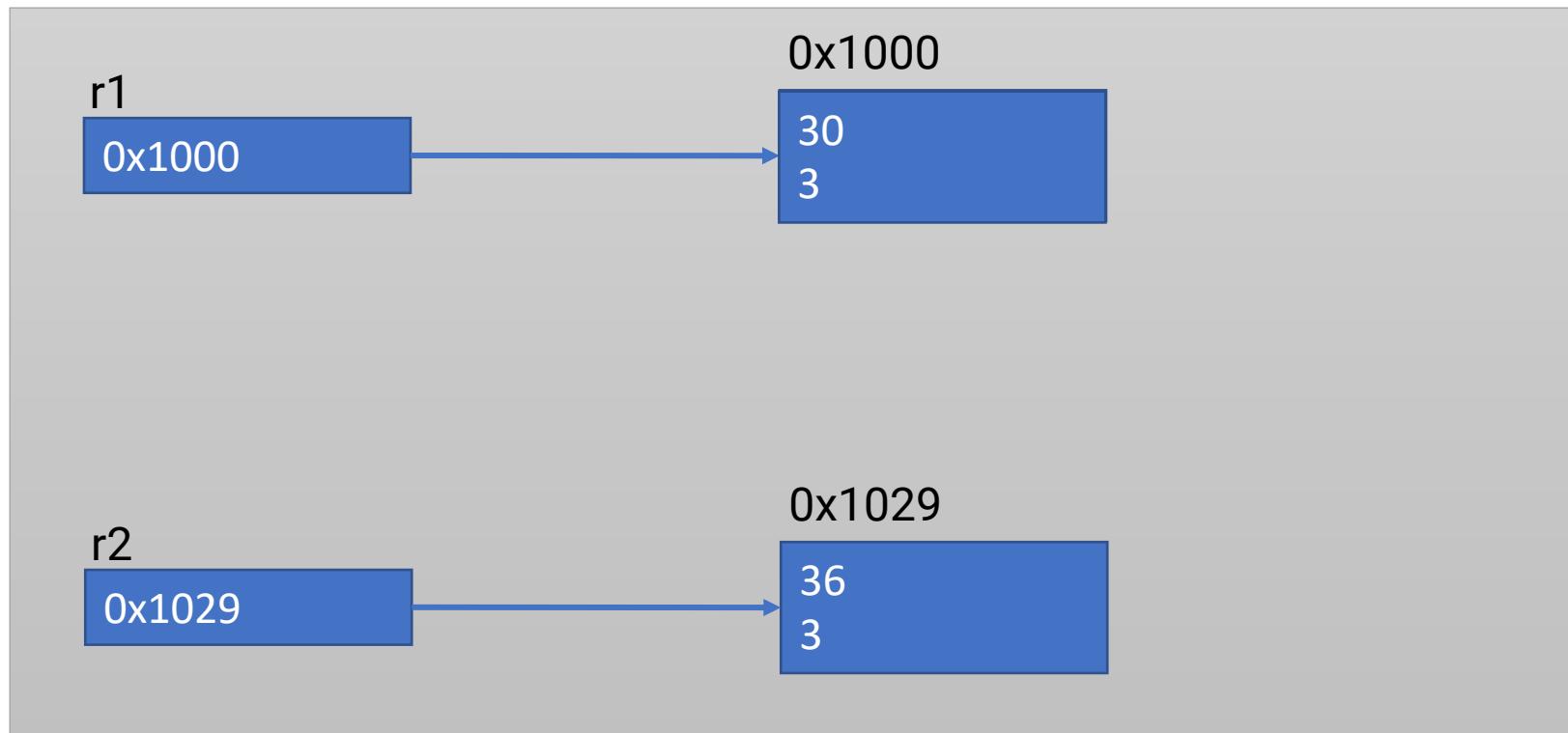
```
Rectangle r1;  
r1 = new Rectangle(10, 3);  
Rectangle r2 = new Rectangle(36, 3);
```



# Objekte und Referenzen (4)

```
Rectangle r1;  
r1 = new Rectangle(10, 3);  
Rectangle r2 = new Rectangle(36, 3);  
r1.setWidth(30);
```

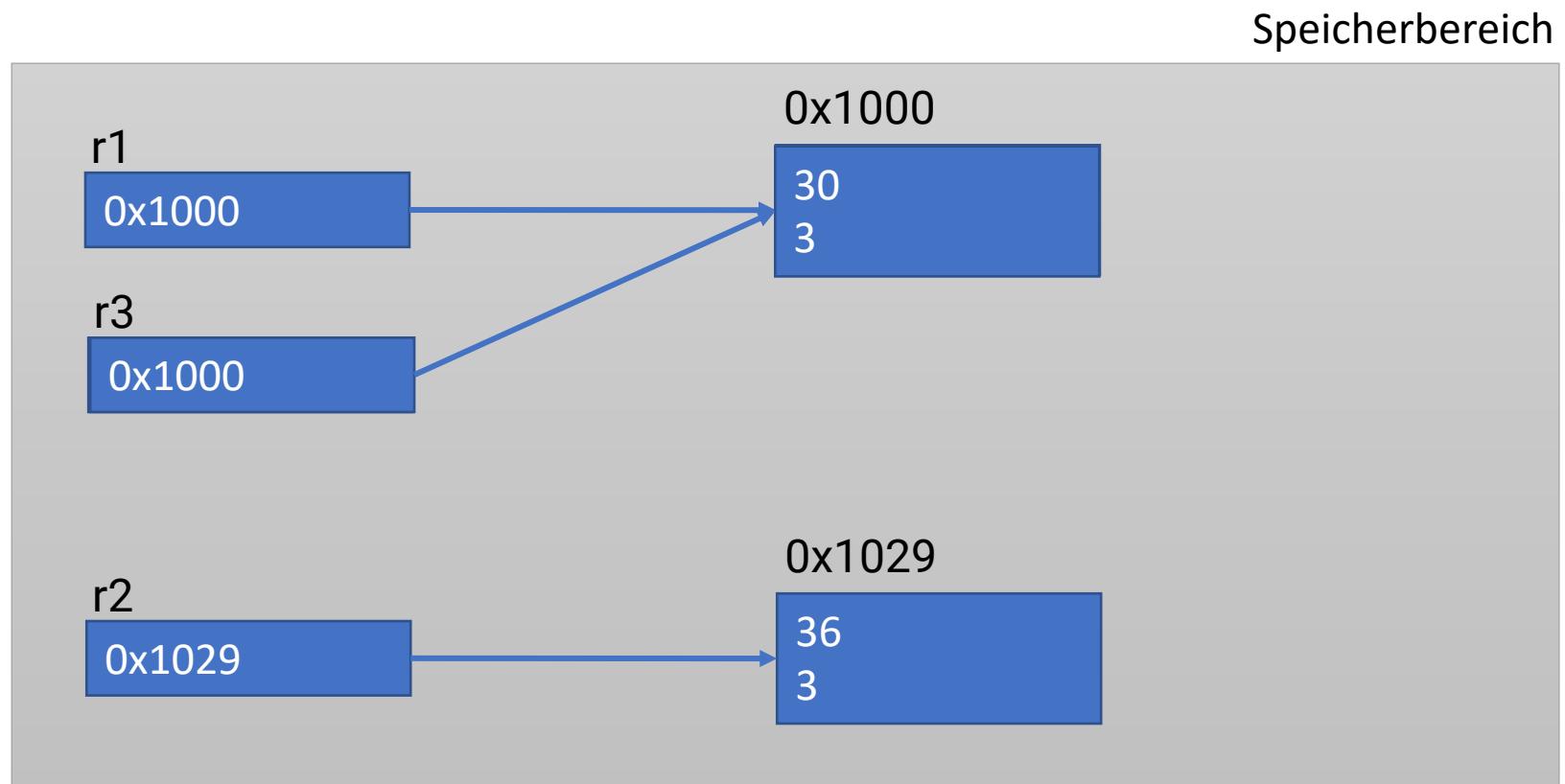
Speicherbereich



# Objekte und Referenzen (5)

...

```
Rectangle r3 = r1;
```

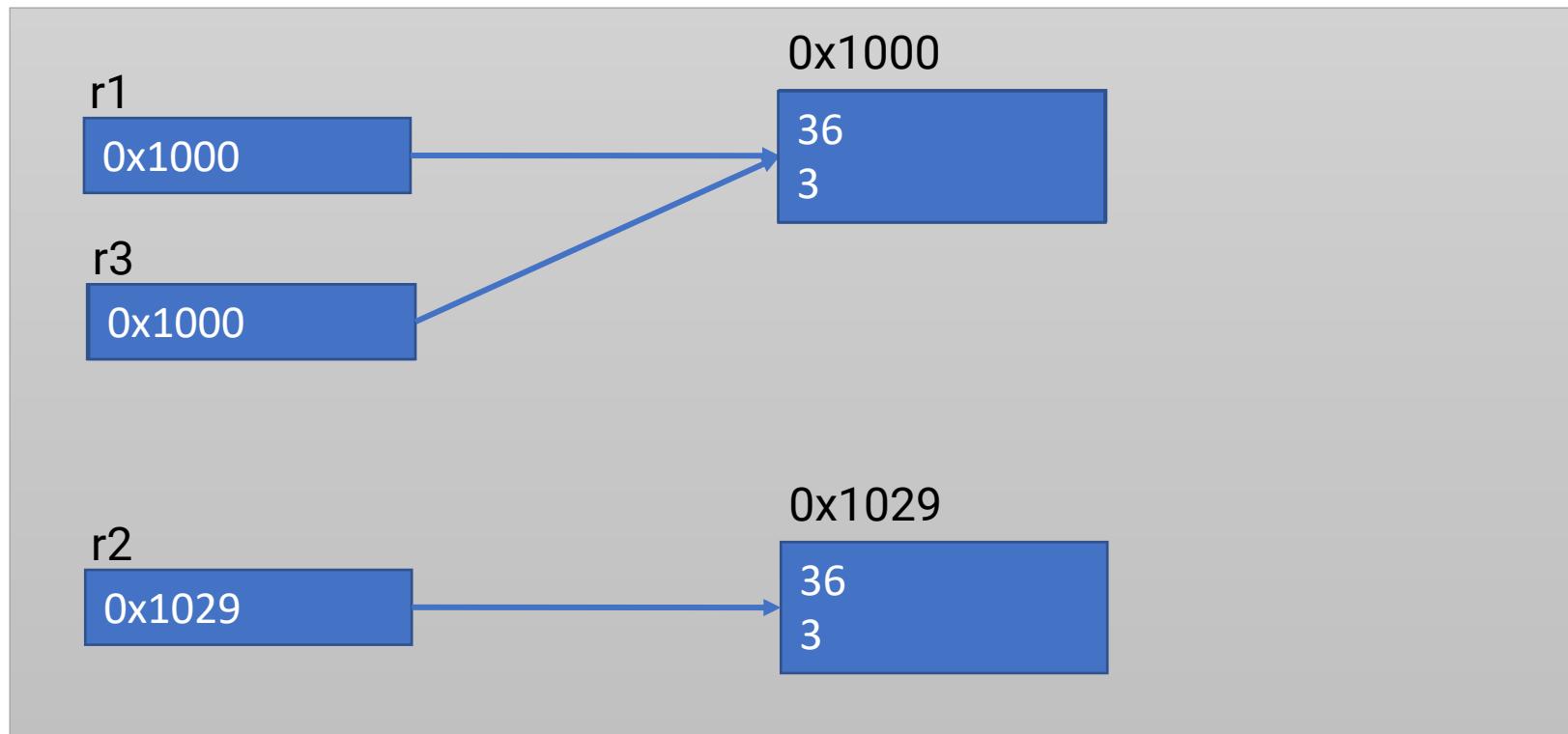


# Objekte und Referenzen (6)

...

```
Rectangle r3 = r1;  
r1.setWidth(36);
```

Speicherbereich



# Vergleiche

- Objekte können mit `==` und `!=` verglichen werden.
  - **Achtung:** Es werden nur die Referenzen und nicht die Werte verglichen.
- Eine Überprüfung, ob Objekte denselben Wert haben, geschieht durch:
  - Eine eigene Vergleichsmethode
  - Den Vergleich der einzelnen Objektvariablen

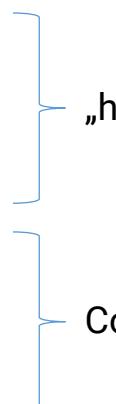
```
...
System.out.println(r1 == r2); // false
System.out.println(r1 == r3); // true
System.out.println(r2 == r3); // false
...
```

# Kopieren von Objekten

- Bei der Zuweisung von Referenzvariablen wird nur die Referenz kopiert. Es wird keine Kopie des Objekts erzeugt.
- Ein Objekt kann beispielsweise durch einen Kopier-Konstruktor (Copy-Constructor) oder die `clone()`-Methode kopiert werden.
- Beim Kopieren von Objekten wird zwischen flacher und tiefer Kopie unterschieden.
- Flache Kopie (shallow copy)
  - Es wird nur das Objekt selbst und die darin enthaltenen Werte kopiert.
  - Bei Referenzvariablen wird im Original und der Kopie auf dasselbe Objekt verwiesen.
- Tiefe Kopie (deep copy)
  - Enthält ein Objekt weitere Objekte, so wird das Kopieren rekursiv fortgesetzt.
  - Original und Kopie enthalten logisch gleiche aber getrennte Datenelemente.

# Beispiel Kopier-Konstruktor

```
public class Rectangle {  
  
    private int width;  
    private int length;  
  
    ...  
    public Rectangle(int width, int length) {  
        this.width = width;  
        this.length = length;  
    }  
  
    public Rectangle(Rectangle toCopy) {  
        this.width = toCopy.width;  
        this.length = toCopy.length;  
    }  
    ...  
}
```



„herkömmlicher“ Konstruktor

Copy-Constructor

```
...  
// usage  
Rectangle r4 = new Rectangle(20, 5);  
Rectangle r5 = new Rectangle(r4);  
...
```





# Statisch vs. Objektbezogen

# Statische Methoden und Felder

- Bisher in diesem Foliensatz: alle Methoden sind an Objekte gebunden.
- Methoden und auch Felder sind nicht immer direkt von Objekten abhängig
  - `Math.max()` – ermittelt das Maximum zweier Zahlen
  - `Math.PI` – Annäherung der Kreiszahl Pi ( $\pi$ )
  - `Integer.parseInt()` – wandelt String in Integer um
- Derartige Methoden sollten nicht dem Objekt, sondern der Klasse zugeordnet sein (unabhängig vom Objekt-Zustand).
- Zugriff auf statische Felder und Methoden
  - `Classname.field` bzw. `Classname.method(...)`
  - Innerhalb der Klasse kann die Qualifizierung (`Classname.`) weggelassen werden

# Statische Methoden

- Statische Methoden werden mit dem Schlüsselwort `static` deklariert.
- Sie werden auch als Klassenmethoden bezeichnet.
- Eine statische Methode wird immer ohne Bezug auf ein bestimmtes Objekt bearbeitet.
- Bei der Deklaration von Klassenmethoden wird ein statischer Kontext eingeführt.
- In einem statischen Kontext kann weder explizit noch implizit auf das aktuelle Exemplar der Klasse verwiesen werden.
- Es gibt beispielsweise folgende Einschränkungen:
  - Die Verwendung von `this` und `super` ist nicht möglich.
  - Unqualifizierte Referenzen auf Objektvariablen und objektbezogene Methoden sind nicht möglich.

# Statische Felder

- Statische Felder werden mit dem Schlüsselwort `static` deklariert.
- Sie werden auch als Klassevariablen oder statische Variablen bezeichnet.
- Ein statisches Feld existiert für eine Klasse immer genau einmal unabhängig davon wie viele Exemplare der Klasse existieren.
- Bei der Deklaration eines statischen Feldes wird ein statischer Kontext eingeführt.

# Rectangle-Beispiel mit statischem Zähler (1)

```
public class Rectangle {  
    private int width;  
    private int length;  
    private static int instanceCounter; // bound to class  
  
    Rectangle(int width, int length) {  
        this.width = width;  
        this.length = length;  
        ++instanceCounter; // bound to class  
    }  
    ...  
    public static int getInstanceCounter() { // bound to class  
        return instanceCounter;  
    }  
    ...  
}
```



# Rectangle-Beispiel mit statischem Zähler (2)

```
public class RectangleApplication {  
  
    public static void main(String[] args) {  
        Rectangle rectangle1 = new Rectangle(20, 3);  
        System.out.println("Instances: " + Rectangle.getInstanceCounter());  
        Rectangle rectangle2 = new Rectangle(10, 5);  
        System.out.println("Instances: " + Rectangle.getInstanceCounter());  
  
        rectangle1.printRectangle();  
        rectangle2.printRectangle();  
  
        System.out.println("Area rectangle 1: " + rectangle1.getArea());  
        System.out.println("Area rectangle 2: " + rectangle2.getArea());  
    }  
}
```

Ausgabe:

```
Instances: 1  
Instances: 2  
Rectangle width: 20, length: 3  
Rectangle width: 10, length: 5  
Area rectangle 1: 60  
Area rectangle 2: 50
```



# Statischer Initialisierer

- Für die Initialisierung statischer Variablen kann auch ein statischer Initialisierer verwendet werden.
  - Hat keinen Namen und keine Parameter

```
static {  
    instanceCounter = 0;  
}
```

- Wird aufgerufen, wenn die Klasse geladen wird.
- Auch mehrere statische Blöcke sind möglich.

# Objektbezogen oder statisch

	Objektbezogen	Statisch
<b>Deklarationen</b>	ohne static	mit static
<b>Existieren</b>	in jedem Objekt	nur einmal pro Klasse
<b>Variablen werden angelegt</b>	wenn das Objekt erzeugt wird	wenn die Klasse geladen wird
<b>Variablen werden freigegeben</b>	vom Garbage-Collector, wenn keine Referenz mehr auf das Objekt existiert	wenn die Klasse entladen wird
<b>Konstruktor/Initialisierer wird aufgerufen</b>	wenn das Objekt erzeugt wird	wenn die Klasse geladen wird
<b>Qualifizierung</b>	über das Objekt	über den Klassennamen (oder das Objekt)



# Beziehungen zwischen Klassen

# Beziehungen zwischen Klassen

- In einem objektorientierten Programm stehen Klassen (und damit Objekte) in Beziehung. Es gibt selten isolierte Objekte.

- Beispiel Verwaltung von Studierenden und Kursen

- Klasse Address



<src/at/ac/uibk/pm/objectorientation/coursemanagement/Address.java>

- Klasse ContactInformation

- Beziehung: Kontaktdaten enthalten eine Adresse



<src/at/ac/uibk/pm/objectorientation/coursemanagement/ContactInformation.java>

- Klasse Person

- Beziehung: Jede Person hat Kontaktdaten.



<src/at/ac/uibk/pm/objectorientation/coursemanagement/Person.java>

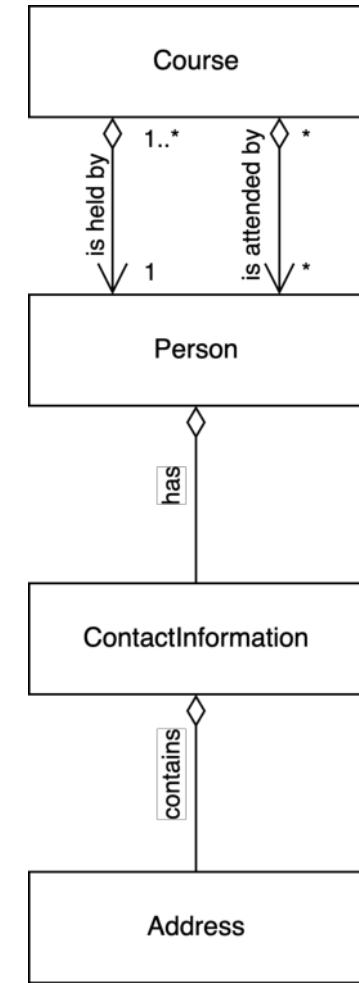
- Klasse Course

- Beziehung: zu einem Kurs können sich mehrere Personen anmelden.

- Beziehung: eine Person leitet den Kurs.



<src/at/ac/uibk/pm/objectorientation/coursemanagement/Course.java>



# Kohäsion und Kopplung

- Kohäsion
  - Kohäsion beschreibt wie gut eine Methode tatsächlich genau eine Aufgabe erfüllt oder wie genau abgegrenzt die Funktionalität einer Klasse ist.
  - Eine **hohe Kohäsion** deutet auf eine gute Trennung der Zuständigkeiten hin.
    - Das bedeutet eine Methode oder Klasse soll nur eine bestimmte Aufgabe erfüllen.
    - Das ist eine wünschenswerte Eigenschaft von objektorientiertem Code.
  - Hohe Kohäsion hilft bei der Wiederverwendung.
- Kopplung
  - Darunter versteht man, wie stark Klassen miteinander in Verbindung stehen.
  - Ziel einer guten Modellierung ist eine möglichst **lose Kopplung!**
    - Geringe Abhängigkeiten von Klassen untereinander.
    - Eine gute Datenkapselung und sinnvolle Zugriffsmethoden ermöglichen dies!

# Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Guido Krüger, Heiko Hansen: **Handbuch der Java-Programmierung**, Addison Wesley, 7. Auflage, 2011
- Christian Silberbauer: **Einstieg in Java und OOP**, Springer Vieweg, 2. Auflage, 2020

# Aufzählungstypen

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Motivation

- Variable sollte nur einen Wert aus einer Menge vordefinierter Werte annehmen können.
  - Beispiel: Farbvariable, welche die Werte Rot, Grün und Blau besitzen kann.
- Simpler Ansatz

```
...  
private static final int RED = 0, GREEN = 1, BLUE = 2;  
...
```

- Compiler kann dabei aber nicht garantieren, dass einer Farbvariable nur Farbwerte zugewiesen werden.

```
...  
int color = RED;  
...  
color = 100; Falsch!  
Keiner der vordefinierten Werte!  
...
```

- Typsicherheit ist nicht gegeben!

# Aufzählungstypen

- Aufzählungstypen werden auch Enumerationstypen oder kurz Enums genannt.
- Ein Aufzählungstyp definiert die Menge seiner Werte (Aufzählungskonstanten) durch namentliche Aufzählung.
- Jede Aufzählungskonstante definiert ein Exemplar des Enumerationstyps.
  - Es existieren neben den Aufzählungskonstanten keine weiteren Exemplare eines Enumerationstyps.
  - Der Versuch weitere Exemplare zu erzeugen führt zu einem Kompilierfehler.
- Beispiel

```
enum Color {RED, GREEN, BLUE}
enum Direction {NORTH, SOUTH, WEST, EAST}
Color c = Color.RED;
Direction d = Direction.NORTH;
// c = 100; // compilation error (incompatible types)
// c = d;   // compilation error (incompatible types)
...
```

# Aufzählungstypen als Klassen

- Aufzählungstypen sind spezielle Klassen.
- Aufzählungstypen können beispielsweise auch Objektvariablen, Methoden und Konstruktoren enthalten.
- Es können keine neuen Exemplare durch new erzeugt werden.
- Konstruktoren sind bei Aufzählungstypen immer private.
  - Der Versuch einen Konstruktor als public oder protected zu deklarieren führt zu einem Kompilierfehler.
- Das Laufzeitsystem stellt sicher, dass Aufzählungskonstanten nicht kopiert werden (es existiert nur ein Exemplar von jedem Wert).
  - Kopieren der Referenz auf eine Aufzählungskonstante ist möglich.

# Aufzählungskonstanten

- Können mit == und != verglichen werden.
- Sind in der Reihenfolge ihrer Deklaration geordnet.
  - Compiler ordnet jeder Aufzählungskonstante eine Ordinalzahl zu.
  - Aufzählungskonstanten sind aber keine Zahlen.
  - Im Farbbeispiel
    - Color.RED hat den Wert 0
    - Color.GREEN den Wert 1
    - Color.BLUE den Wert 2
- Können als case-Labels in switch-Anweisungen und switch-Ausdrücken verwendet werden.

```
...
switch(c) {
    case RED: ...
    case GREEN: ...
    case BLUE: ...
}
```

# Auszug bereitgestellter Methoden

- Aufzählungskonstanten bieten
  - `name`: liefert den exakten Name der Aufzählungskonstante
  - `ordinal`: liefert die Ordinalzahl der Aufzählungskonstante
  - `toString`: liefert den Name der Aufzählungskonstante
    - Kann eine lesbarere Repräsentation des Namens liefern

```
...
Color c = Color.GREEN;
System.out.println(c.name());          // GREEN
System.out.println(c.ordinal());       // 1
System.out.println(c.toString());      // GREEN
...
```

- Aufzählungstypen bieten
  - `values`: liefert ein Array zurück, das alle Aufzählungskonstanten des Enums enthält.

# Beispiel (1)

```
public enum Roman {  
    I(1), V(5), X(10), L(50), C(100), D(500), M(1000);
```

```
    private final int value;
```

Konstruktoraufruf

```
    Roman(int value) {  
        this.value = value;  
    }
```

Konstruktoren sind bei  
Aufzählungstypen implizit private

```
    public int getValue() {  
        return value;  
    }  
}
```

# Beispiel (2)

```
public class RomanApplication {  
  
    public static void main(String[] args) {  
        Roman r = Roman.V;  
        System.out.println(r.getValue());  
  
        for (Roman x : Roman.values()) {  
            System.out.print(x + " ");  
        }  
    }  
}
```

Ausgabe:

5  
I V X L C D M

# Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (Java SE 17 Edition)**, Oracle, 2021

# UML - Klassendiagramme

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Modell und Diagramm

- Ein Modell stellt eine Abstraktion eines Realitätsausschnitts dar.
  - Um Informationen verständlicher darzustellen
    - Analog zum Erstellen von Bauplänen von Gebäuden
  - Um essenzielle Systemaspekte aufzuzeigen
  - Zur Kommunikation
    - Mit Projektmitarbeitenden
    - Mit Kundschaft
  - Um komplexe Architekturen darstellen zu können
- Ein Diagramm ist die grafische Repräsentation eines Modells.

# Unified Modeling Language (UML)

- Die Unified Modeling Language (UML) ist eine standardisierte ausdruckstarke Modellierungssprache.
- Mit Hilfe der UML können Softwaresysteme besser entworfen, analysiert und dokumentiert werden.
- Begriff Unified bedeutet
  - Unterstützung des gesamten Entwicklungsprozesses.
  - Unabhängigkeit von Entwicklungswerkzeugen, sowie Programmiersprachen oder auch Anwendungsbereichen.
- Die UML ist aber nicht
  - ein Allheilmittel und vollständig,
  - ein vollständiger Ersatz für eine Textbeschreibung,
  - eine Methode oder Vorgehensmodell.

# UML Diagrammarten

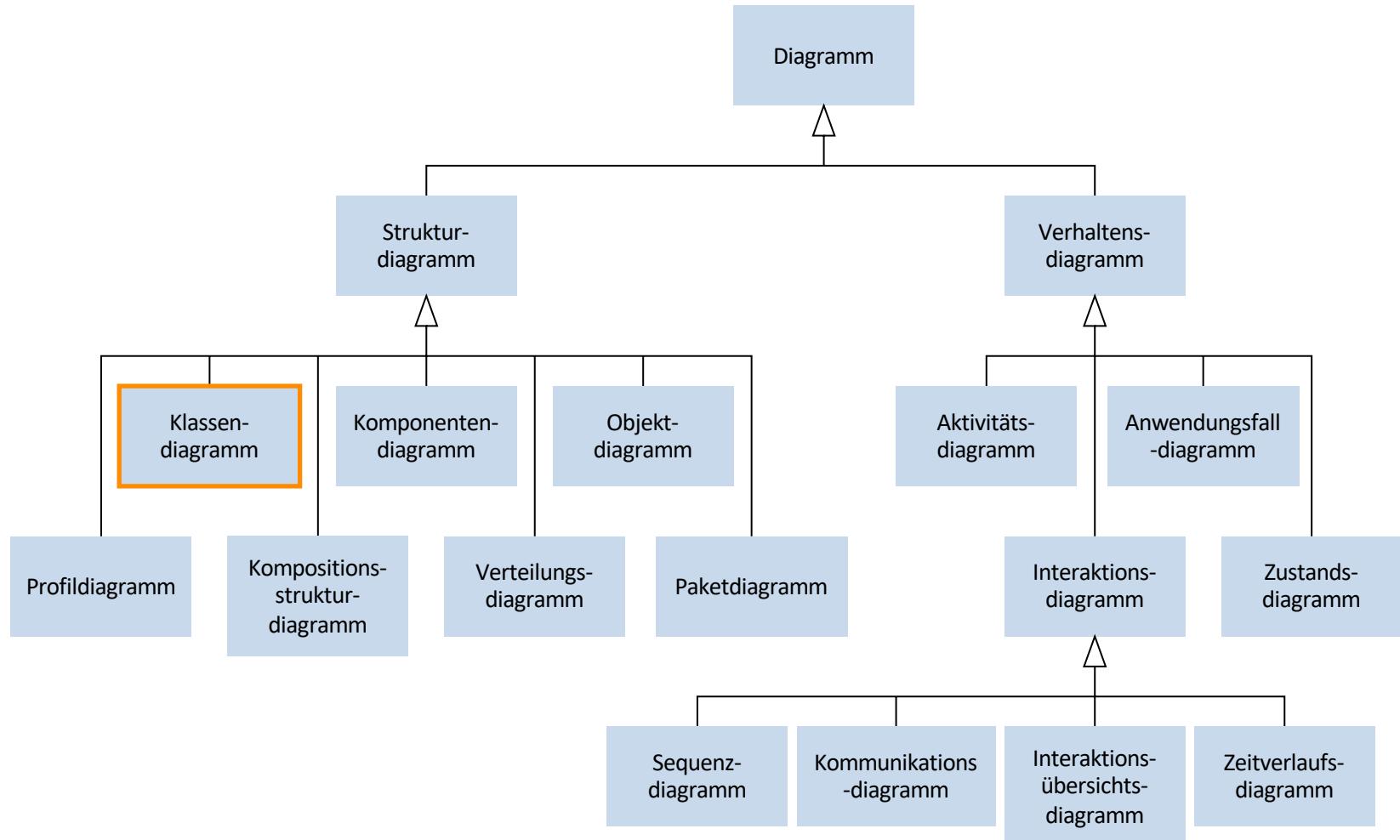


Abbildung angelehnt an „UML @ Classroom“, Seidl et al.

# Notation

## Sprachkonzept

Klasse

Klasse mit Abschnitten

Objekt

Anonymes Objekt

Objekt mit Attributen

## Notation

Klassenname

Klassenname

Attribut 1

Attribut 2

Operation 1

Operation 2

objektname

objektname:Klassenname

:Klassenname

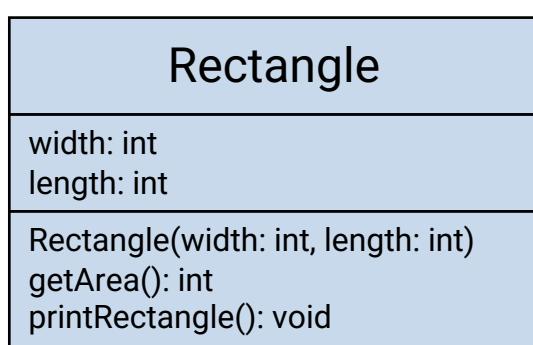
objektname

Attribut 1 = Wert 1

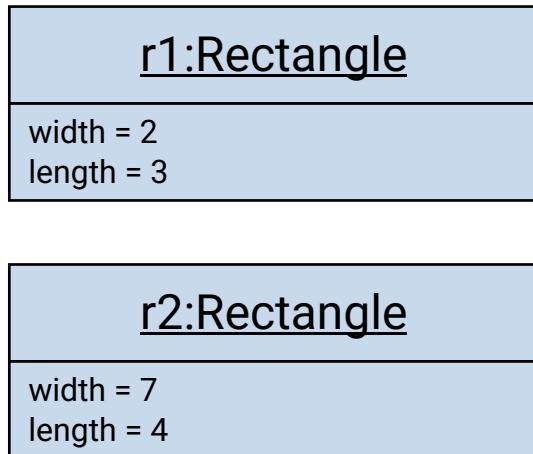
Attribut 2 = Wert 2

Attribute können auch bei den anderen Notationen für Objekte angegeben werden

# Klassen und Objekte

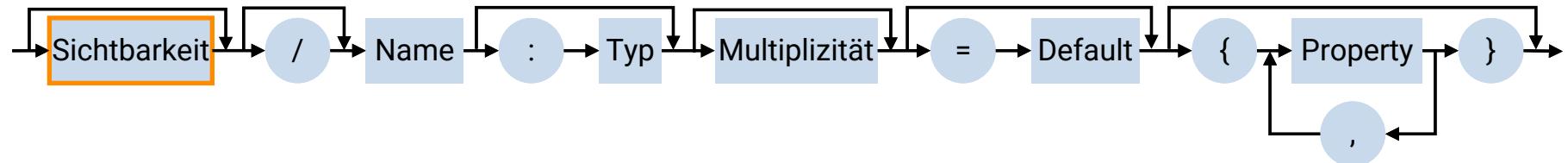


```
public class Rectangle {  
  
    int width;  
    int length;  
  
    Rectangle(int width, int length) {...}  
    int getArea() {...}  
    void printRectangle() {...}  
}
```



```
...  
Rectangle r1 = new Rectangle(2, 3);  
Rectangle r2 = new Rectangle(7, 4);  
...
```

# Attribute (1)

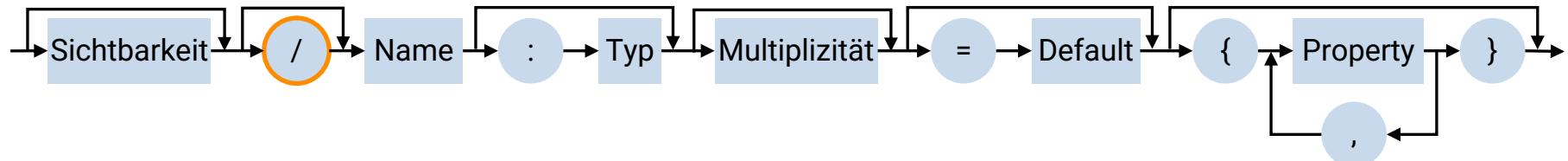


Name	Symbol	Zugriff auf Attribut
public	+	Objekte beliebiger Klassen
private	-	Nur innerhalb der Klasse
protected	#	Objekte derselben Klasse und deren Subklasse
package	~	Objekte, deren Klassen sich im selben Paket befinden

## Course

- name: String {readOnly}
  - leader: Person {readOnly}
  - participants: Person[1..\*] {unique, ordered}
  - numberParticipants: int = 1

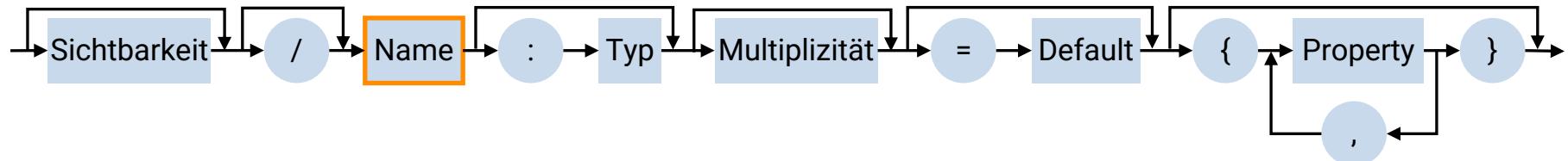
# Attribute (2)



- Kennzeichnung eines abgeleiteten Attributs
- Abgeleitete Attribute
  - Werden aus Werten anderer Attribute berechnet.
  - Benötigen in der Regel keinen dedizierten Speicher im Objekt.
  - Beispiel: Berechnung des Alters anhand des Geburtsdatums.

Course
- name: String {readOnly} - leader: Person {readOnly} - participants: Person[1..*] {unique, ordered} - numberParticipants: int = 1

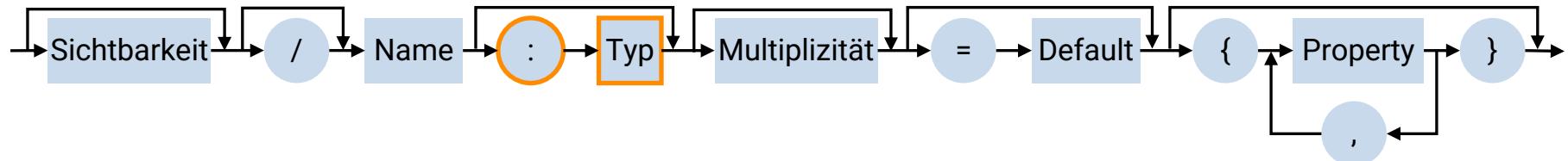
# Attribute (3)



- Bezeichnung des Attributs

Course
<pre>- name: String {readOnly} - leader: Person {readOnly} - participants: Person[1..*] {unique, ordered} - numberParticipants: int = 1</pre>

# Attribute (4)



- Angabe des Datentyps

Course
<ul style="list-style-type: none"><li>- name: String {readOnly}</li><li>- leader: Person {readOnly}</li><li>- participants: Person[1..*] {unique, ordered}</li><li>- numberParticipants: int = 1</li></ul>

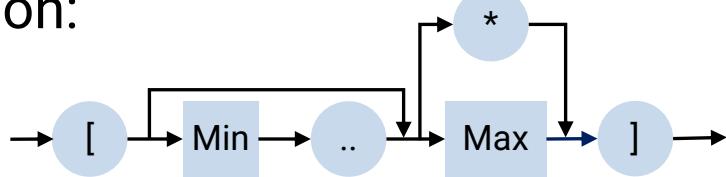
# Attribute (5)



- Spezifiziert wie viele Werte ein Attribut aufnehmen kann.

- Default Anzahl: 1

- Notation:

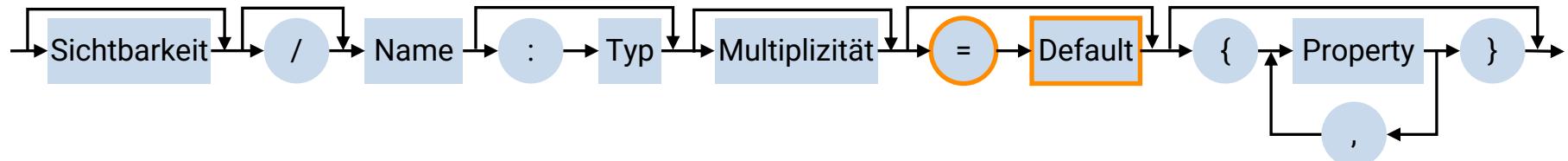


- Beispiele

- [0..1] null oder eins
- [0..\*] beliebig viele (entspricht [\*])
- [1..\*] beliebig viele aber zumindest eins
- [5] genau fünf

Course
- name: String {readOnly} - leader: Person {readOnly} - participants: Person[1..*] {unique, ordered} - numberParticipants: int = 1

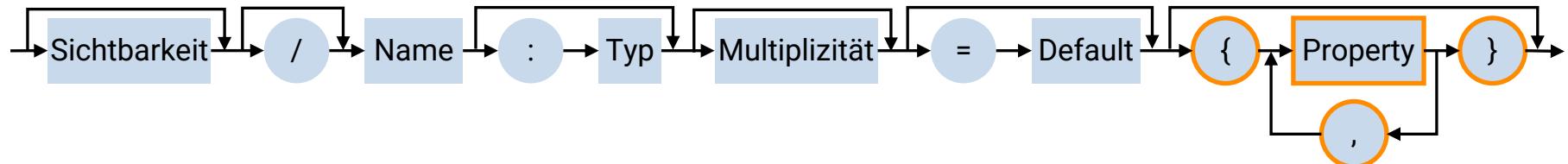
# Attribute (6)



- Standardwert, der verwendet wird, wenn bei der Erzeugung nicht explizit ein anderer Wert angegeben wird.

Course
- name: String {readOnly} - leader: Person {readOnly} - participants: Person[1..*] {unique, ordered} - numberParticipants: int = 1

# Attribute (7)

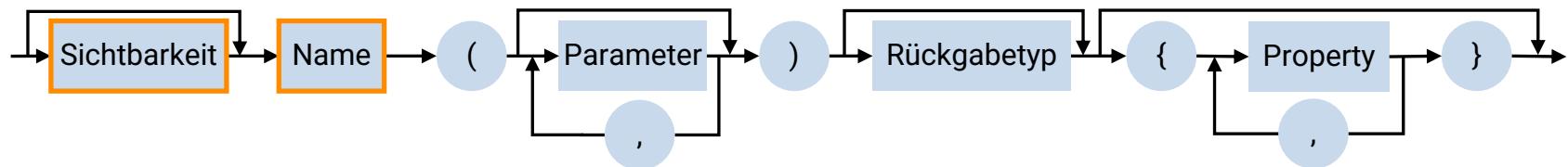


- Gibt zusätzliche Eigenschaften von Attributen an.
- Beispiele:

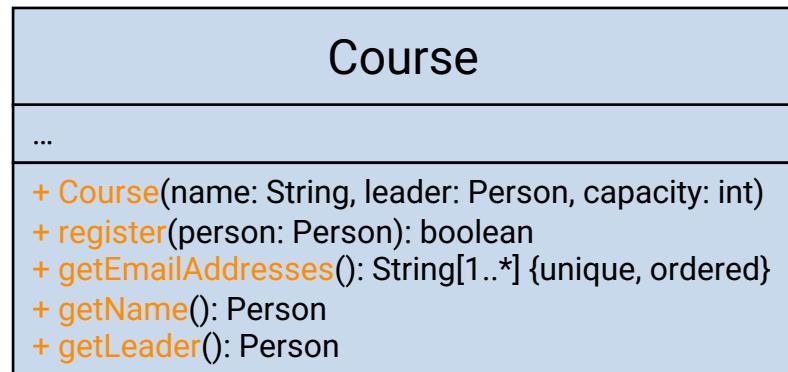
Property	Beschreibung
{readOnly}	Keine Änderung nach der Initialisierung
{unique}	Kann keine Duplikate enthalten
{non-unique}	kann Duplikate enthalten
{ordered}	fixe Reihenfolge
{unordered}	keine fixe Reihenfolge

Course
- name: String {readOnly} - leader: Person {readOnly} - participants: Person[1..*] {unique, ordered} - numberParticipants: int = 1

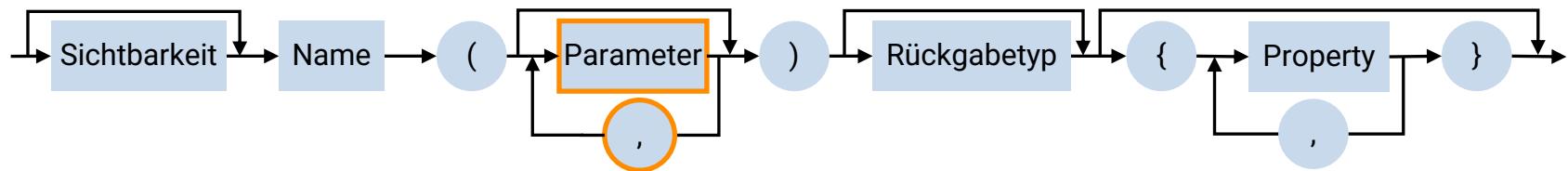
# Operationen (1)



- Sichtbarkeit und Name wie bei Attributen

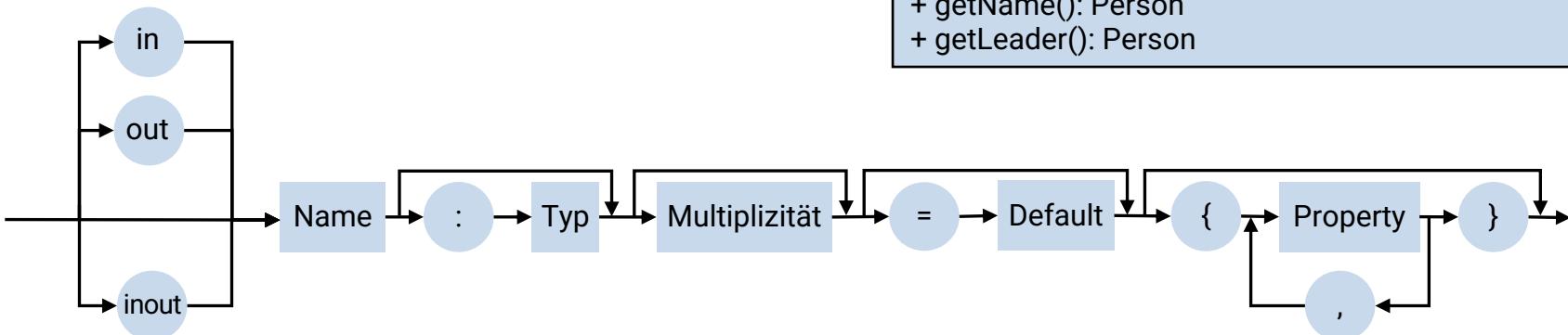
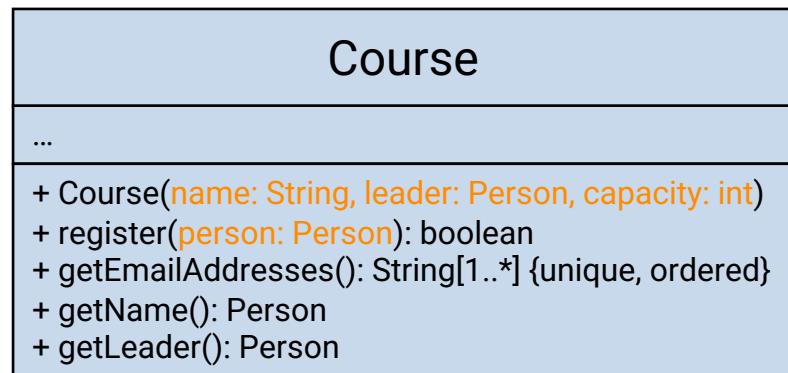


# Operationen (2)

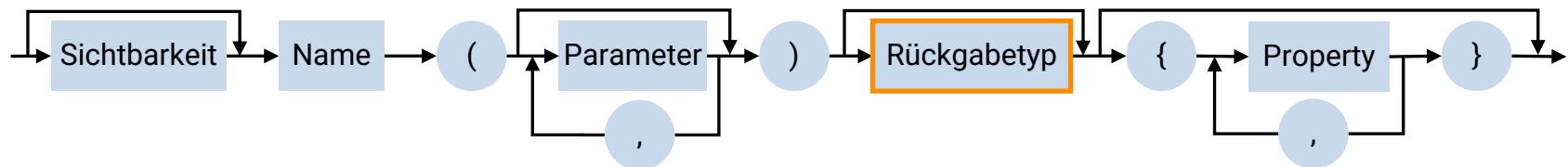


- Parameter

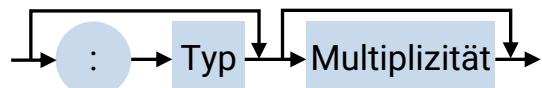
- Für Parameter gilt die folgende, an Attribute angelehnte, Notation.



# Operationen (3)

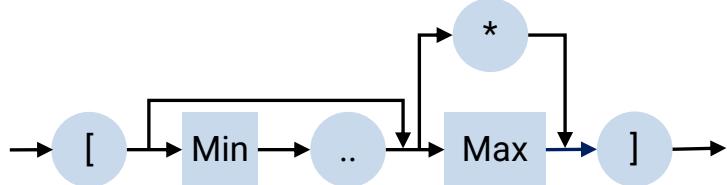


- Rückgabetyp

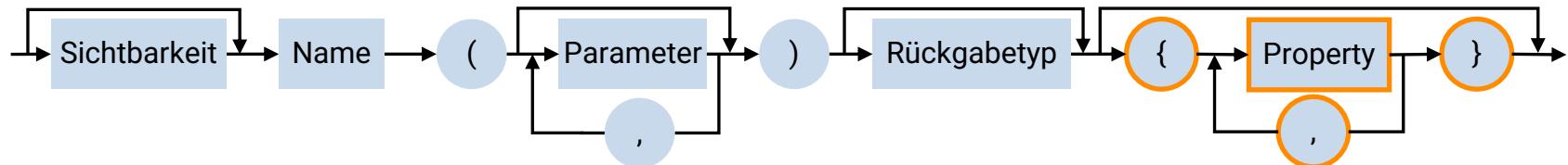


- Typ gibt den Datentyp des Rückgabewerts an.
- Multiplizität wie bei Attributen:

Course
...
+ Course(name: String, leader: Person, capacity: int)
+ register(person: Person): boolean
+ getEmailAddresses(): String[1..*] {unique, ordered}
+ getName(): Person
+ getLeader(): Person



# Operationen (4)



- Gibt zusätzliche Eigenschaften von Operationen an.
- Beispiele:

Property	Beschreibung
{abstract}	Die Operation ist abstrakt. Alternative: die Operation kursiv schreiben
{leaf}	Die Operation darf in Unterklassen nicht überschrieben werden.
{query}	Die Operation verändert den Objektzustand nicht.
{sequential}	Angaben zum Kontrollfluss über das Objekt.
{guarded}	
{concurrent}	

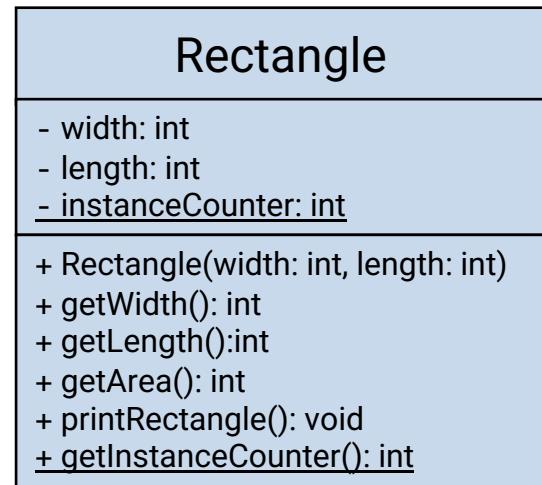
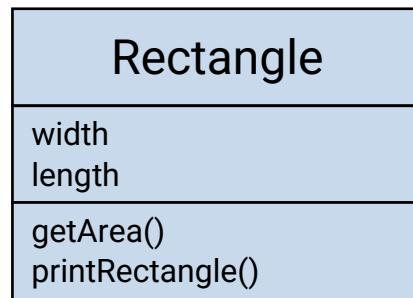
Course
...
+ Course(name: String, leader: Person, capacity: int) + register(person: Person): boolean + getEmailAddresses(): String[1..*] {unique, ordered} + getName(): Person + getLeader(): Person

# Klassenvariable und Klassenoperation

- Merkmale sind standardmäßig auf Exemplarebene definiert.
  - Wird von Attribut oder Operation gesprochen sind im allgemeinen Exemplarattribut und Exemplaroperation gemeint.
- Unterstreichen kennzeichnet Klassenattribute bzw. Klassenvariablen
  - Sie können auch statische Attribute genannt werden
- Das selbe gilt auch für Klassenoperationen bzw. statische Operationen.

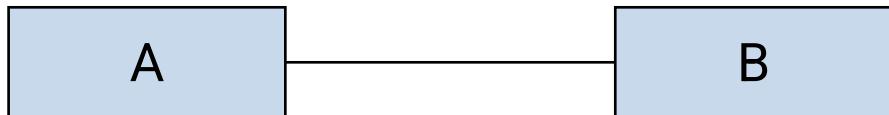
Rectangle
<p>- width: int - length: int <u>- instanceCounter: int</u></p>
<p>+ Rectangle(width: int, length: int) + getWidth(): int + getLength(): int + getArea(): int + printRectangle(): void <u>+ getInstanceCounter(): int</u></p>

# Detailierungsgrad



# Assoziation

- Grundform (binäre Assoziation)



- Zusätzliche optionale Angaben

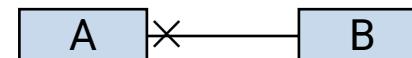
- Benennung in der Mitte der Kante
  - Anzeigen der Leserichtung ist durch ► bzw. ◀ möglich
- Rollen an den Enden der Assoziation
- Multiplizitäten an den Enden, wie bei Attributen
- Angabe von Eigenschaften
  - {ordered}
  - {unique}
  - ...

- Navigationsangaben

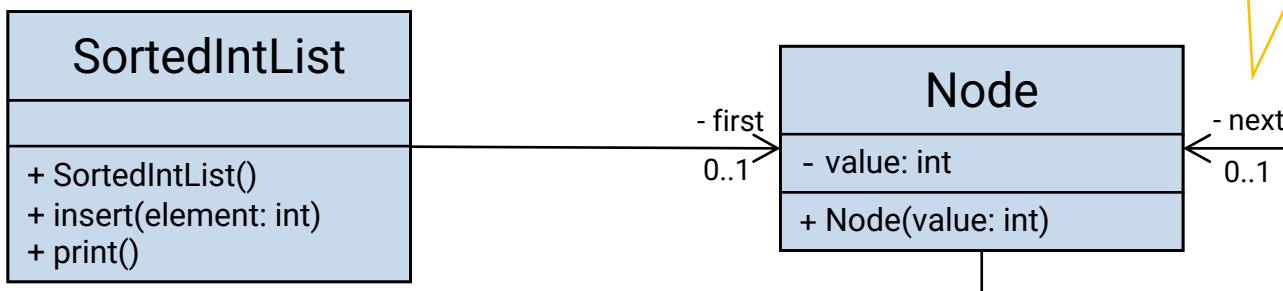
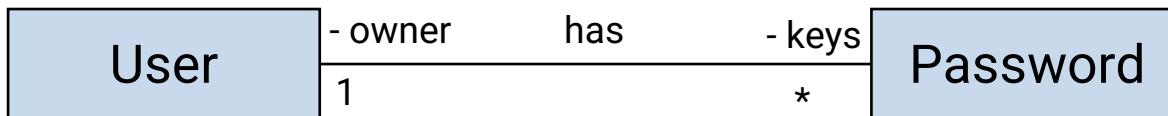
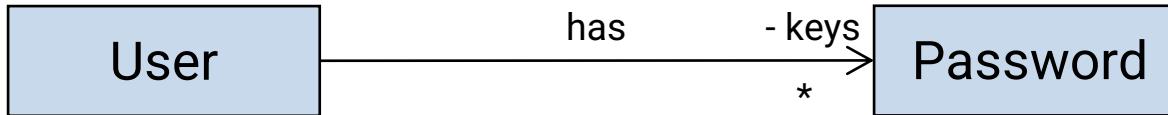
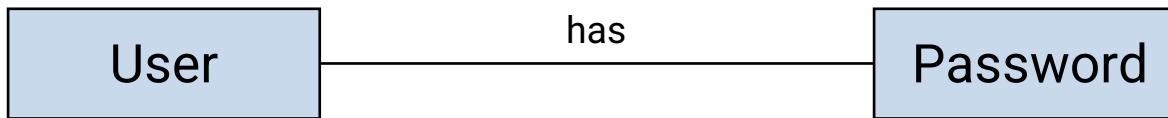
- Zulässige Navigationsrichtung (durch Pfeil →)



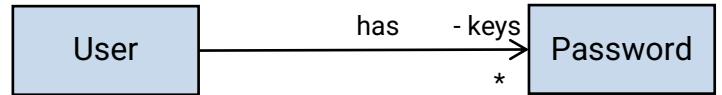
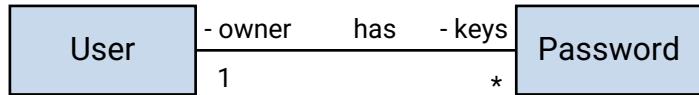
- Keine Navigation erlaubt (durch X ✗ )



# Beispiele



# Beispiel (Java Umsetzung)



```
public class User {  
    private Password[] keys;  
    ...  
}
```

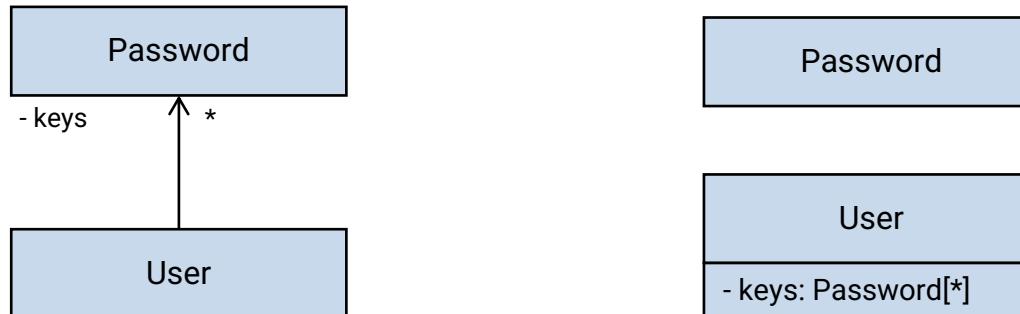
```
public class User {  
    private Password[] keys;  
    ...  
}
```

```
public class Password {  
    private User owner;  
    ...  
}
```

```
public class Password {  
    ...  
}
```

# Darstellungsmöglichkeiten für Assoziationen

- Beide Repräsentationen sind äquivalent.
  - In der Praxis ist die linke Darstellung zu bevorzugen, da die Assoziation sofort ersichtlich ist.



- Äquivalenter Java Code

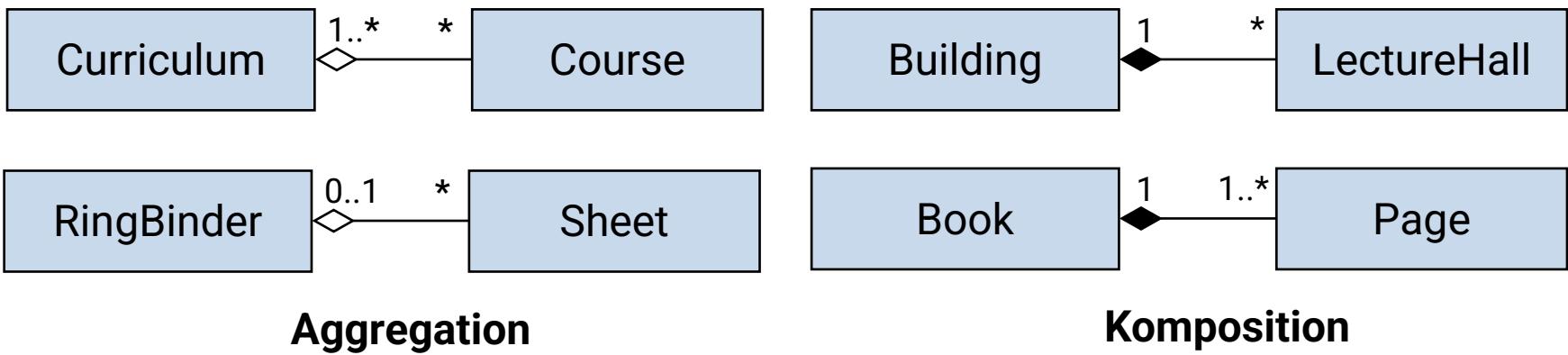
```
public class Password {  
    ...  
}
```

```
public class User {  
    private Password[] keys;  
    ...  
}
```

# Aggregation

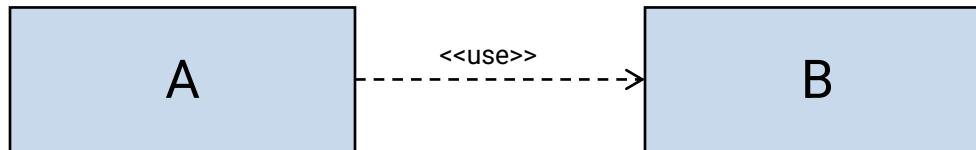
- Aggregation ist eine spezielle Assoziation (Teile-Ganzes-Beziehung).
- Komposition ist eine speziellere (strengere) Form der Aggregation mit folgenden Einschränkungen.
  - Ein Teil darf Kompositionsteil höchstens eines Ganzen sein.
  - Ein Teil existiert nur so lange wie sein Ganzes.
- Eigenschaften
  - Transitiv
  - Asymmetrisch
- Beispiele

Der Hörsaal existiert nur, wenn auch das Gebäude existiert

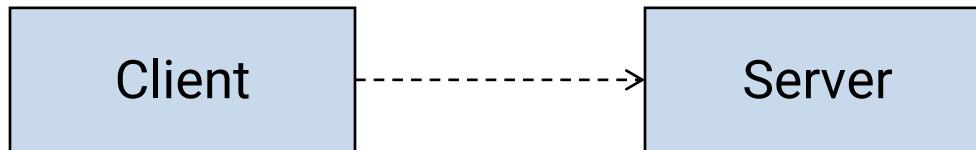


# Abhängigkeitsbeziehungen

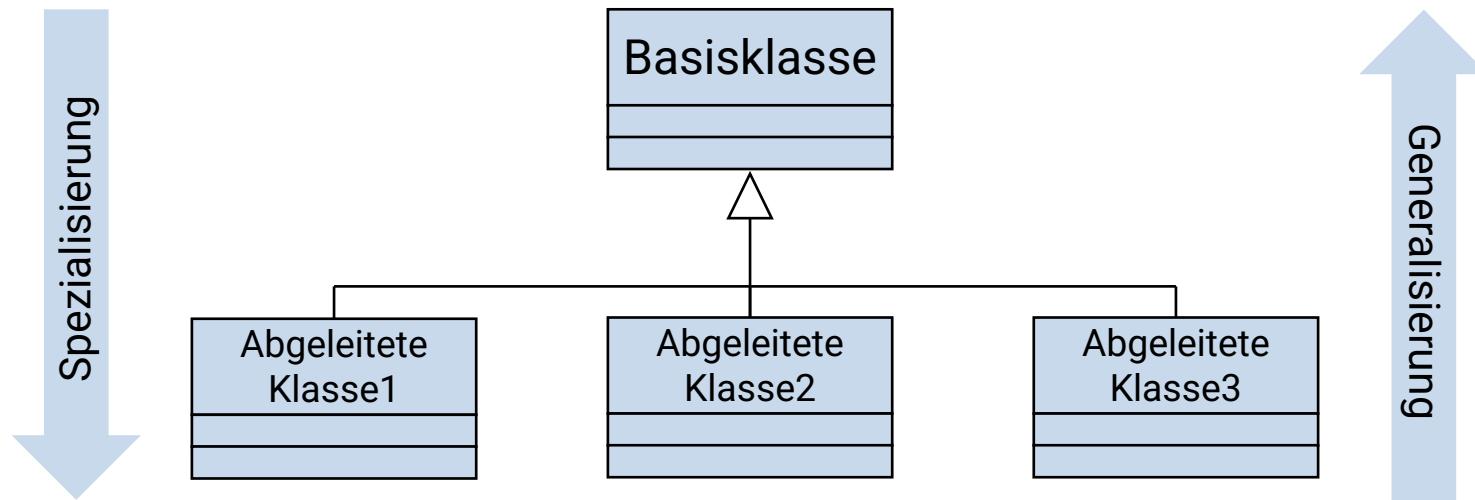
- Abhängigkeitsbeziehungen stellen allgemeine, nicht näher spezifizierte Abhängigkeiten dar.
- Notation:



- A hängt von B ab
- Änderungen in B können zu Änderungen in A führen
  - A beinhaltet Attribute/Methodenparameter vom Typ B
  - A erzeugt Objekte vom Typ B
- Beispiel: Client- Server



# Generalisierung



# Quellen

- Martina Seidl, Marion Brandsteidl, Christian Huemer, Gerti Kappel: **UML @ Classroom: Eine Einführung in die objektorientierte Modellierung**, dpunkt.verlag, 2012
- Joachim Goll: **Methoden und Architekturen der Softwaretechnik**, Vieweg + Teubner Verlag, 2011

# Vererbung und Polymorphie

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck



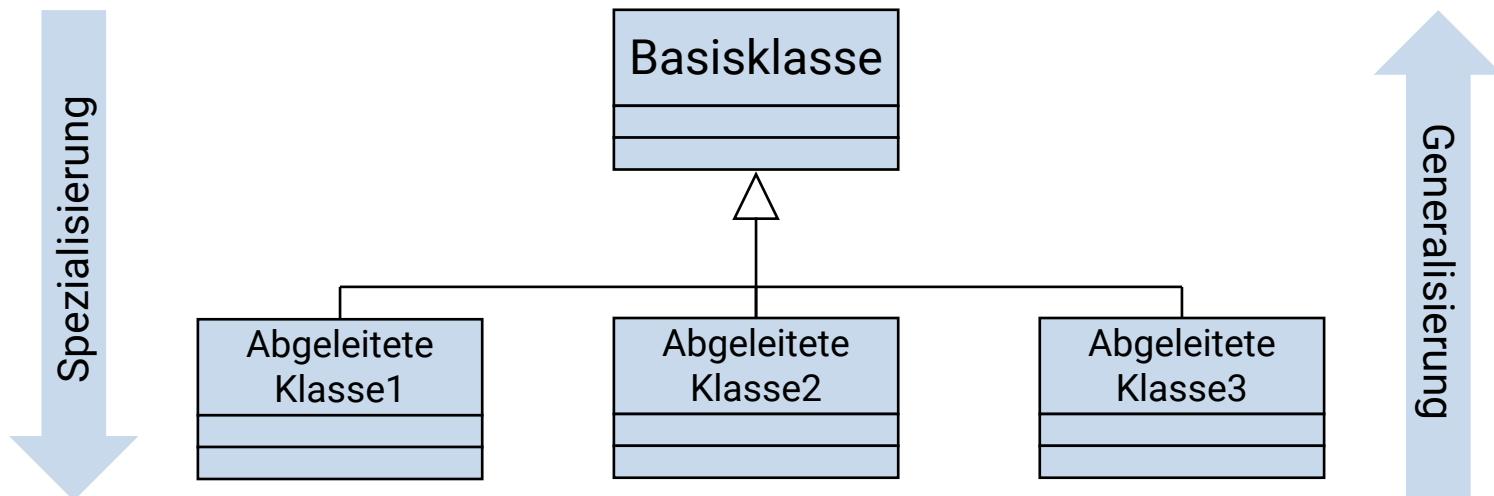
# Vererbung allgemein

# Vererbung allgemein

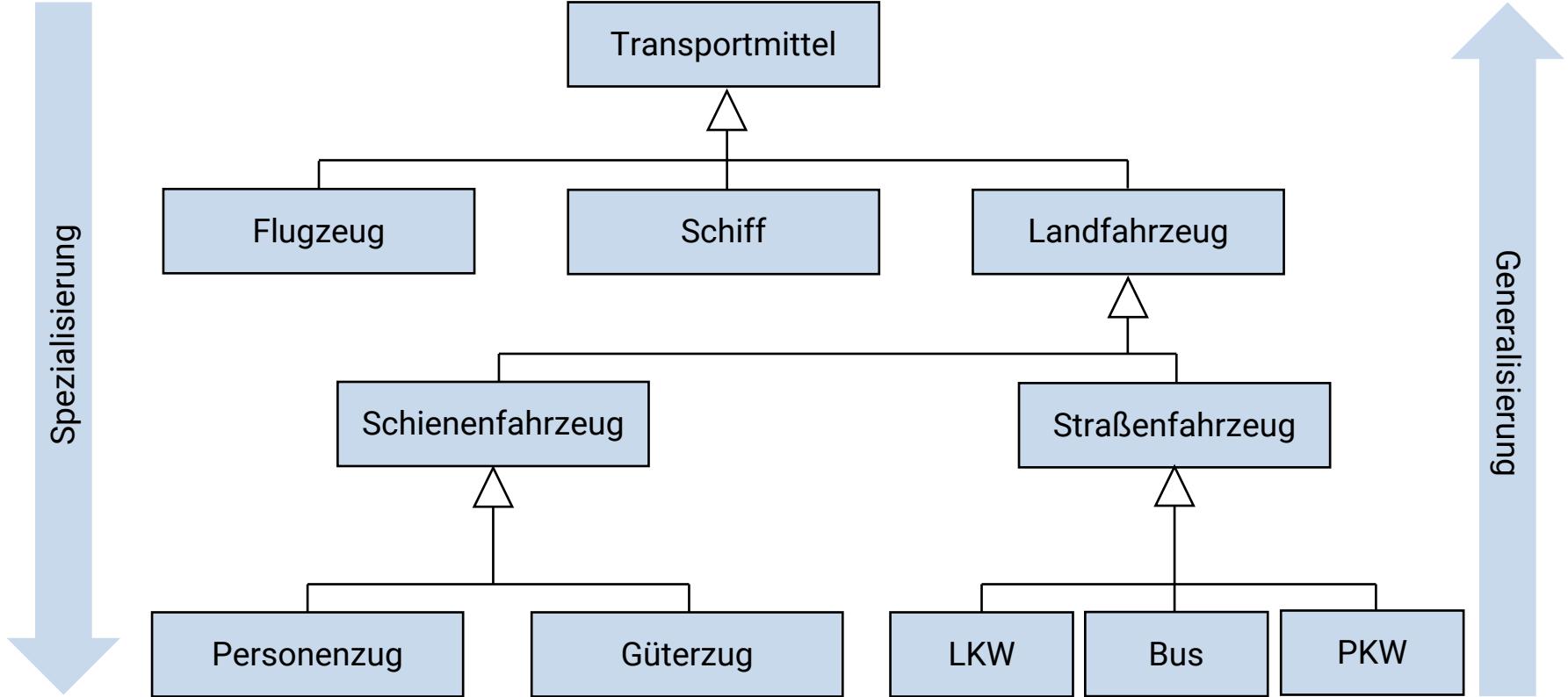
- Klassen modellieren Dinge der realen Welt.
- Diese Dinge kommen oft in verschiedenen Varianten vor, die durch Klassifikation hierarchisch gegliedert werden können.
- Klassen sind in der Regel Gruppierungen von gleichartigen Objekten.
- Programme sollten
  - mit den verschiedenen Varianten arbeiten und
  - in bestimmten Situationen Varianten nicht unbedingt unterscheiden (gleich behandeln).
- Java bietet die Möglichkeit
  - hierarchische Klassenstrukturen zu bilden und
  - Varianten von Objekten gleich zu behandeln.

# Unterklassen und Oberklassen

- Eine Klasse S ist eine Unterklasse der Klasse A, wenn S die Spezifikation von A erfüllt, umgekehrt aber A nicht die Spezifikation von S (A ist eine Oberklasse von S).
  - Beziehung zwischen A und S wird **Spezialisierung** genannt.
  - Beziehung zwischen S und A wird **Generalisierung** genannt.



# Beispiel



# Prinzip der Ersetzbarkeit

- Wenn eine Klasse **B** eine Unterklasse der Klasse **A** ist, dann können in einem Programm alle Exemplare der Klasse **A** durch Exemplare der Klasse **B** ersetzt werden und es gelten weiterhin alle zugesicherten Eigenschaften der Klasse **A**.
  - Exemplare der Unterklasse sind gleichzeitig Exemplare der Oberklasse in Bezug auf die der Oberklasse zugrunde liegende Spezifikation.
- Konsequenzen für Unterklassen
  - Aus der Oberklasse stammende Vorbedingungen für Operationen können nicht verschärft werden, sie dürfen lediglich eingehalten oder abgeschwächt werden.
  - Aus der Oberklasse stammende Nachbedingungen für Operationen dürfen nicht gelockert werden, sie dürfen lediglich eingehalten oder verschärft werden.
  - Aus der Oberklasse stammende Invarianten müssen immer eingehalten werden.

# Kategorien von Klassen

- Klassen können kategorisiert werden.
  - Die Kategorisierung ist abhängig davon, in welchem Umfang sie selbst für die von ihnen spezifizierte Schnittstelle auch Methoden anbieten.
- Kategorien:
  1. Konkrete Klassen
  2. Schnittstellenklassen
  3. Abstrakte Klassen

# Konkrete Klassen

- Stellen für alle von der Klasse spezifizierten Operationen auch Methoden bereit.
- Es können Exemplare erzeugt werden.
- Wurden bisher in dieser Vorlesung besprochen.

# Schnittstellenklassen

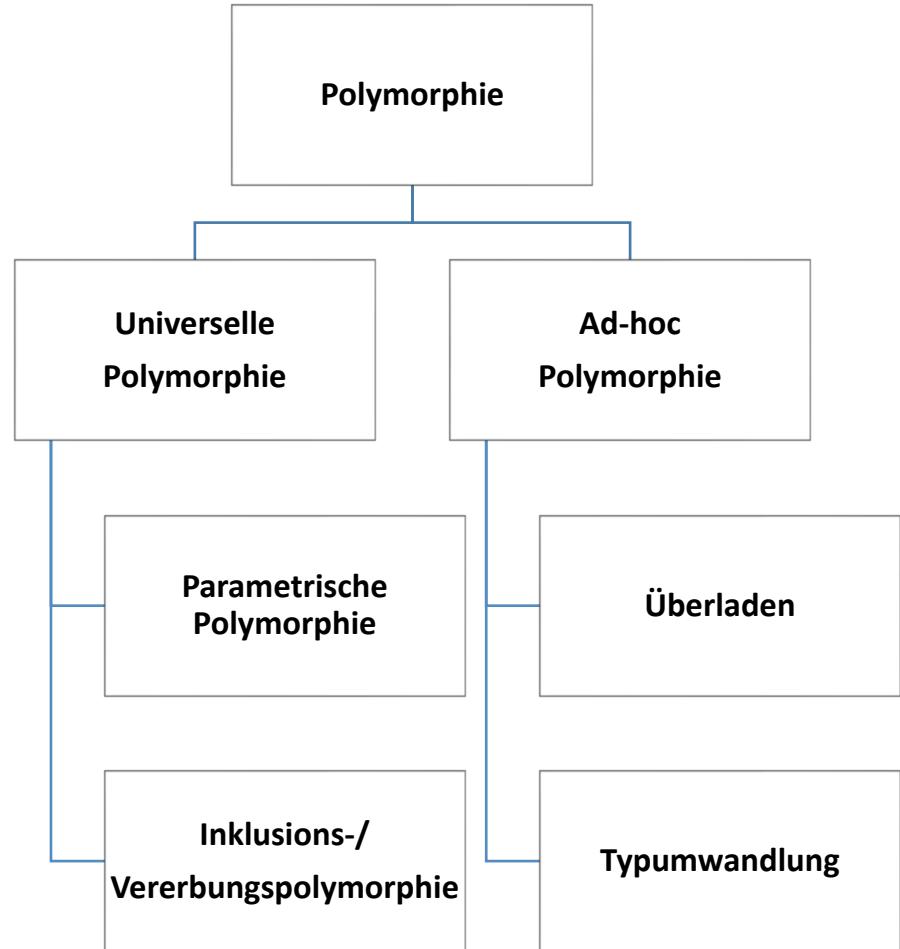
- Schnittstellenklassen (engl. Interfaces)
  - Dienen alleine der Spezifikation einer Menge von Operationen ("was").
  - Für keine Operation wird eine Implementierung bereitgestellt ("wie").
  - Es können keine Exemplare erzeugt werden.
  - Trennung Spezifikation vs. Implementierung
- „Eine Schnittstelle implementieren“
  - Eine Unterklasse implementiert eine Schnittstellenklasse, wenn sie alle in der Schnittstellenklasse spezifizierten Operationen implementiert.
- Einsatz
  - Bei statisch typisierten Programmiersprachen (wie Java)
  - Als gemeinsamer Typ für Klassen, welche dieselbe Schnittstellenklasse implementieren.

# Abstrakte Klassen

- Abstrakte Klassen
  - Zwischenstufe zwischen Schnittstellenklassen und konkreten Klassen.
  - Es kann keine direkten Exemplare geben.
  - Alle Exemplare einer abstrakten Klasse müssen gleichzeitig Exemplare einer nicht abstrakten Unterklasse sein.
  - Stellen meist für mindestens eine der spezifizierten Operationen keine Implementierung bereit.
- Abstrakte Methoden
  - Erlauben eine Operation für eine Klasse zu definieren, ohne dafür eine Methodenimplementierung zur Verfügung zu stellen.
  - Methode dient nur zur Spezifikation.
  - Eine Implementierung erfolgt in einer abgeleiteten Unterklasse.

# Polymorphie

- Polymorph = „vielgestaltig“
- Eine Variable oder eine Methode kann gleichzeitig mehrere Typen haben.
- Objektorientierte Sprachen sind **polymorph**.  
(konventionelle Sprachen wie zum Beispiel Pascal sind **monomorph**)
- Verschiedene Arten (siehe rechts)



# Polymorphie (Arten)

- Universelle Polymorphie
  - Ein Name oder Wert kann theoretisch unendlich viele Typen besitzen.
  - Die Implementierung einer universell polymorphen Operation führt generell gleichen Code unabhängig von den Typen Ihrer Argumente aus.
- Ad-hoc-Polymorphie
  - Ein Name oder ein Wert kann nur endlich viele verschiedene Typen besitzen.
  - Typen sind zur **Übersetzungszeit** bekannt.
  - Ad-hoc-polymorphe (also überladene) Operationen können abhängig von den Typen ihrer Argumente unterschiedlich implementiert sein.

# Polymorphie in dieser Vorlesung

- Ad-hoc-Polymorphie wurde schon behandelt
  - Siehe Typumwandlung
  - Siehe Überladen von Methoden
- Universelle Polymorphie
  - Parametrische Polymorphie wird noch behandelt (Generische Programmierung).
  - Inklusionspolymorphie bzw. Vererbungspolymorphie ist zentrales Thema dieses Foliensatzes.
    - Vererbung der Spezifikation
    - Vererbung der Implementierung

# Vererbungspolymorphie

- Einer Variable können Objekte unterschiedlichen Typs zugeordnet werden.
  - Der Typ einer Variable beschreibt nur die Schnittstelle.
  - Objekte, deren Klassen die Schnittstellen erfüllen, können zugewiesen werden.
  - Beim Aufruf einer Operation (zur Laufzeit) wird die entsprechende Methode abhängig vom Objekt ausgewählt.
- Späte Bindung (dynamisches Binden)
  - Im Hauptprogramm wird zufällig eine Implementierung ausgewählt.
  - Wie wird die richtige Methode gefunden?
  - Zur Laufzeit wird abhängig vom momentan zugewiesenen Objekt die entsprechende Methode aufgerufen – dynamisches Binden.
  - Ein gegebener Methodenaufruf wird abhängig vom Kontext in unterschiedliche Abläufe umgesetzt.



# Vererbung der Spezifikation in Java

# Interfaces (1)

- Inhalt

- Methoden-Spezifikation ohne Rumpf
  - Sind **abstrakte** Methoden
  - Sind **immer** public
- Konstanten
  - Alle Variablen sind immer public static final (nur Konstanten).
- Statische Methoden
  - Sind public oder private
- Private objektbezogene Methoden
- Default-Methoden
  - Sind implizit public

# Interfaces (2)

- Implementierung
  - Klassen können Interfaces mit `implements` implementieren.
  - Eine Klasse kann mehrere Interfaces implementieren.
  - Ein Interface kann von mehreren Klassen implementiert werden.
  - Die Implementierung eines Interfaces muss die Spezifikation erfüllen.
    - Alle Methoden müssen implementiert werden.
- Datentypen
  - Ein Interface stellt einen Datentyp dar.
  - Alle Exemplare von Klassen, die das Interface implementieren, sind mit dem Interfacetyp zuweisungskompatibel.
- Erzeugung von Objekten
  - Es können keine Exemplare von Interfaces erzeugt werden.
  - Einem Interfacetyp zugewiesene Exemplare sind immer Exemplare konkreter Klassen, welche das Interface implementieren.
  - Interfaces können keine Konstruktoren bereitstellen.

# Beispiel Komplexe Zahlen (1)

- Einfaches Interface für komplexe Zahlen

```
public interface Complex {  
    double getReal();  
    double getImaginary();  
    double getDistance();  
    double getPhase();  
    Complex multiply(Complex other);  
    Complex add(Complex other);  
}
```

- Jede Klasse, die dieses Interface implementiert, muss die sechs angegebenen Methoden implementieren.
- Alle deklarierten Methoden sind implizit public und abstract.
- Beispiel:

 <src/at/ac/uibk/pm/inheritance/complexnumbers/Complex.java>

 <src/at/ac/uibk/pm/inheritance/complexnumbers/Polar.java>

 <src/at/ac/uibk/pm/inheritance/complexnumbers/Cartesian.java>

# Beispiel Komplexe Zahlen (2)

- Klasse `Cartesian` und `Polar` implementieren das Interface `Complex`.
- Beide Klassen implementieren die vorgegebenen Methoden (jeweils unterschiedlich).
- Beide Klassen geben zusätzlich einen Konstruktor an.
  - Die Form des Konstruktors wird nicht vom Interface vorgegeben.
  - Es könnten noch mehrere Konstruktoren angegeben werden.
  - Es können zusätzliche Methoden angegeben werden (z.B. `subtract`) – d.h. es gibt keine Einschränkung für weitere Methoden.
- Beide Klassen sind gleichberechtigte und unabhängige Implementierungen von `Complex`.

# Polymorphie

- Interfaces definieren einen Datentyp.
  - Können in Variablendeklarationen, Parameterlisten, und als Ergebnistyp von Methoden verwendet werden.
- Complex complexNumber = new Polar(3, 5);
- Warum funktioniert das Beispiel?
  - **Vererbungspolymorphie**
  - Die Klasse Polar implementiert das Interface Complex.
  - Objekte dieser Klasse können einer Variable vom Typ Complex zugewiesen werden (Prinzip der Ersetzbarkeit).
  - Die Variable complexNumber kann daher auf unterschiedliche Objekte zeigen, der Typ der zugewiesenen Klasse wird zur Laufzeit bestimmt (dynamischer Typ).

# Statischer/Dynamischer Typ

## Statischer Typ

- Typ der Variable laut Deklaration
- Bestimmt, welche Objektvariablen und Methoden angesprochen werden können.
- Kompatibilitätsüberprüfung

## Dynamischer Typ

- Typ zur Laufzeit (abhängig vom tatsächlich zugewiesenen Objekt)
- Kann sich nach jeder (gültigen) Zuweisung ändern.
- Er bestimmt, welche Methoden wirklich aufgerufen werden.

# Beispiel dynamischer Typ

```
Complex position = new Polar(2, 0);
position = new Cartesian(3, 1);
position = Math.random() > 0.5 ? position : new Polar(1, 1);
position = position.multiply(position);
```

# Codebeispiele

```
Polar startPosition = new Polar(2, 1);
Cartesian endPosition = new Cartesian(5, 0);
Cartesian midPosition = (Cartesian) startPosition.add(new Polar(1, 0));
```



# < > Favor Abstract Over Concrete Types

```
Complex startPosition = new Polar(2, 1);
Complex endPosition = new Cartesian(5, 0);
Complex midPosition = startPosition.add(new Polar(1, 0));
```



- Vorher:
  - Konkrete Typen für Variablen verwendet
  - Keine Kompatibilität zwischen Variablen
  - Casts notwendig für Zuweisung
- Nachher:
  - Flexiblerer Code
  - Keine Casts mehr notwendig
  - Spezifikation der Funktionalität, nicht konkrete Implementierung wird genutzt.

# Implementierung mehrerer Interfaces

- Eine Klasse kann mehrere Interfaces implementieren.
- Mehrere Interfaces können die gleiche Methode vorschreiben (gleiche Signatur).
- Die Klasse, die diese Interfaces implementiert, muss die Methode nur einmal implementieren.

```
public interface Printable {  
    void print();  
}
```

```
public class Cartesian implements Complex, Printable {  
    // methods for interface Complex as in previous examples  
    ...  
    @Override  
    public void print() {  
        System.out.println(real + " + " + imaginary + "i");  
    }  
}
```

# Ableiten (bei Interfaces)

- Ein Interface kann von einem anderen Interface abgeleitet werden.
- Es wird das Schlüsselwort `extends` verwendet.
- Ein Interface kann mehrere Super-Interfaces haben.
- Das Interface übernimmt alle Konstanten und alle öffentlichen, nicht statischen Methoden aus den Super-Interfaces.

# Konstanten

- In Interfaces können Konstanten deklariert werden.
- Konstanten werden weitervererbt.
- Wird normalerweise **nicht** empfohlen!
  - **Sollte in einer sauberen Implementierung vermieden werden (Mischung zwischen Implementierung und Spezifikation)!**

# Statische Methoden

- In Interfaces können statische Methoden deklariert werden.
- Sie können public oder private sein.
- Statische Methoden von Interfaces werden, egal ob public oder private, nicht weitervererbt.
  - Ein Zugriff auf diese Methoden ist somit nur über das deklarierende Interface möglich.
- Statische Methoden können nicht zusätzlich abstract oder default sein.
- Sind alle Methoden eines Interfaces statisch, deutet das auf Designprobleme hin.
  - Sollte typischerweise als finale Klasse mit privatem Konstruktor umgesetzt werden.

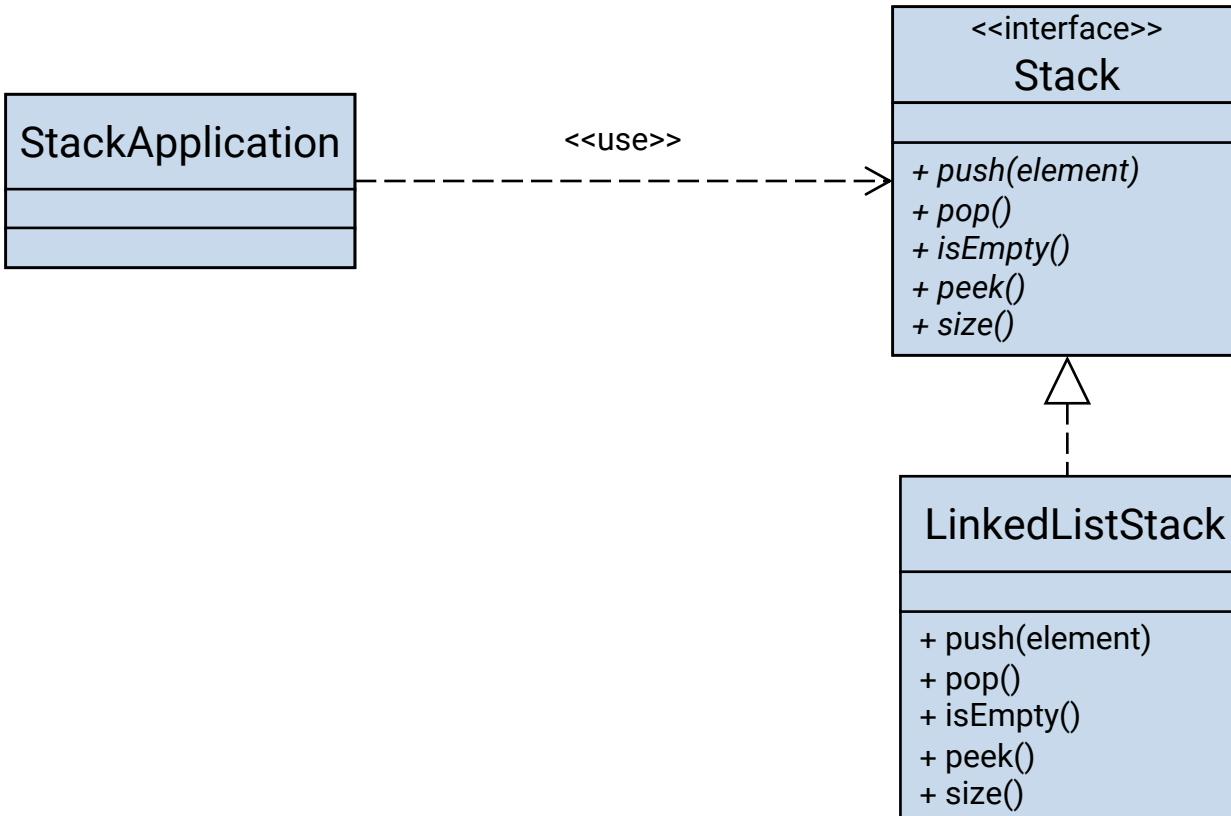
# Default Methoden

- In Interfaces können default-Methoden deklariert werden, welche eine Default-Implementierung bereitstellen.
  - Die default-Implementierung wird herangezogen, wenn die default-Methode nicht überschrieben wird.
- default-Methoden können nicht zusätzlich abstract oder static sein.
- Auf eine überschriebene default-Methode kann durch InterfaceName.super.methodName(...) zugegriffen werden.
- Werden zwei default-Methoden oder eine abstrakte- und eine default-Methode mit derselben Signatur geerbt, kommt es zu einem Kompilierfehler.
  - Dieser Fehler kann durch überschreiben der Methode behoben werden.

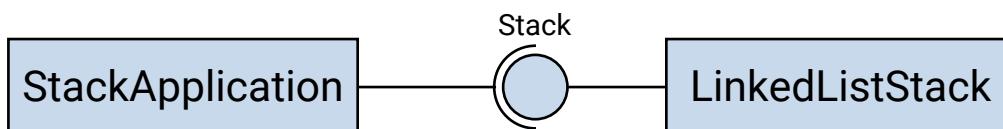
# Einsatz von Interfaces

- Interfaces erlauben die isolierte Entwicklung von Implementierungen.
- Anwendungen können nur auf Interfaces aufbauen.
  - Anwendung deklariert Variablen vom Interface-Typ.
  - Zukünftige Klassen, die das Interface implementieren, können sofort in die Anwendung eingebunden werden.
- Leere Interfaces
  - Marker-Interfaces (werden noch besprochen).
- Functional Interfaces (werden noch besprochen).

# Einschub: UML-Notation für Interfaces



- „Lollipop“-Notation





# **Vererbung der Implementierung in Java**

# Vererbung und Interfaces

- Ein Interface fixiert gemeinsame Eigenschaften von Klassen.
  - Klassen erfüllen den gleichen Zweck (im Interface angegeben).
  - Sind ansonsten unabhängig.
- Bei vielen Klassen beruht die Verwandtschaft nicht nur auf gleichen Eigenschaften sondern auf **Erweiterung** und **Modifikation** von Eigenschaften.

# Vererbung der Implementierung (allgemein)

- Unterklassen erben die in der Oberklasse bereits implementierte Funktionalität.
- Unterklassen erben
  - Verpflichtungen
  - Alle Methoden
  - Alle Daten
- Funktionalität kann komplett übernommen werden **oder** von der UnterkLASSE verändert/erweitert werden (sogenanntes **Überschreiben**).



Sofern diese zur Schnittstelle der Oberklasse gehören oder durch Sichtbarkeitsregeln freigegeben wurden.

# Überschreiben

- Eine Klasse kann Methoden, die sie von der Superklasse erbt, implementieren (**Überschreiben**).
- Wird die Methode auf ein Exemplar der Unterklasse aufgerufen, dann wird die überschriebene Implementierung aufgerufen.
- Folgende Regeln gelten dabei:
  - Die Signatur der überschriebenen Methode muss übernommen werden.
    - Ansonsten: Überladen (auch in Subklassen möglich)
  - Der Zugriffsschutz darf gelockert werden (z.B. protected in public).
  - Der Rückgabetyp darf vom Rückgabetyp der zu überschreibenden Methode abgeleitet werden.
  - Der Rumpf kann komplett ersetzt werden oder es kann auf die geerbte Implementierung zugegriffen und diese erweitert werden.

# Zugriffsschutz

	Klasse	Paket	Subklasse(n)	Alle Klassen
<b>private</b>	✓			
default	✓	✓		
<b>protected</b>	✓	✓	✓	
<b>public</b>	✓	✓	✓	✓

# Beispiel Fahrzeugverwaltung

- Klasse KFZ (Vehicle)
  - Objektvariablen
    - Nummernschild
    - Standort
  - Methode zur Ausgabe der KFZ-Daten
  - Methode zum Setzen des Nummernschilds
  - Methode zum Setzen des Standorts
- Klasse LKW (Truck)
  - Objektvariablen
    - Nummernschild
    - Standort
    - Ladung
  - Methode zur Ausgabe der KFZ-Daten
  - Methode zum Setzen des Nummernschilds
  - Methode zum Setzen des Standorts
  - Methode zum Abfragen der Ladung
  - Methode zum Ändern der Ladung

# Fahrzeugverwaltung: Lösung ohne Vererbung

- Vehicle

 <src/at/ac/uibk/pm/inheritance/vehicles/noinheritance/Vehicle.java>

- Truck

 <src/at/ac/uibk/pm/inheritance/vehicles/noinheritance/Truck.java>

- VehicleApplication (Main)

 <src/at/ac/uibk/pm/inheritance/vehicles/noinheritance/VehicleApplication.java>

Vehicle
- licencePlate: String - location: String
+ Vehicle(licencePlate: String, location: String) + getInfo(): String + setLicencePlate(licencePlate: String) + setLocation(location: String)

Truck
- licencePlate: String - location: String - cargo: String
+ Truck(licencePlate: String, location: String, cargo: String) + getInfo(): String + setLicencePlate(licencePlate: String) + setLocation(location: String) + getCargo(): String + setCargo(cargo: String)

# Vererbung in Java

- Durch das Schlüsselwort `extends` kann die direkte Superklasse bei der Klassendeklaration angegeben werden.
  - Eine Klasse kann nur eine andere Klasse mit `extends` erweitern.
- Eine Klasse B kann die Subklasse einer anderen Klasse A sein.
- Ein Objekt der Subklasse ist auch ein Objekt der Superklasse.
- Eine Subklasse erbt alle Member der direkten Superklasse.
  - Der Zugriffsschutz wird berücksichtigt.
    - Private Member werden nicht vererbt.
    - Default Member werden nur im selben Paket vererbt.
  - Überschriebene Methoden werden nicht vererbt.
- In der Klassendeklaration können Felder, Methoden, Klassen und Interfaces als Member deklariert werden.
- Konstruktoren, Exemplarinitialisierer und statische Initialisierer sind **keine** Member.
- Eine Subklasse kann weitere Member deklarieren bzw. implementieren.

# Konstruktoren bei der Vererbung (1)

- Konstruktoren werden nicht vererbt.
- In jedem Konstruktor einer Subklasse muss direkt oder indirekt (über Konstruktorenverkettung) ein Konstruktor der Superklasse aufgerufen werden.
- Wird nichts angegeben, dann ruft ein Konstruktor implizit den parameterlosen Konstruktor der Superklasse auf.
- Wenn kein parameterloser Konstruktor existiert?
  - Entweder in der Superklasse implementieren.
  - Einen anderen Konstruktor der Subklasse oder Superklasse explizit aufrufen.

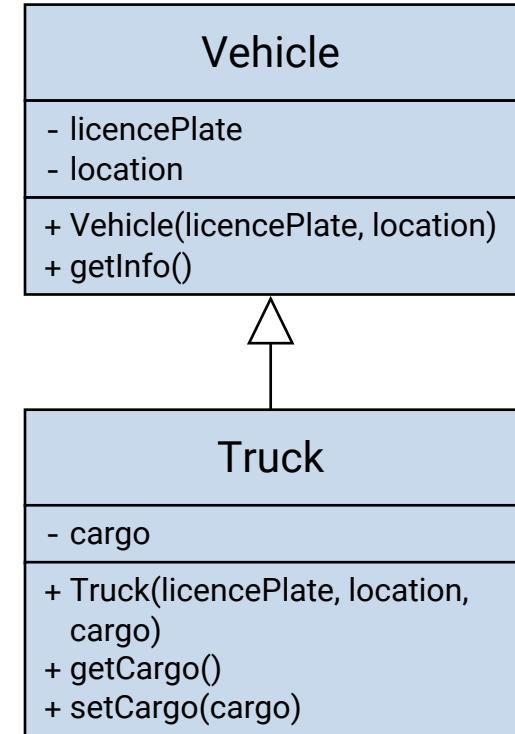
# Konstruktoren bei der Vererbung (2)

- Der parameterlose Konstruktor der Superklasse kann auch explizit mit `super();` aufgerufen werden (redundant!).
- Wie bei der Konstruktorenverkettung mit `this()` kann mit `super()` zu allen möglichen Konstruktoren der Superklasse eine Verbindung hergestellt werden.
- Folgende Einschränkungen existieren:
  - Der super-Aufruf darf nur einmal vorkommen.
  - Der super-Aufruf muss als **erste** Anweisung auftreten.
  - `this()`-Aufrufe und `super()`-Aufrufe können nicht gleichzeitig verwendet werden (immer erste Anweisung!).

# Fahrzeugverwaltung: Lösung mit Vererbung

```
public class Truck extends Vehicle {  
    ...  
}
```

- Vehicle
  - 💡 [src/at/ac/uibk/pm/inheritance/vehicles/basicinheritance/Vehicle.java](#)
- Truck
  - 💡 [src/at/ac/uibk/pm/inheritance/vehicles/basicinheritance/Truck.java](#)
- VehicleApplication (Main)
  - 💡 [src/at/ac/uibk/pm/inheritance/vehicles/basicinheritance/VehicleApplication.java](#)



- Anmerkung: wir werden diese erste Version mit Vererbung schrittweise optimieren. Die verschiedenen „Versionen“ liegen der Übersicht halber in eigenen Packages.

# Zugriffsschutz mit protected

- Im Beispiel sind `licensePlate` und `location` mit `private` gekennzeichnet.
  - In `Truck` kann nicht direkt darauf zugegriffen werden.
- Mit `protected` markierte Objektvariablen und Methoden stehen allen Subklassen zur Verfügung.
  - Sind in den Subklassen sowie in Klassen des gleichen Pakets sichtbar.
  - Sichtbarkeit erstreckt sich auch über mehrere Stufen einer Vererbungshierarchie.
  - Wie bei `private` gibt es eine Unterscheidung zwischen klassenbasiert und objektbasiert Definition (`klassenbasiert` in Java).

# Optimierung 1 Fahrzeugverwaltung

- Das Beispiel kann noch weiter optimiert werden.
- Problem: Die `getInfo()`-Methode berücksichtigt noch nicht die Besonderheiten (`cargo`) des Trucks.
  - Überschreiben der `getInfo()`-Methode
- Für Zugriff auf `licensePlate` und `location` der Superklasse müssen diese `protected` sein.

```
public class Vehicle {  
    protected String licensePlate;  
    protected String location;  
  
    ...  
}
```

```
public class Truck extends Vehicle {  
    @Override  
    public String getInfo() {  
        return String.format("%s at %s (cargo: %s)", licensePlate,  
                            location, cargo);  
    }  
    ...  
}
```



# Optimierung 2 Fahrzeugverwaltung

- Problem: Das `protected`-Setzen der Felder der Superklasse widerspricht dem Kapselungsprinzip.  
→ Verwenden der Getter der Klasse Vehicle

```
public class Vehicle {  
    private String licensePlate;  
    private String location;  
  
    ...  
}
```

```
public class Truck extends Vehicle {  
    @Override  
    public String getInfo() {  
        return String.format("%s at %s (cargo: %s)", getLicensePlate(),  
                            getLocation(), cargo);  
    }  
    ...  
}
```



# Optimierung 3 Fahrzeugverwaltung

- Problem: duplizierter Code durch die zwei getrennten getInfo()-Implementierungen.  
→ Verwenden der getInfo()-Methode der Klasse Vehicle

```
public class Truck extends Vehicle {  
    @Override  
    public String getInfo() {  
        return String.format("%s (cargo: %s)", super.getInfo(), cargo);  
    }  
    ...  
}
```



# Zugriff auf die Superklasse

- `super` kann auch in Methoden eingesetzt werden.
- `super.m()` ruft die entsprechende Methode `m()` in der **direkten** Superklasse auf.
- Eine weitere Verkettung ist nicht möglich!
  - d.h. `super.super.m()` ist **nicht** möglich.
- Kann auch für andere Member eingesetzt werden.
- Der Zugriffsschutz der Member wird berücksichtigt.

# Typkonvertierung bei Vererbung

- Es ist erlaubt, dass einer Variable  $o_A$  vom Typ A ein Objekt  $o_B$  einer beliebigen Subklasse B zugewiesen werden darf.
  - $o_A = o_B$  ist immer erlaubt (Ersetzbarkeitsprinzip).
  - Wird als Up-Cast bezeichnet.
- $o_B = (B) o_A$  ist nur zulässig, wenn  $o_A$  auf ein B-Objekt zeigt (wenn der Typ von  $o_A$  also B ist).
  - Expliziter Cast notwendig.
  - Wird als Down-Cast bezeichnet.
  - **Down-Cast ist problematisch!**
    - Kann meist mit Hilfe der dynamischen Polymorphie und dem Prinzip der Ersetzbarkeit umgangen werden.

```
// up-cast
Vehicle vehicle = new Truck("IL 473 NJ", "Technik Campus", "Water");
// valid down-cast
Truck truck = (Truck) vehicle;
```

# instanceof - Operator

- Objektvariablen sind **polymorph**.
  - Sie haben einen statischen Typ (Deklaration, z.B. Complex).
  - Sie haben eine aktuelle Klassenzugehörigkeit (dynamischer Typ, z.B. Polar).
- Mit dem instanceof – Operator kann ermittelt werden, ob eine Variable einen bestimmten dynamischen Typ aufweist.
  - Vererbung muss aber berücksichtigt werden.
  - Erbt eine Klasse B von einer Klasse A, dann ist ein entsprechendes Exemplar der Klasse B auch Exemplar der Klasse A, die Umkehrung gilt aber nicht!
  - instanceof sollte nur in Ausnahmefällen eingesetzt werden.
  - instanceof mit null ergibt immer false
  - Beim instanceof kann ein Pattern-Matching durchgeführt werden, dabei wird nach dem Typ eine Variable angegeben. Falls instanceof den Wert true ergibt, wird die Variable initialisiert.

```
public static void detectType(Vehicle vehicle) {  
    String type = vehicle instanceof Truck ? "Truck" : "Vehicle";  
    System.out.println(type + ": " + vehicle.getInfo());  
}
```

# Dynamische Bindung

- Beim Übersetzen von Methodenaufrufen prüft der Compiler den statischen Typ des Zielobjekts.
- Zur Laufzeit wird der Methodenaufruf dynamisch gebunden.
  - Der dynamische Typ wird herangezogen um die tatsächliche Methode zu bestimmen.
  - Wenn die Methode nicht überschrieben wurde, wird die entsprechende Methode aus der Superklasse aufgerufen.
- Ausnahme: `private` und `final` Methoden werden nicht dynamisch gebunden (sie können nicht überschrieben werden)

# Beispiel 1: Dynamische Bindung

```
public class A {  
    private String name = "A";  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public class B extends A {  
    private String name = "B";  
}
```

```
...  
A object1 = new A();  
B object2 = new B();  
A object3 = new B();  
System.out.println("object1 name: " + object1.getName());  
System.out.println("object2 name: " + object2.getName());  
System.out.println("object3 name: " + object3.getName());  
...
```

Ausgabe:

```
object1 name: A  
object2 name: A  
object3 name: A
```

(für bessere Lesbarkeit wurden abstrahierte Klassennamen gewählt)

# Beispiel 2: Dynamische Bindung

```
public class A {  
    private String name = "A";  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public class B extends A {  
    private String name = "B";  
  
    @Override  
    public String getName() {  
        return name;  
    }  
}
```

```
...  
A object1 = new A();  
B object2 = new B();  
A object3 = new B();  
System.out.println("object1 name: " + object1.getName());  
System.out.println("object2 name: " + object2.getName());  
System.out.println("object3 name: " + object3.getName());  
...
```

Ausgabe:  
object1 name: A  
object2 name: B  
object3 name: B

# Statische Bindung

- In bestimmten Situationen werden Methoden statisch gebunden:
  - **static**-Methoden
    - Gehören zu einer Klasse und nicht zu einem Objekt.
  - **private**-Methoden
    - Sind in der Subklasse nicht sichtbar.
    - Eine neue Definition einer privaten Methode ist kein Überschreiben!
  - **final**-Methoden
    - Kann in einer Subklasse nicht überschrieben werden.
- Binden von Konstruktoren
  - Konstruktoren werden statisch gebunden.
  - Objekt muss erst vom Konstruktor erzeugt werden.
- Binden von Datenelementen
  - Datenelemente werden statisch gebunden.
  - Beim Zugriff wird der statische Typ herangezogen.
  - Datenelemente werden aber geerbt.

# Beispiel: Statische Bindung Datenelemente

```
public class A {  
    public String name = "A";  
}
```

```
public class B extends A {  
    public String name = "B";  
}
```

```
...  
A object1 = new A();  
B object2 = new B();  
A object3 = new B();  
System.out.println("object1 name: " + object1.name);  
System.out.println("object2 name: " + object2.name);  
System.out.println("object3 name: " + object3.name);  
...
```

Ausgabe:

```
object1 name: A  
object2 name: B  
object3 name: A
```

# final bei Methoden und Klassen

- Wird eine Methode zusätzlich mit dem Schlüsselwort `final` versehen,
  - dann kann die Methode in einer Subklasse nicht mehr überschrieben werden und
  - dynamisches Binden wird unterbunden.
- Auch Klassen können mit `final` versehen werden.
  - Es kann keine Subklasse gebildet werden.
  - Alle Methoden sind automatisch `final`.
  - Beispiele (Performance als Grund für `final`)
    - `String`
    - `StringBuffer`
    - `StringBuilder`
    - `Integer`
    - ...

# Kovarianz, Kontravarianz, Invarianz

- Was darf beim Überschreiben verändert werden?
- Drei Varianten möglich:
  - Kovarianz (entlang der Vererbungsrichtung)
    - Redefinition von Parameter- und Rückgabetypen mit *Subtypen*
  - Kontravarianz (entgegen der Vererbungsrichtung)
    - Redefinition von Parameter- und Rückgabetypen mit *Supertypen*
  - Invarianz
    - Typen bleiben gleich
- Beim Überschreiben in Java
  - Kovarianz beim Rückgabetyp
  - Sonst Invarianz

# Kovarianter Rückgabetyp

- Kovariante Rückgabetypen erlauben jeden kompatiblen Rückgabetyp bei der Redefinition geerbter Methoden und bei der Implementierung von Interfacemethoden.
- „Verschärfung“ des Rückgabetyps
- Beispiel:

```
public class A {  
    public A get() {...}  
}
```

```
public class B extends A {  
    @Override  
    public B get() {...}  
}
```

```
A a = new B();  
B b = new B();  
  
A object1a = a.get(); // OK  
// B object1b = a.get(); // not OK  
A object2a = b.get(); // OK - covariance & Liskov substitution principle  
B object2b = b.get(); // OK - covariance
```

(für bessere Lesbarkeit wurden abstrahierte Klassennamen gewählt)

# Vererbung der Implementierung

- Vorteil der Vererbung der Implementierung
  - Methoden müssen nicht neu implementiert werden.
  - Redundanzen im Quellcode werden vermieden.
  - DRY!
- Aber
  - Vererbung legt eine starre Struktur fest.
  - Erweiterungen sind mit Aufwand verbunden.

# Probleme

- Klasse B kann Funktionalität der Klasse A nutzen durch:
  - Vererbung (B erbt von A)
  - Beziehung (Assoziation zwischen B und A)
- Fälle
  - Vererbung erscheint einfacher.
  - Aber nicht immer sinnvoll.
  - **Prinzip der Ersetzbarkeit sollte immer gelten!**

# Verletzung des Prinzips der Ersetzbarkeit (1)

```
public class Rectangle {  
    private int width;  
    private int length;  
  
    public void setWidth(int width) { ... }  
    public void setLength(int length) { ... }  
    ...  
}
```

```
public class Square extends Rectangle {  
}
```

Probleme:

- Für ein Square wird Speicherplatz verschwendet (eine Seite reicht).
- Invariante von Square (`width == length`) ist nicht garantiert.
- `setWidth` und `setLength` sollten nicht zur Verfügung gestellt werden.

# Verletzung des Prinzips der Ersetzbarkeit (2)

```
public class Rectangle {  
    private int width;  
    private int length;  
  
    public void setWidth(int width) { ... }  
    public void setLength(int length) { ... }  
    ...  
}
```

```
public class Square extends Rectangle {  
  
    @Override  
    public void setWidth(int width) {  
        super.setWidth(width);  
        super.setLength(width);  
    }  
  
    @Override  
    public void setLength(int length) {  
        super.setWidth(length);  
        super.setLength(length);  
    }  
}
```

Lösungsversuch:

- `setWidth` und `setLength` überschreiben.

Probleme:

- `Square` kann nicht mehr für `Rectangle` eingesetzt werden.
  - Nachbedingung `setLength`:
    - `width` unverändert
    - `length` wird entsprechend gesetzt
  - Nachbedingung `setWidth`
    - `width` wird entsprechend gesetzt
    - `length` unverändert
- Prinzip der Ersetzbarkeit wird verletzt!

# Problem der instabilen Basisklasse

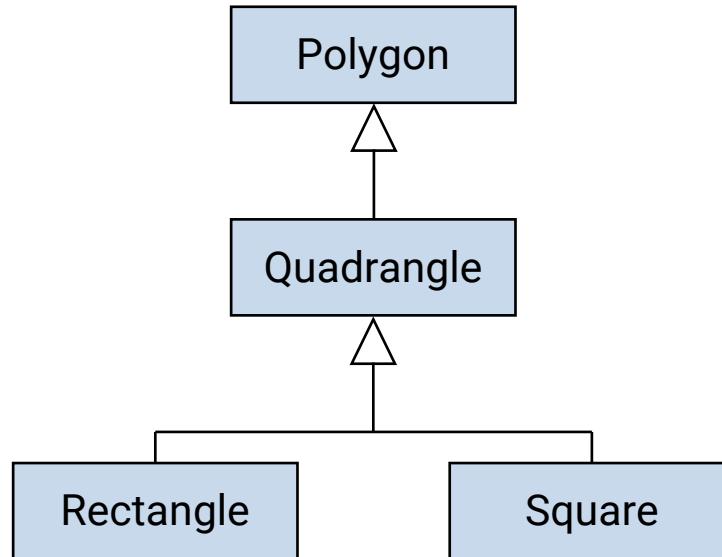
- Gilt das Prinzip der Ersetzbarkeit? Gilt dies auch in Zukunft?
- Problem der instabilen Basisklasse (Fragile Base Class Problem).
  - Anpassungen an der Basisklasse können zu unerwartetem Verhalten von abgeleiteten Klassen führen.  
→ Wartung von Systemen, die nur Vererbung der Implementierung benutzen, ist schwierig.
- Sind spätere Änderungen an der Basisklasse sehr wahrscheinlich:
  - Vererbung der *Spezifikation*
  - Vererbung der Implementierung vermeiden
  - Redundanten Code vermeiden durch
    - Aggregation
    - Komposition
    - Delegation

# Delegation

- Ein Objekt setzt eine Operation so um, dass der Aufruf der Operation an ein anderes Objekt delegiert (weitergereicht) wird.
- Verantwortung, eine Implementierung für eine bestimmte Schnittstelle bereitzustellen, wird an ein Exemplar einer anderen Klasse delegiert.
- Vorteil
  - Kann mit der Vererbung der Spezifikation gemeinsam benutzt werden (Quellcode einsparen).
  - Auch ohne Mehrfachvererbung kann die Funktionalität von mehreren Klassen benutzt werden.
  - Dynamisch (Delegat kann zur Laufzeit geändert werden).

# Abstraktionsebenen

- Vererbung einsetzen, wenn Superklasse und Subklasse konzeptionell auf unterschiedlichen Abstraktionsebenen stehen.
  - Klassename ist umgangssprachlich ein Oberbegriff des anderen Klassennamens.
- Beispiel



# „is-a“ vs. „has-a“

- „B ist ein A“ → B wird von A abgeleitet.
- „B enthält ein A“ → B definiert eine Objektvariable vom Typ A.
- Beispiel
  - 2 Klassen Dreieck und Polygon (is-a)
    - Dreieck ist ein Polygon – Richtig
    - Dreieck enthält ein Polygon – Falsch
    - Dreieck wird von Polygon abgeleitet.
  - 2 Klassen Dreieck und Punkt (has-a)
    - Dreieck ist ein Punkt – Falsch
    - Dreieck enthält einen Punkt – Richtig
    - Klasse Dreieck verwendet daher die Klasse Punkt (Objektvariable vom Typ Punkt).

Grundregel: Composition over inheritance (Ausnahme: is-a)



# Abstrakte Klassen in Java

# Abstrakte Klassen

- Konkrete Superklassen und Interfaces bilden zwei Extreme.
- Abstrakte Klassen bilden einen Mittelweg.
  - Können Felder enthalten.
  - Können vollständige Methoden enthalten.
  - Können Schnittstellen (abstrakte Methoden) enthalten.
- Eine abstrakte Klasse wird mit dem Schlüsselwort `abstract` markiert.
  - Zusätzlich werden die abstrakten Methoden mit `abstract` gekennzeichnet.
  - Es kann auch keine abstrakten Methoden in einer abstrakten Klasse geben.
- Von einer abstrakten Klasse können **keine** Objekte angelegt werden.

# Vererbung bei abstrakten Klassen

- Eine Subklasse muss alle abstrakten Methoden implementieren, damit von dieser Subklasse Objekte erzeugt werden können.
- Implementiert eine Subklasse nur einen Teil (oder keine) der abstrakten Methoden, dann ist sie auch eine abstrakte Klasse.
- In einer Subklasse können neue Methoden definiert und Methoden überschrieben werden.
- Auch `this` und `super` kann verwendet werden.

# Abstrakte Klasse vs. Interface (1)

	Abstrakte Klasse	Interface
<b>Variablen</b>	Objekt- und Klassenvariablen	Konstanten (public static final)
<b>Zugriffsmodifikatoren für abstrakte Methoden</b>	public, protected, default	public (implizit)
<b>Konstruktoren</b>	Definition möglich	Definition unmöglich
<b>Exemplare erzeugen</b>	Unmöglich	Unmöglich
<b>Vererbung</b>	Einfachvererbung	Mehrfachvererbung

# Abstrakte Klasse vs. Interface (2)

- Abstrakte Klassen können verwendet werden:
  - Wenn Code zwischen eng verwandten Klassen geteilt werden soll.
  - Wenn abgeleitete Klassen viele gemeinsame Methoden oder Felder haben.
  - Wenn konkrete oder abstrakte Methoden einen anderen Zugriffsschutz als `public` benötigen (beispielsweise `protected`).
  - Wenn Variablen deklariert werden sollen, die nicht `static` und `final` sind.
  - Wenn der Zustand des Objekts dargestellt werden soll.
  - Wenn der Zustand durch Methoden abgefragt und verändert werden soll.
- Interfaces können verwendet werden:
  - Wenn verschiedene nicht verwandte Klassen die Spezifikation implementieren möchten.
  - Wenn das Verhalten eines Datentyps definiert werden soll, wer genau das Verhalten implementiert aber unwichtig ist.
  - Wenn Mehrfachvererbung der Spezifikation benötigt wird.

# Quellen

- Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (Java SE 17 Edition)**, Oracle, 2021

# Wrapper-Klassen

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck

# Motivation

- Kapselt die primitive Variable in einer objektorientierten Hülle.
  - Methoden für den Zugriff und die Umwandlung.
  - Java Datencontainer (Listen, Mengen, etc.) nehmen nur Referenzen auf.
  - Primitive Datentypen können nicht bei Generics verwendet werden (später mehr dazu).
- Zu jedem primitiven Datentyp in Java gibt es eine sogenannte **Wrapper-Klasse**.

Klasse	Primitiver Datentyp
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean
Character	char

# Handling Wrapper-Klassen (1)

- Erzeugung
  - Konstruktor mit primitivem Typ (deprecated)  
`Integer i = new Integer(10);`
  - Konstruktor mit String (deprecated)  
`Integer i = new Integer("300");`
  - Statische Methode (primitiver Typ → Cache für mindestens -128 bis 127)  
`Integer i = Integer.valueOf(10);`
  - Autoboxing (verhält sich wie valueOf())  
`Integer i = 3;`

# Handling Wrapper-Klassen (2)

- Rückgabemethoden
  - Rückgabe als primitiver Typ

```
Integer i = 3;  
int j = i.intValue();
```
  - Rückgabe als String
- Alle Wrapper-Klassen sind unveränderlich (immutable).
  - Die Wrapper-Klassen sind nur als Ummantelung und nicht als vollständiger Datentyp gedacht.
  - "Werte-Objekte"
  - Änderung des Wertes führt zu neuem bzw. anderem Objekt.
- Alle numerischen Wrapper-Klassen erweitern die abstrakte Klasse Number.

# Hilfsmethoden

- Wrapper-Klassen bieten nützliche (statische) Hilfsmethoden an.
- Beispiele:
  - Parsen von Strings (z.B. Integer)  
`public static int parseInt(String s)`
    - Versucht einen String in eine Ganzzahl umzuwandeln.
    - Wenn der übergebene String nicht einer Ganzzahl entspricht, dann wird das Programm abgebrochen (Ausnahme/Exception).
  - Berechnen des Hash-Werts (z.B. Integer)  
`public static int hashCode(int value)`
  - Vergleiche mit compare (z.B. Double)  
`public static int compare(double d1, double d2)`

```
double d1 = -0.0;
double d2 = 0.0;
System.out.println(d1 == d2);
System.out.println(Double.compare(d1, d2) == 0);
```

Ausgabe:

true

false

# Konstanten

- Byte, Short, Integer, Long und Character:
  - MIN\_VALUE
  - MAX\_VALUE
  - ...
- Float und Double:
  - MIN\_VALUE
  - MAX\_VALUE
  - NEGATIVE\_INFINITY
  - POSITIVE\_INFINITY
  - NaN
  - ...

# Autoboxing/Autounboxing (1)

- Automatische Konvertierung zwischen primitiven Datentypen und Wrapper-Klassen.
- Compiler fügt notwendigen Code automatisch ein.
- Zwei Richtungen
  - Umwandlung primitiver Wert → Wrapper-Objekt (Boxing)
  - Umwandlung Wrapper-Objekt → primitiver Wert (Unboxing)

```
Integer i = Integer.valueOf(5); // Boxing  
int j = i.intValue();          // Unboxing
```

```
Integer i = 5; // Autoboxing  
int j = i;     // Autounboxing
```

# Autoboxing/Autounboxing (2)

- Primitive Typen können damit wie Referenztypen verwendet werden.
- Durch das Autounboxing können arithmetische und bitweise Operatoren auf Objekten von Wrapper-Klassen angewendet werden.
- Funktioniert nicht bei Arrays, nur bei primitiven Typen.
- Bei Autoboxing kann ein Wert eines primitiven Datentyps nur in die entsprechende Wrapper-Klasse umgewandelt werden.

```
Integer[] array1 = {1, 2, 3};  
++array1[0];  
int[] array2 = array1; // compiler error due to type missmatch  
  
float fa = 123;  
Float fb = 123; // compiler error due to type missmatch  
Float fc = 123f;
```

# Autoboxing/Autounboxing (3)

- Autoboxing verschleiert Objektgenerierung.
  - z. B. wenn zwei Wrapper-Typen verglichen werden, wird ein Referenzvergleich durchgeführt (kein Autounboxing).
- Ganzzahl-Caching bei Autoboxing bzw. `valueOf`-Methode:
  - Alle Objekte im Bereich -128 bis +127 werden garantiert gecached.
  - In diesem Bereich erhält man immer die gleiche Referenz!

```
Integer i1 = 2;
Integer i2 = 2;
System.out.println(i1 == i2); // true

Integer j1 = 128;
Integer j2 = 128;
System.out.println(j1 == j2); // false (if cache size was not increased)

Integer k1 = new Integer(10);      // new object
Integer k2 = Integer.valueOf(10); // Boxing => Cache
Integer k3 = 10;                 // Autoboxing => Cache
System.out.println(k1 == k2); // false
System.out.println(k1 == k3); // false
System.out.println(k2 == k3); // true
```

# Autoboxing/Autounboxing (4)

- Wird ein primitiver Typ mit einem Wrapper-Typ verglichen, findet immer Autounboxing statt.
- Bei Operationen, welche nicht für Referenztypen definiert sind, findet Autounboxing statt.

```
Integer k1 = new Integer(10);
Integer k2 = 10;
System.out.println(k1 == 10); // true
System.out.println(k2 == 10); // true

System.out.println(k1 >= k2); // true
System.out.println(k1 <= k2); // true
System.out.println(k1 == k2); // false
```

# Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021

# Die Klasse Object

Programmiermethodik

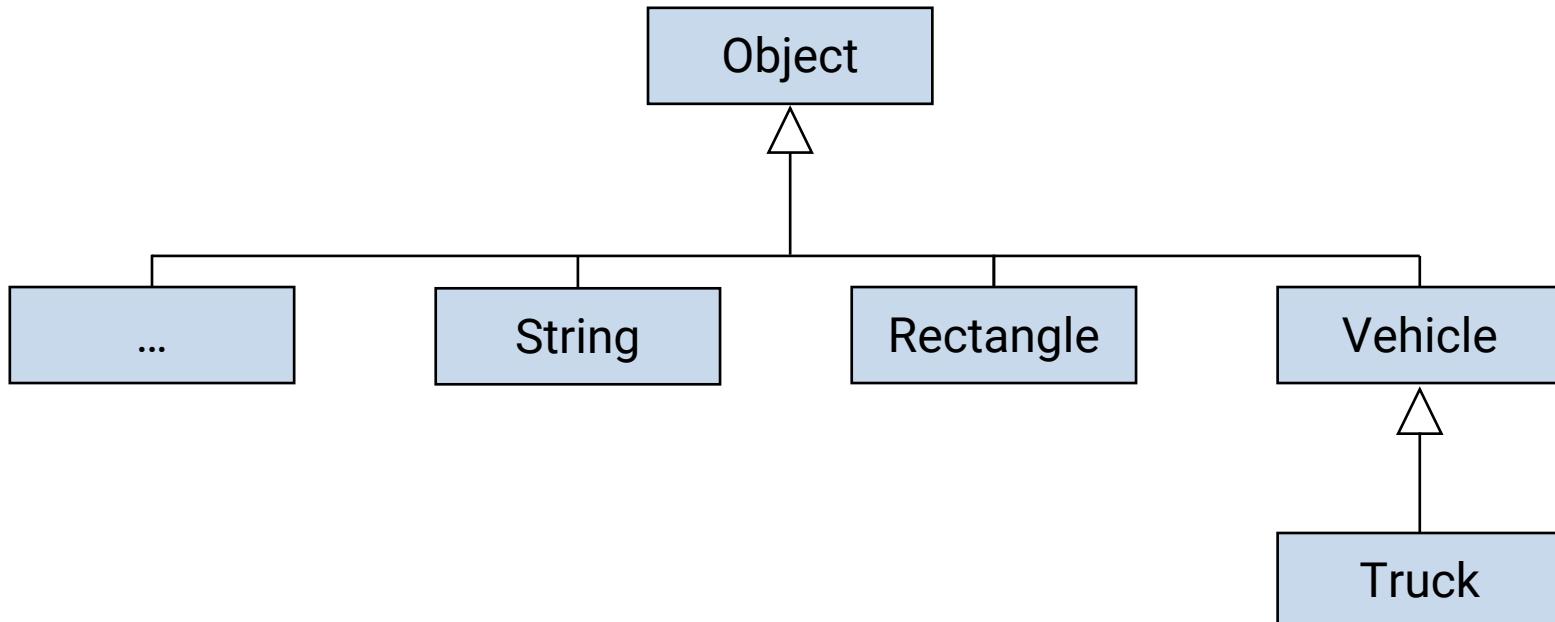
Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck

# Die Wurzelklasse Object

- Jede Klasse, die von keiner Klasse erbt, erbt automatisch von der Wurzelklasse Object.
  - Folgende Definitionen sind äquivalent:

```
public class className {...}  
public class className extends Object {...}
```
- Damit werden alle Klassen direkt oder indirekt von Object abgeleitet.



# Vordefinierte Methoden

- Object bietet einige Methoden an, die an jede Java-Klasse vererbt werden.
- Methoden können überschrieben werden.
- Auszug:

**public** String **toString()**

- Liefert eine lesbare Repräsentation des Objekts.

**public boolean** **equals(Object x)**

- Prüft ob das Zielobjekt und x gleich sind.

**public int** **hashCode()**

- Liefert eine Kennnummer (Hashcode) des Objekts.

**protected** Object **clone()**

- Liefert eine Kopie des Objekts.

**protected void** **finalize()**

- Wird vom Garbage Collector aufgerufen.
- Problematisch

- Vollständige Übersicht in der [Java 17 API Dokumentation](#)

# toString

- Die Implementierung aus der Klasse Object gibt den Klassennamen gefolgt von einem @-Zeichen und der hexadezimale Repräsentation des Hash-Codes zurück.
  - Beispiel: at.ac.uibk.pm.inheritance.rectangle.Rectangle@630
- Sollte in der Regel in allen Subklassen überschrieben werden, sofern sie nicht in einer Superklasse entsprechend überschrieben wurde.
  - Bei Hilfsklassen, von welchen kein Exemplar erzeugt werden kann, und bei Enums ist ein Überschreiben typischerweise nicht erforderlich.
- Wenn möglich, sollten alle relevanten Informationen in der String-Repräsentation abgebildet werden.
- Eine gute `toString`-Implementierung hilft beim Verwenden der Klasse.
- Die `toString`-Methode wird automatisch aufgerufen, wenn z.B.
  - ein Objekt der Methode `println`, `print` oder `printf` übergeben oder
  - eine Stringkonkatenation durchgeführt wird.

# Beispiel `toString`

```
public class Rectangle {  
    private int width;  
    private int length;  
    ...  
    @Override  
    public String toString() {  
        return "Rectangle{" + "width=" + width + ", length=" + length + '}';  
    }  
    ...  
}
```

```
public class RectangleApplication {  
    public static void main(String[] args) {  
        Rectangle rectangle1 = new Rectangle(20, 3);  
        System.out.println(rectangle1);  
    }  
}
```

```
Rectangle{width=20, length=3}
```



# equals (1)

- Implementierung aus der Klasse Object vergleicht nur Referenzen.

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

- Eine korrekte Implementierung erfüllt:

- Reflexivität

x.equals(x) → true

- Symmetrie

x.equals(y) == y.equals(x)

- Transitivität

x.equals(y) → true, y.equals(z) → true dann gilt auch x.equals(z) → true

- Konsistenz

Zwei Objekte müssen bei wiederholten Aufrufen immer das gleiche Ergebnis liefern, so lange sie sich nicht verändert haben.

- Für alle Objekte x, die nicht gleich null sind, gilt:

x.equals(null) → false

# equals (2)

- Es kann zu Problemen beim Verwenden dieser Methode führen, wenn dieser Vertrag nicht eingehalten wird.
- Fundamentales Problem:
  - Eine Klasse, von welcher Exemplare erzeugt werden können, wird erweitert.
  - Die abgeleitete Klasse erweitert die Klasse mit zusätzlichen Objektvariablen.
  - Der equals-Vertrag kann nicht eingehalten werden, ohne das Prinzip der Ersetzbarkeit zu verletzen.
  - Durch Komposition kann dieses Problem umgangen werden.
- Empfehlung für eigene equals-Methode:
  1. Mittels Referenzvergleich die Referenzen vergleichen. Sofern diese gleich sind, true zurückgeben.
  2. Mit dem instanceof-Operator überprüfen, ob der Parameter den korrekten Typ hat. Falls nicht, false zurückgeben.
  3. Für jedes signifikante Feld der Klasse, überprüfen, ob das Feld mit dem entsprechenden Feld der Pattern-Variable übereinstimmt. Falls alle übereinstimmen, wird true zurückgegeben, sonst false.

# Beispiel equals

```
public class Rectangle {  
    private int width;  
    private int length;  
  
    ...  
    @Override  
    public boolean equals(Object other) {  
        if (this == other) {  
            return true;  
        }  
        if (!(other instanceof Rectangle rectangle)) {  
            return false;  
        }  
        return width == rectangle.width && length == rectangle.length;  
    }  
    ...  
}
```



# hashCode

- Für jedes Objekt sollte eine eindeutige Kennnummer (nicht immer möglich) produziert werden.
- Wird vor allem im Collection-Framework (z.B. HashMap; wird noch besprochen) benutzt.
- Die Methoden `equals` und `hashCode` hängen zusammen und sollten immer gemeinsam überschrieben werden.
- Anforderungen:
  - Wenn zwei Objekte gemäß `equals` gleich sind, müssen sie auch den gleichen Hash produzieren.
  - Die Umkehrung gilt nicht, d.h. zwei Objekte können den gleichen Hash haben, aber verschieden sein.
  - Der Wert von `hashCode` muss immer gleich bleiben, solange sich die für `equals` signifikanten Objektvariablen des Objekts nicht ändern.

# Empfehlung für eigene hashCode-Methode

1. int-Variable result mit dem Hash c der ersten signifikanten Objektvariable initialisieren (siehe 2.a.).
2. Für jede weitere signifikante Objektvariable a:
  - a. Einen int-Hash c berechnen:
    - i. Falls a einen primitiven Typ hat und der Typ type ist, Type.hashCode(a) verwenden.
    - ii. Falls a eine Objektreferenz und null ist, 0 verwenden.
    - iii. Falls a eine Objektreferenz und nicht null ist, rekursiv hashCode aufrufen.
    - iv. Falls a ein Array ist, für alle signifikanten Elemente den Hash, wie in 2.a. beschrieben, ermitteln. Falls alle Felder signifikant sind, Arrays.hashCode verwenden.
  - b. Den Hash c zu result hinzufügen:  
$$\text{result} = 31 * \text{result} + c$$
3. result zurückgeben

Falls Performance irrelevant ist, kann die statische Methode Objects.hash verwendet werden, um den Hash beliebig vieler Variablen zu berechnen.

# Beispiel hashCode

```
public class Rectangle {  
    private int width;  
    private int length;  
  
    ...  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = Integer.hashCode(width);  
        result = prime * result + Integer.hashCode(length);  
        return result;  
    }  
    ...  
}
```

```
public class Rectangle {  
    ...  
    @Override  
    public int hashCode() {  
        return java.util.Objects.hash(width, length);  
    }  
    ...  
}
```

# clone

- Ein Kopier-Konstruktor eignet sich zum Kopieren eines Objekts ohne Vererbung.
- Kopieren bei einem dynamischen Typ kann nur durch eine dynamisch gebundene Methode erfolgen – `clone`-Methode.
- `clone` hat den Zugriffsschutz `protected`.
  - Damit kann die Methode nicht von außen aufgerufen werden, wenn sie nicht überschrieben wird.
  - Beim Überschreiben muss der Zugriffsschutz gelockert werden (`public`).
- Voraussetzung für Anwendbarkeit: Klasse muss
  - das Interface `Cloneable` implementieren,
  - eine eigene öffentliche Methode `clone()` implementieren,
  - in `clone` eine Kopie des Superklassenobjekts mit `super.clone()` erzeugen,
  - in `clone` alle Datenelemente veränderlicher Klassen einzeln mit `clone`-Aufrufen auf diese Objekte kopieren.
- `clone` kann nur verwendet werden, wenn jede verwendete Klasse `clone` korrekt implementiert.

# finalize

- Ist seit Java 9 als `@Deprecated` markiert und sollte somit nicht verwendet werden
- Ist inhärent problematisch
  - Kann unter anderem zu Performanceeinbußen, Deadlocks und Ressourcen-Lecks führen.
  - Ausführungszeitpunkt ist nicht festgelegt (kann unbegrenzt aufgeschoben werden).
- Wenn eine Klasse Ressourcen die sie hält explizit wieder freigeben muss, sollte eventuell `AutoCloseable` implementiert werden.
- Durch die `Cleaner` und `PhantomReference` Klassen werden effizientere und flexiblere Wege Ressourcen wieder freizugeben bereitgestellt.
- Mehr zum Thema lernen wir am Ende des Semesters bei JVM.

# Quellen

- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018

# Exceptions

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Motivation

- In einem ablaufenden Programm können Fehler auftreten.
  - Logische Fehler im Programm
  - Fehlerhafte Bedienung
  - Probleme im Java-Laufzeitsystem
  - Technische Fehler (Speicherplatz, Klasse von Laufzeitumgebung nicht gefunden)
- Programme müssen auf Fehlersituationen vorbereitet werden und **kontrolliert** darauf reagieren.
- In C:
  - Kein einheitliches Vorgehen.
  - Fehler können über Rückgabewerte erkannt werden.
- In Java:
  - Ausnahmebehandlung (Exception-Handling)
  - Exceptions sind ein Sprachmittel zur kontrollierten Reaktion auf Laufzeitfehler!

# Exception-Handling

- Exceptions (Ausnahmen) sind ein Sprachmittel zur kontrollierten Reaktion auf Laufzeitfehler!
- Sofern eine Exception auftritt, wird durch das Exception-Handling der normale Programmfluss verlassen.
  - Die Kontrolle geht an den Mechanismus der Ausnahmebehandlung über.
  - Im Exception-Handler wird die Ausnahme behandelt.
  - Nach der Behandlung wird der Kontrollfluss wieder an das Programm zurückgegeben.
- Das Auslösen einer Exception wird als Werfen bezeichnet.
- Das Behandeln einer Exception wird als Fangen bezeichnet.

# Vorteile durch Exception-Handling

- Mechanismus zur strukturierten Behandlung von Ausnahmesituationen.
- Code für den regulären Programmablauf und die Fehlerbehandlung wird getrennt.
- Exceptions können entlang der Aufrufhierarchie propagiert werden. Dadurch kann die Ausnahme an der Stelle behandelt werden, die am Besten dafür geeignet ist.
  - Eine Exception wird an bestimmten Punkten im Programm geworfen.
  - An einer anderen Stelle im Programm steht Code zum Fangen potenzieller Ausnahmesituationen.
- Bestimmte Exceptions können nicht ignoriert werden und es müssen Maßnahmen zur Behandlung getroffen werden.

# Exception-Handling in Java

- Information über die Exception wird in ein spezielles Objekt verpackt, welches ein Exemplar der Klasse Throwable oder einer Subklasse von Throwable ist.
- Durch die throw-Anweisung können Exceptions explizit ausgelöst werden.
- Die Ausnahme kann an einer bestimmten Stelle (äußerer Block, aufrufende Methode etc.) in einer catch-Klausel abgefangen werden.
- Grundkonstrukt: try-Anweisung

# try-Anweisung

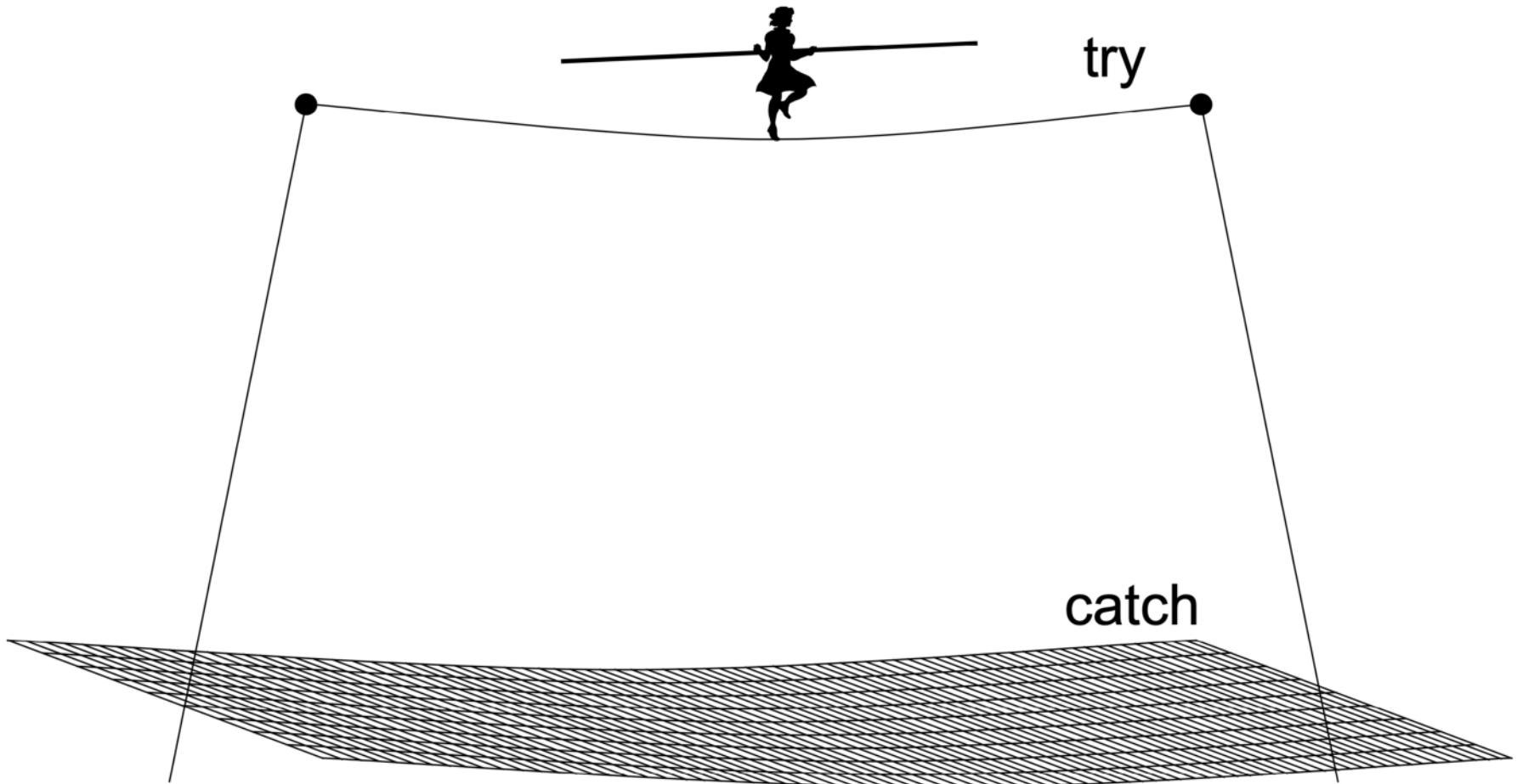


Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

# try-Anweisung (Grundform)

```
try {  
    ...  
    ...  
}  
} catch (ExceptionType1 e) {  
    ...  
    ...  
}  
} catch (ExceptionType2 e) {  
    ...  
    ...  
}
```

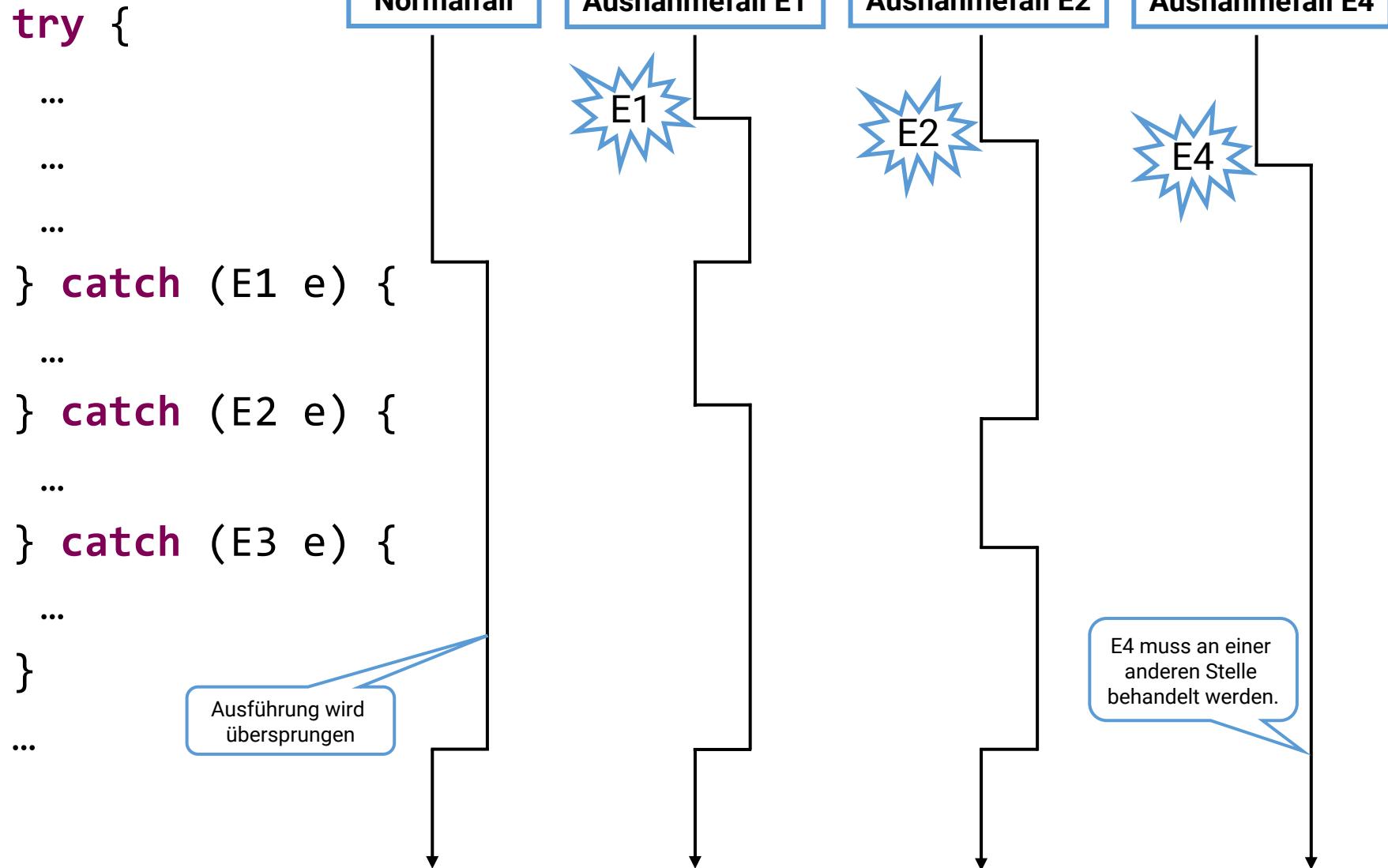
regulärer Code, in dem Fehler auftreten kann

Code für Fehlerbehandlung der Exception e vom Typ ExceptionType1

Code für Fehlerbehandlung der Exception e vom Typ ExceptionType2

- Nach dem try-Block folgen die catch-Klauseln.
- Es können beliebig viele catch-Klauseln verwendet werden.
- Eine catch-Klausel hat genau einen Parameter (Exception-Parameter).
- Der Typ des Exception-Parameters bestimmt, welche Exceptions durch diese catch-Klausel gefangen werden können.
- Mindestens eine catch-Klausel oder ein finally-Klausel muss vorhanden sein.

# Ablauf (einfach)



# Ablauf (1)

- Wird eine Ausnahme vom Typ E geworfen:
  - Dann wird eine entsprechende catch-Klausel für den Typ E gesucht.
  - Statt E kann auch eine Klausel mit einer Superklasse von E verwendet werden, da E dort substituiert werden kann!
- Reihenfolge der catch-Klauseln ist entscheidend.
  - Eine Ausnahme wird der Reihe nach mit den catch-Klauseln verglichen.
  - Ausgeführt wird die erste catch-Klausel, zu der die Ausnahme kompatibel ist.
  - Nachfolgende catch-Klauseln werden ignoriert.
  - Daher muss die Ordnung der catch-Klauseln beachtet werden!
    - Von speziell zu allgemein!
  - Wenn zuerst eine allgemeinere Ausnahme angegeben wird (und dann eine speziellere) kommt es zu einem Compiler-Fehler.

# Ablauf (2)

- Falls keine catch-Klausel gefunden wird, wird in den äußenen try-Anweisungen gesucht.
  - In verschachtelten try-Anweisungen
  - Entlang der Methodenaufrufkette (wird noch besprochen)
- Wird nie eine catch-Klausel gefunden (auch nicht in main), dann bricht das Programm ab.

# Termination Model / Resumption Model

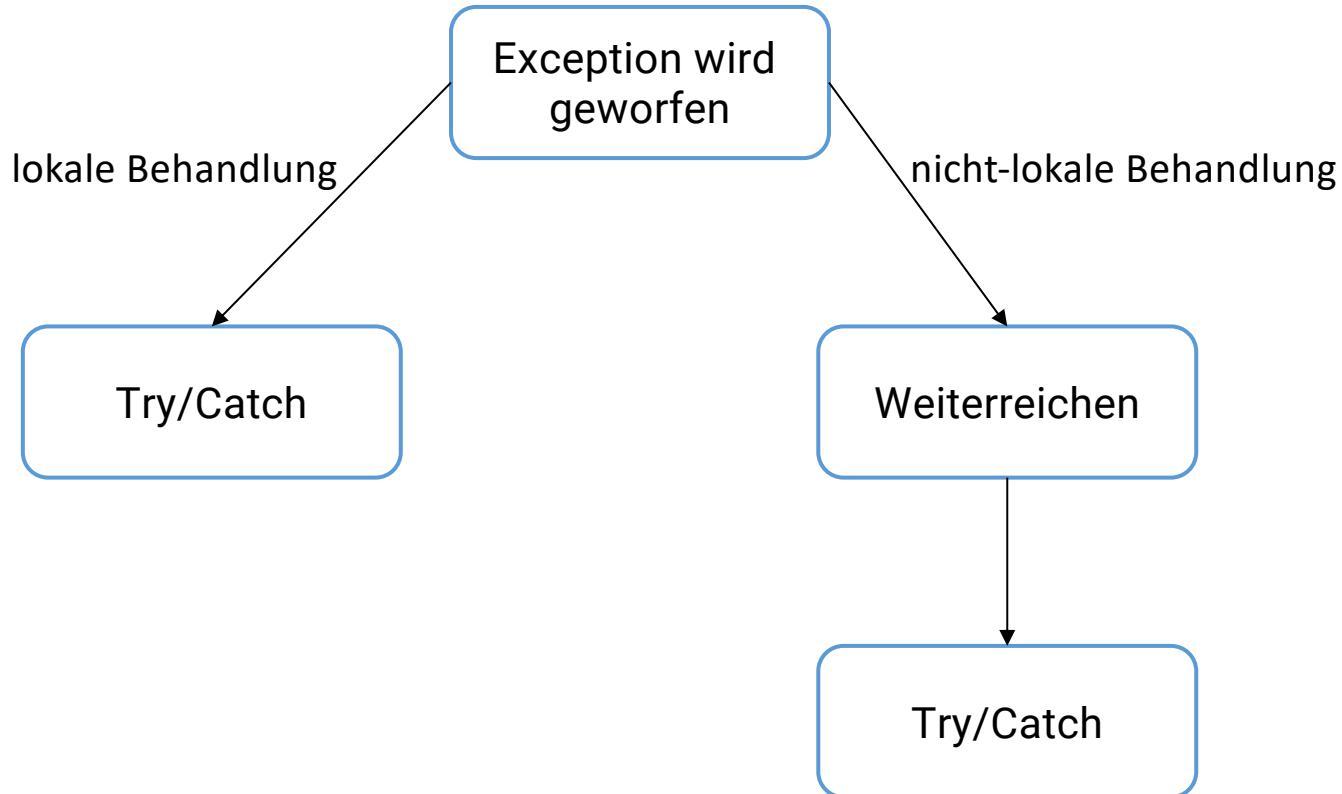
## Termination Model

- Der Kontrollfluss kehrt nicht mehr an die Stelle zurück, an der die Ausnahme aufgetreten ist (kehrt an Position nach try-Anweisung zurück).
- Die Ausnahmebehandlung in Java folgt dem **Termination Model**.

## Resumption Model

- Bei diesem Modell erfolgt eine Rückkehr an die Aufrufstelle, d.h. das Programm setzt an der Stelle der Ausnahme fort.
- Das **Resumption Model** wird nicht in Java verwendet.

# Behandlung von Exceptions



# Beispiel lokale Behandlung

```
public class ExceptionTest {  
    public static void main (String[] args) {  
        try {  
            int x = Integer.parseInt(args[0]);  
            int y = Integer.parseInt(args[1]);  
            System.out.println(x / y);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Please provide at least two command-line parameters!");  
            // additional error handling  
        } catch (NumberFormatException e) {  
            System.out.println("Please provide two integers!");  
            // additional error handling  
        } catch (ArithmaticException e) {  
            System.out.println("Division by zero!");  
            // additional error handling  
        }  
    }  
}
```

**java** ExceptionTest 4 2  
2

**java** ExceptionTest  
Please provide at least two command-line parameters!

Kann in parseInt() auftreten.

**java** ExceptionTest 1 z  
Please provide two integers!

# Hierarchie von Exceptions

- Ausnahmen sind in Java Objekte.
  - Enthalten Informationen über die aufgetretenen Fehler
  - Bieten Methoden an, um auf die Informationen zuzugreifen
- Alle Ausnahmen sind in einer Klassenhierarchie gruppiert.
  - Dient zur Differenzierung
  - Eine catch-Klausel, die eine Klasse E behandeln kann, kann auch alle Unterklassen von E behandeln.
- Oberste Klasse ist `java.lang.Throwable`
  - Enthält Methoden zur Fehleranalyse
    - `String getMessage()` (Text bei Ausnahme)
    - `String toString()` (Klassenname + `getMessage()`)
    - `void printStackTrace()`
  - Konstruktoren (leer, mit Fehlermeldung,...)
- Unterklassen
  - `java.lang.Error`
    - Irreparable Fehler
  - `java.lang.Exception`
    - Fehler, die sinnvoll behandelt werden können

# Arten von Ausnahmen

## Unchecked Exceptions

- Werden automatisch ausgelöst (z.B. illegale Instruktionen)
- Exemplare, die aus den Klassen Error, RuntimeException sowie davon abgeleiteten Klassen erzeugt wurden.
- Können abgefangen werden, müssen es aber nicht!

## Checked Exceptions

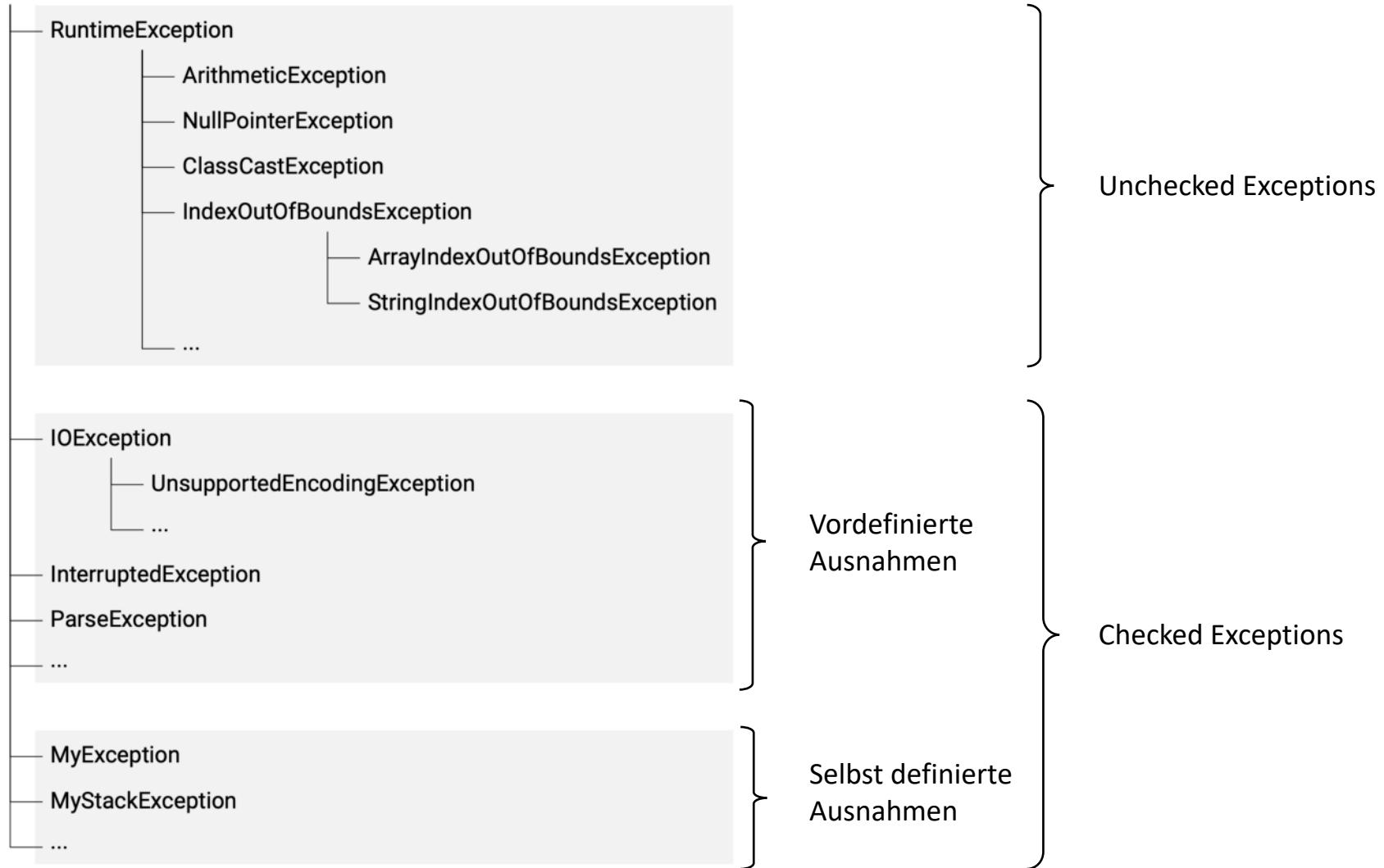
- Müssen explizit beim Programmieren mit der throw – Anweisung ausgelöst werden.
- Alle Exceptions bis auf Exemplare, die von Error, RuntimeException sowie davon abgeleiteten Klassen entstammen, sind Checked Exceptions.
- Der Compiler überprüft, ob die Exceptions behandelt werden.

# Error-Klasse

- Error sind Fehler, die mit der JVM in Verbindung stehen.
- Beispiel:
  - `OutOfMemoryError`: Zu wenig Speicher für die JVM bei der Objekterzeugung vorhanden
- Da die Fehler „abnormales“ Verhalten anzeigen, müssen sie auch nicht mit einer `catch`-Klausel aufgefangen werden.
  - Wie `RuntimeException` können sie aber abgefangen werden.
- Ein Auffangen macht wenig Sinn.
  - Wie sollte auf solche Fehler reagiert werden?

# Exception-Hierarchie

## Exception



# Eigene Exception erstellen

- Exceptions müssen direkt oder indirekt von der Klasse Throwable erben.
  - Eigene Exceptions sollten immer nur direkt oder indirekt von der Klasse Exception erben.
- Von der Klasse Exception ableiten:

```
public class MyException extends Exception {...}
```

- Wenn eine Nachricht mitgegeben werden soll:

```
public class MyException extends Exception {  
  
    public MyException() {  
    }  
  
    public MyException(String specialInfo) {  
        super(specialInfo);  
    }  
}
```

# Exceptions auslösen

- Auslösen („Werfen“) einer Ausnahme mit der throw-Anweisung:  
`throw new MyException();`  
`throw new MyException("Falsche Eingabe");`
- Die throw-Anweisung unterbricht den laufenden Code sofort!

- Siehe Beispiel:

 <src/at/ac/uibk/pm/exceptions/stack/StackException.java>

 <src/at/ac/uibk/pm/exceptions/stack/resourceloader/ResourceLoader.java>

# Weiterreichen von Exceptions (1)

- Methoden müssen ihre Ausnahmen nicht lokal behandeln, sondern können sie auch weiterreichen.
- Exception wird an aufrufende Methode weitergereicht.
- Immer dann verwenden, wenn die Exception von der aufrufenden Methode sinnvoller behandelt werden kann (z.B. Parameter-Neueingabe).
- Die throws-Klausel einer Methode weißt darauf hin welche Exceptions geworfen bzw. weitergereicht werden könnten.
  - Typischerweise werden nur Checked Exceptions in der throws-Klausel angegeben
- Form

```
[Zugriffsmodifikatoren] Rückgabetyp Methodename([Parameter]) throws  
ExceptionType1, ExceptionType2, ...
```
- Beispiele

```
int test1(int x) throws IOException {...}  
int test2() throws FileNotFoundException, UnsupportedEncodingException {...}
```

# Weiterreichen von Exceptions (2)

- Ausnahmen können über mehrere Aufrufe durchgereicht werden.
- Eine Methode kann nur Checked Exceptions werfen und weiterreichen, welche in der throws-Klausel angegeben wurden.
- Unchecked Exceptions können immer geworfen werden.
- Compiler stellt sicher, dass keine Checked Exception einer untergeordneten Methode übersehen wird.
- Weitergereichte Exceptions sollten möglichst spezifisch sein.
- Siehe Beispiel



<src/at/ac/uibk/pm/exceptions/stack/ArrayStack.java>



<src/at/ac/uibk/pm/exceptions/stack/StackApplication.java>

# Beispiel nicht-lokale Behandlung

```
public void push(Object element) throws StackFullException {  
    if (position >= data.length) {  
        throw new StackFullException(position, data.length);  
    }  
    data[position] = element;  
    ++position;  
}  
  
public void push(Object[] elements) throws StackFullException {  
    for (Object element : elements) {  
        push(element);  
    }  
}
```

Exception wird hier geworfen.

Exception wird hier weitergereicht.

```
public static void main(String[] args) {  
    Stack stack = new ArrayStack(5);  
    try {  
        stack.push(args);  
    } catch (StackFullException e) {  
        System.out.println("Stack is full!");  
    }  
}
```

Exception wird hier behandelt.



# throws-Klausel (Überschreiben, Überladen)

- Überschreiben
  - throws-Klausel darf bei der Redefinition der Methode
    - Dieselben Exceptions wie Oberklasse auslösen
    - Exceptions spezialisieren
    - Exceptions weglassen
  - Bei der Redefinition dürfen in der throws-Klausel alle Typen aufscheinen, die zu irgendeinem Typ in der Signatur irgendeiner direkten oder indirekten Basisklassenmethode kompatibel sind (auch mehrere Subklassen für eine Exceptionklasse).
- Überladen
  - throws-Klauseln überladener Methoden sind völlig unabhängig.
  - Unterschiedliche throws-Klauseln reichen nicht aus, um Methoden zu überladen.
- Siehe externes Beispiel:



<src/at/ac/uibk/pm/exceptions/stack/ArrayStack.java>



<src/at/ac/uibk/pm/exceptions/stack/StackApplication.java>

# Exception-Chaining

- Wenn Methoden Ausnahmen immer weiterreichen
  - Sehr lange throws-Klausel
  - Detailfehler auf niederer Ebene müssen an ganz anderer Stelle behandelt werden.
  - Änderungen in der throws-Klausel würden viele Veränderungen nach sich ziehen.
- Lösung – Exception-Chaining
  - Mehrere Ausnahmen untergeordneter Aufrufe werden aufgefangen.
  - Die Ausnahmen werden zusammengefasst und in einer neuen Ausnahme weitergegeben.

# Exception-Chaining (Allgemeines Schema)

```
public void exceptionChaining() throws MyException
    try {
        ...
    } catch(E1 ex) {
        throw new MyException(ex);
    } catch(E2 ex) {
        throw new MyException(ex);
    }
}
```

Ausnahme vom Typ E2 wird in eine Ausnahme vom Typ MyException verpackt.

# Chaining bei eigenen Ausnahmen

- Entsprechende Konstruktoren implementieren, um Grund für Exception rekonstruieren zu können (Cause).
  - Exception-Klasse gibt Beispiele für solche Konstruktoren:  
Exception(String message, Throwable cause)  
Exception(Throwable cause)
  - Beispiel für Konstruktor für die MyException-Klasse:

```
MyException(Throwable cause) {  
    super(cause);  
}
```

- Oder initCause-Methode verwenden:

```
...  
catch (E1 ex) {  
    MyException m = new MyException();  
    m.initCause(ex);  
    throw m;  
}
```

- Siehe externes Beispiel:



<src/at/ac/uibk/pm/exceptions/stack/resourceloader/ResourceLoader.java>

# Nachverfolgung von Ausnahmen: StackTrace

- Methode `printStackTrace` gibt Aufrufstack aus, erbt von `Throwable` (Superklasse von `Error` und `Exception`).
- Damit kann man den genauen Verlauf der `Exception` (wie wurde weitergereicht?) inspizieren.

```
public static void main(String[] args) {  
    Stack stack = new ArrayStack(5);  
    try {  
        stack.push(args);  
    } catch (StackFullException e) {  
        System.out.println("Stack is full!");  
        e.printStackTrace();  
    }  
}
```

```
java StackApplication a b c d e f  
Stack is full!
```

```
at.ac.uibk.pm.exceptions.stack.StackFullException: Expected a stack with less than 5  
elements but got stack with 5 elements.  
at at.ac.uibk.pm.exceptions.stack.ArrayStack.push(ArrayStack.java:41)  
at at.ac.uibk.pm.exceptions.stack.ArrayStack.push(ArrayStack.java:56)  
at at.ac.uibk.pm.exceptions.stack.StackApplication.main(StackApplication.java:18)
```



# Multi-Catch (1)

- Manchmal sollen mehrere Ausnahmetypen gleichartig behandelt und dafür nur eine catch-Klausel verwendet werden.
- Fangen mehrerer Ausnahmen mit nur einem catch-Block möglich.
  - Wird als Multi-Catch bezeichnet.
  - Bei der Aufzählung werden die einzelnen Ausnahmen mit | getrennt.
  - Die Variable (im unteren Beispiel e) ist implizit final.

```
try {  
    ...  
} catch (MyException | OtherException e) {  
    ...  
}
```

Hier können sowohl Ausnahmen vom Typ MyException als auch vom Typ OtherException gefangen werden.

Gemeinsame Behandlung der unterschiedlichen Ausnahmen

# Multi-Catch (2)

- Die Abarbeitung erfolgt wie bei mehreren catch-Blöcken.
- Neben den Standard-Tests kommen neue Überprüfungen hinzu:
  - Kommt die exakt gleiche Ausnahme mehrfach in der Liste vor?
  - Gibt es Mehrdeutigkeit aufgrund von Mengenbeziehungen?
- Beispiel für fehlerhaftes Multi-Catch (Mengenbeziehung):

```
try {  
    ...  
} catch (FileNotFoundException | IOException e) {  
    ...  
}
```



# Final Rethrow (1)

- Immer dann, wenn in einem catch-Block ein throw stattfindet, ermittelt der Compiler die im try-Block tatsächlich aufgetretenen Typen der in der catch-Klausel geprüften Ausnahmen.
    - Der im catch genannte Typ für das rethrow wird beim Weiterreichen nicht berücksichtigt.
    - Statt dem gefangen Typ wird der Compiler den durch die Codeanalyse gefundenen Typ beim rethrow melden.
- Allgemeine Exception fangen, spezielle Exception werfen.

# Final Rethrow (2)

```
public class Exception1 extends Exception {}
```

```
public class Exception2 extends Exception {}
```

```
public class Test {  
    public void method1() throws Exception1 {...}  
    public void method2() throws Exception2 {...}  
    ...  
    public void method3() throws Exception1, Exception2 {  
        try {  
            method1();  
            method2();  
        } catch (Exception e) {  
            // do something  
            throw e;  
        }  
    }  
}
```

# Konstruktoren und Ausnahmen

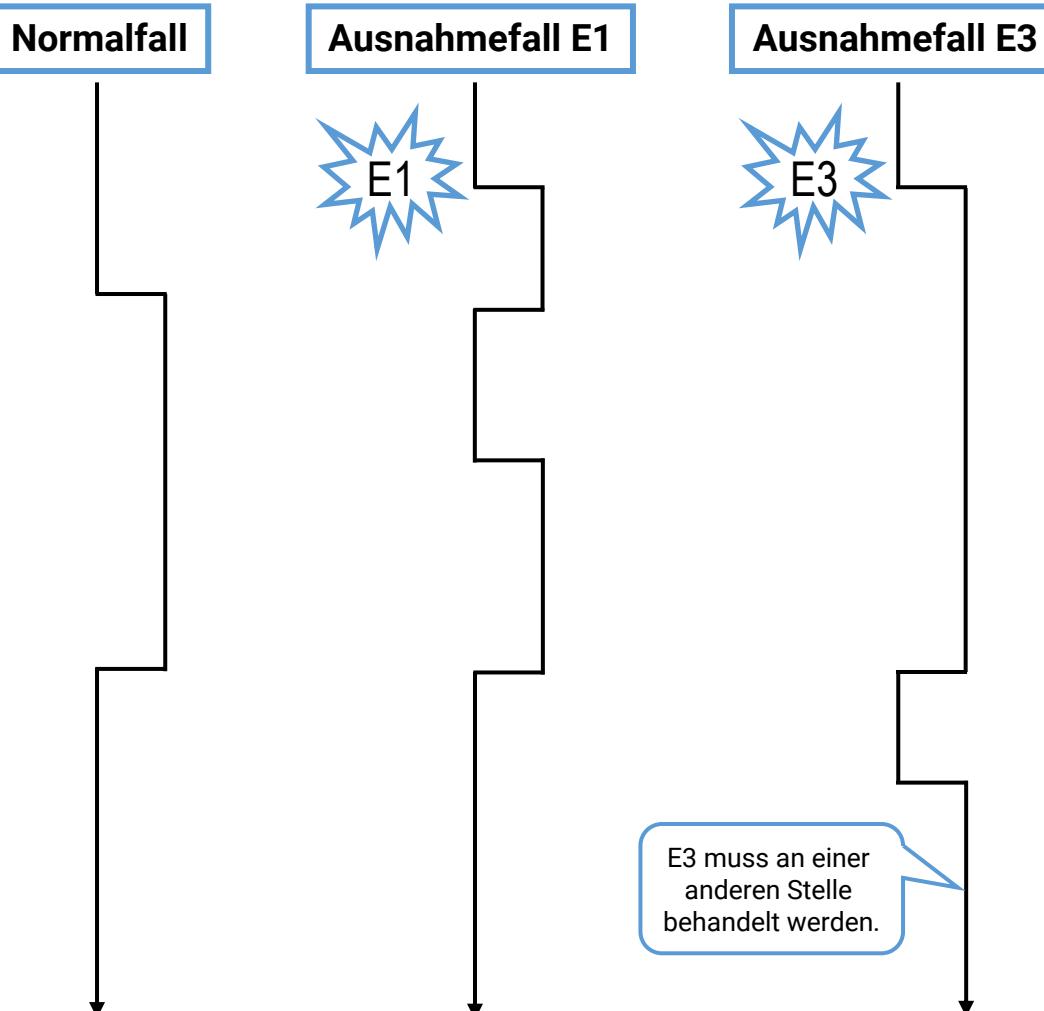
- Ausnahmen können auch in Konstruktoren verwendet werden.
  - Fehler beim Anlegen eines Objekts sollten vermieden werden.
  - Objekt könnte in einem inkonsistenten Zustand sein.
- Der Konstruktor sollte die Ausnahme weiterreichen (`throws`).
- Aufrufende Methode kann dann entsprechend auf die gescheiterte Erzeugung des Objekts reagieren.
- Vorsicht bei `finally` → wird immer aufgerufen

# finally

- Die try-Anweisung kann nach den catch-Blöcken **eine finally-Klausel** enthalten.
- Die Anweisungen im finally-Block werden immer als Abschluss der try-Anweisung ausgeführt, egal ob eine Ausnahme auftrat oder nicht.
- Hat den Zweck Arbeiten, die im try-Block begonnen wurden, sauber abzuschließen.
- Beispiel
  - Im try-Block wird eine Datei geöffnet.
  - Die Datei muss wieder geschlossen werden (egal ob bei der Verarbeitung der Daten ein Fehler auftritt oder nicht).
  - Hinweis: Ab Java 7 gibt es noch eine andere Möglichkeit. Diese wird im Foliensatz über Streams noch ausführlich besprochen.
- Auch bei verschachtelten try-Anweisungen möglich!

# finally (Abarbeitung)

```
try {  
    ...  
    ...  
} catch (E1 e) {  
    ...  
} catch (E2 e) {  
    ...  
} finally {  
    ...  
}
```



# Empfehlungen Exception-Handling (1)

- Art der Ausnahme
  - Wenn ein Aufrufer eine außergewöhnliche Situation behandeln kann, so sollte eine Checked Exception geworfen werden.
  - Ist ein Aufrufer nicht in der Lage, eine Fehlersituation zu korrigieren, so sollte eine Unchecked Exception verwendet werden.
- Behandlung von Ausnahmen
  - Behandle auftretende Ausnahmen, wenn dies sinnvoll möglich ist.
  - Propagiere im Zweifelsfall Ausnahmen an höhere Aufrufebenen weiter.
    - Dabei sollte aber immer ein möglichst aussagekräftiger Ausnahmetyp gewählt werden!
- Verwendung der try-Anweisung
  - try-Anweisung findet sich meist in übergeordneten Methoden.
  - Je mehr über den aktuellen Programmzustand vorhanden ist, desto besser lässt sich angemessen auf einen Fehler reagieren.

# Empfehlungen Exception-Handling (2)

- Standard Exceptions der Java API sollten bevorzugt werden.
- Ausnahmen werden oft für Vorbedingungen verwendet.
  - Vorbedingungen beziehen sich oft auf Parameter.
  - Die Parameter dürfen meist nicht beliebige Werte annehmen.
  - Ohne Überprüfung kann ein Objekt in einen falschen (inkonsistenten) Zustand geraten.

Exception	Mögliche Anwendung
IllegalArgumentException	Wert eines Parameters weist einen ungeeigneten Wert auf
IllegalStateException	Zustand eines übergebenen Objektes ist unzureichend
NullPointerException	Parameter weist den Wert null auf
IndexOutOfBoundsException	Indexwert ist außerhalb des Bereiches
UnsupportedOperationException	Hinweis auf eine fehlende Implementierung

# Handling von unerwarteten Parametern

```
private static final int MAX_REPETITIONS = 5;

public static void printHelloWorld(int repetitions) {
    if (repetitions < 0) {
        throw new IllegalArgumentException();
    } else if (repetitions <= MAX_REPETITIONS) {
        for (int i = 0; i < repetitions; ++i) {
            System.out.println("Hello World!");
        }
    } else {
        throw new IllegalArgumentException();
    }
}
```





# Fail Fast

```
private static final int MAX_REPETITIONS = 5;

public static void printHelloWorld(int repetitions) {
    if (repetitions < 0 || repetitions > MAX_REPETITIONS) {
        throw new IllegalArgumentException();
    }
    for (int i = 0; i < repetitions; ++i) {
        System.out.println("Hello World!");
    }
}
```



- Vorher:
  - Alle Zweige sind verbunden und müssen verstanden werden.
- Nachher:
  - Lesbarer Code
  - Zuerst wird der Parameter validiert und anschließend wird der Hauptteil der Methode abgearbeitet.



# Explain Cause in Message (1)

```
private static final int MAX_REPETITIONS = 5;

public static void printHelloWorld(int repetitions) {
    if (repetitions < 0) {
        String error = String.format("Expected repetitions >= 0 but got %s.", repetitions);
        throw new IllegalArgumentException(error);
    }
    if (repetitions > MAX_REPETITIONS) {
        String error = String.format("Expected repetitions <= %d but got %s.",
            MAX_REPETITIONS, repetitions);
        throw new IllegalArgumentException(error);
    }
    for (int i = 0; i < repetitions; ++i) {
        System.out.println("Hello World!");
    }
}
```





# Explain Cause in Message (2)

- Vorher:
  - Keine detaillierten Informationen über die aufgetretene Exception.
- Nachher:
  - Grund der Exception wird klarer gemacht.
  - Alle Parameter und Felder, welche zum Auftreten der Exception beigetragen haben, wurden kommuniziert.
  - Grundschema:
    - Was wurde erwartet?
    - Was wurde übergeben?
    - Was liegt im Objekt vor?

# Verwendung von eigenen Exceptions

- Falls keine der Standard-Exceptions zutrifft, können eigene Exceptions erstellt werden.
  - Subklassen von `Exception`, `RuntimeException` oder einer Subklasse
- Trade-Off
  - Wenige Einzelklassen: schlechte Differenzierung, aber kurze `throws`-Klauseln oder `catch`-Listen
  - Viele Einzelklassen: feingranulare Behandlung, aber lange `throws`-Klauseln oder `catch`-Listen
- Hierarchien aufbauen
  - Spezielle Klassen für differenzierte Fehlersituationen
  - Spezielle Klassen nach Fehlerart gruppieren und mit gemeinsamen Basisklassen versehen
  - Detaillierte Ausnahmen anbieten
    - Ausnahmen beinhalten weitere Objektvariablen.
    - Beim Werfen der Ausnahme werden diese Objektvariablen belegt (z.B. mit den falschen Parameterwerten etc.).
    - Damit stehen bei der Fehlerbehandlung mehr Informationen zur Verfügung!
- Wann soll von `RuntimeException` abgeleitet werden?
  - Wenn das aufgezeigte Problem von vornherein vermieden werden könnte!

# Verwendung der verschiedenen Basisklassen

- Afangen in catch-Klauseln
  - Error – nein!
  - Exception – nein (würde alle Subklassen abfangen), nur Subklassen davon!
  - RuntimeException – nein (zu allgemein), meist ein Indiz für einen Fehler im Programm, der behoben werden sollte! (siehe clean code Tipp nächste Folien)
- Werfen mit throw-Anweisung
  - Error – nein, im Allgemeinen der JVM bzw. Assertions vorbehalten!
  - Exception – nein (zu allgemein), nur mit einer entsprechenden Subklasse!
  - RuntimeException – nein (zu allgemein), nur mit einer entsprechenden Subklasse!

# Beispiel Runtime-Probleme

```
public boolean equals(Object other) {  
    if (this == other) {  
        return true;  
    }  
  
    Product otherProduct = (Product) other;  
    return productID == otherProduct.productID;  
}
```





# Always Check Type Before Cast

```
public boolean equals(Object other) {  
    if (this == other) {  
        return true;  
    }  
    if (!(other instanceof Product otherProduct)) {  
        return false;  
    }  
    return productID == otherProduct.productID;  
}
```



- Vorher:
  - ClassCastException (Runtime-Exception) möglich
- Nachher:
  - ClassCastException nicht mehr möglich – zur Laufzeit kann dieser Fehler nicht mehr auftreten



# Ausnahmen - Don'ts (1)

- Eine Ausnahme sollte **niemals** grundlos ignoriert werden!
  - Fehler verschwinden nicht von selbst!
  - Programm wird möglicherweise nicht richtig funktionieren (inkonsistente Daten etc.).
  - Soll eine Exception ignoriert werden:
    - Soll der catch-Block eine Begründung mittels Kommentar enthalten
    - Soll die Variable ignored benannt werden.

```
try {  
    ...  
} catch (MyException e) {  
}
```



```
try {  
    ...  
} catch (MyException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```



# Ausnahmen - Don'ts (2)

- Ausnahmen sollten niemals Kontrollstrukturen ersetzen!!!
  - z.B. bei einem Arraydurchlauf nicht auf die Länge überprüfen, sondern auf Ausnahme warten und dann abfangen.
- Rückgabe von null statt einer Ausnahme im Fehlerfall
  - Ausnahmen müssen behandelt werden, null kann ignoriert werden!
  - Rückgabe von null ist nur in sehr wenigen Fällen sinnvoll.
  - Muss in der aufrufenden Methode separat behandelt werden.

# Assertions

# Assertions allgemein

- Eine Assertion (Zusicherung) ist ein boolescher-Ausdruck, der immer zutreffen muss.
  - Assertions werden beim Programmablauf von der JVM überwacht.
  - Programmabbruch, wenn eine Assertion **nicht** zutrifft (Error)!
- Assertions sind einfache Anweisungen.
  - Form

```
assert Ausdruck1;  
assert Ausdruck1 : Ausdruck2;
```
  - Ausdruck1 muss true liefern, ansonsten wirft die Assertion eine Exception vom Typ `AssertionError`.
  - Ausdruck2 ist optional und wird nur ausgewertet, wenn Ausdruck1 false ist.
  - Ausdruck2 wird dem Konstruktor der Klasse `AssertionError` übergeben, um den Fehler genauer zu beschreiben.
  - Beispiele

```
assert a <= b && b <= c;  
assert x > 0 : "x must be positive!";
```

# Arbeitsweise von Assertions

- Assertions werden zur Laufzeit ausgewertet.
- Wenn das Ergebnis `false` ist, stoppt das Programm mit einem `AssertionError`.
- Die nachfolgenden Anweisungen werden nicht mehr ausgeführt.
- Bei Abbruch wird Information über den Ort der gescheiterten Assertion ausgegeben.

# Aktivierung von Assertions

- Assertions sind standardmäßig deaktiviert.
- Assertions kosten Rechenzeit.
- Sie können zur Laufzeit wahlweise aktiviert werden.
  - Deaktiviert sind sie automatisch.
  - Programm muss nicht neu übersetzt werden.
- Aktivierung im Programm Test (enable assertions)  
`java -ea Test`
- Deaktivierung (default, muss nicht angeführt werden; disable assertions)  
`java -da Test`
- In Eclipse
  - Unter Run->RunConfigurations Reiter Arguments auswählen.
  - -ea im Feld Vm arguments angeben.
- In IntelliJ
  - Unter Run -> Edit Configurations...
  - Klick auf Modify options und Add VM options
  - -ea im Feld VM options eingeben
- Normalerweise nur **während der Entwicklungszeit** aktiviert!

# Selektive Aktivierung

- Assertions können auch selektiv auf der Ebene von Klassen und Packages getrennt aktiviert werden.
- Assertions in Klasse `classname` aktivieren:  
`-ea:classname`
- Assertions in Package `packagename` (3 Punkte müssen angegeben werden):  
`-ea:packagename...`
- Mehrere Schalter erlaubt:  
`java -ea:project2... -da:project2.datastore...`  
`-ea:project2.datastore.Store Test`

# Anwendung von Assertions

- Überprüfung von Parametern, die an nicht-öffentliche Methoden übergeben werden (nur falsch, wenn das eigene Programm fehlerhaft ist).
- Verwendung in Nachbedingungen, die am Ende einer Methode erfüllt sein müssen.
- Überprüfung von Schleifeninvarianten.
- Markierung von Codeblöcken die nicht erreicht werden sollten (Assertion mit `false` als Argument).
- Nicht für Bedingungen geeignet, die von irgendwelchen äußeren Einflüssen abhängen.
  - z.B. am Anfang einer öffentlichen Methode die Werte der Parameter überprüfen.

# Beispiel (1)

```
private static int min(int a, int b) {
    int minimum = a <= b ? a : b;
    assert minimum <= a && minimum <= b;
    return minimum;
}

private static int max(int a, int b) {
    int maximum = a <= b ? a : b;
    assert maximum >= a && maximum >= b;
    return maximum;
}

public static void print(int a, int b) {
    System.out.printf("%d is less than or equal to %d%n", min(a, b), max(a, b));
}

public static void main(String[] args) {
    print(1, 1);
    print(5, 2);
    print(3, 9);
}

java AssertionApplication
1 is less than or equal to 1
2 is less than or equal to 2
3 is less than or equal to 3
```

# Beispiel (2)

```
private static int min(int a, int b) {  
    int minimum = a <= b ? a : b;  
    assert minimum <= a && minimum <= b;  
    return minimum;  
}  
  
private static int max(int a, int b) {  
    int maximum = a <= b ? a : b;  
    assert maximum >= a && maximum >= b;  
    return maximum;  
}  
  
public static void print(int a, int b) {  
    System.out.printf("%d is less than or equal to %d%n", min(a, b), max(a, b));  
}  
  
public static void main(String[] args) {  
    print(1, 1);  
    print(5, 2);  
    print(3, 9);  
}
```

```
java -ea AssertionApplication  
1 is less than or equal to 1  
Exception in thread "main" java.lang.AssertionError  
at at.ac.uibk.pm.exceptions.AssertionApplication.max(AssertionApplication.java:12)  
at at.ac.uibk.pm.exceptions.AssertionApplication.print(AssertionApplication.java:17)  
at at.ac.uibk.pm.exceptions.AssertionApplication.main(AssertionApplication.java:22)
```

Programmierfehler

# Quellen

- Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018

# Unit Tests

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller  
Universität Innsbruck

# Testen

# Testen (1)

- Testen ist der Vorgang, ein Programm oder einen Teil davon mit der Absicht auszuführen, möglicherweise enthaltene Fehler zu finden.
- Für die Bestimmung des Testergebnisses werden die spezifizierten Anforderungen (Soll) mit den gelieferten Ergebnissen (Ist) verglichen.
  - Pass: Gelieferte Ergebnisse stimmen mit den spezifizierten Anforderungen überein.
  - Fail: Gelieferte Ergebnisse stimmen nicht mit den spezifizierten Anforderungen überein.
  - Error: Während der Ausführung des Tests trat ein unerwarteter Fehler auf.
- Für das Testen ist es erforderlich, dass die Anforderungen bekannt sind.
- Testen dient der Sicherstellung von Softwarequalität.
- Debugging != Testen

# Testen (2)

- Auswahl Testfälle immer abhängig von der zu testenden Anwendung.
- Um die Anzahl der Tests möglichst klein zu halten, werden jene Tests gewählt, die das Programm in kritische Situationen bringen.
  - In diesen Situationen sind Fehler am ehesten zu erwarten.
- Zwei Vorgehensweisen bei der Konstruktion eines Testfalls
  - Black-Box-Test
  - White-Box-Test

# Black-Box-Test

- Nur die Anforderungen des Programms werden berücksichtigt.
  - Die Schnittstelle gibt die Testfälle vor.
- Der Source-Code spielt keine Rolle, das Programm wird als Black-Box betrachtet.
- Änderungen am Quelltext (nicht Schnittstelle) erfordern keine neue Implementierung der Tests.
- Black-Box-Testtechniken sind beispielsweise Äquivalenzklassentests, Grenzwertanalyse, kombinatorisches Testen.

# White-Box-Test

- Test orientiert sich am Quelltext des Programms.
- Änderungen am Quelltext erfordern teilweise neue Tests bzw. alte Tests werden überflüssig.
- White-Box-Testtechniken können in kontrollflussorientierte und datenflussorientierte Überdeckungskriterien unterteilt werden.
  - Kontrollflussorientierte Tests: Mit den Testfällen sollen – je nach Überdeckungskriterium – die einzelnen Anweisungen, Zweige, Bedingungen oder Pfade explizit ausgetestet werden.
  - Datenflussorientierte Tests: Zugriffe auf Variablen sind maßgeblich für die Testerstellung

# Anweisungsüberdeckung

- Anweisungsüberdeckung ist ein kontrollflussorientiertes White-Box-Überdeckungskriterium.
- Dabei sollen alle Anweisungen mindestens einmal ausgeführt werden.

```
public static int hammingDistance(final String first, final String second) {  
    if (first == null || second == null) {  
        throw new IllegalArgumentException("error description...");  
    }  
    if (first.length() != second.length()) {  
        throw new IllegalArgumentException("error description...");  
    }  
    int distance = 0;  
    for (int i = 0; i < first.length(); ++i) {  
        if (first.charAt(i) != second.charAt(i)) {  
            ++distance;  
        }  
    }  
    return distance;  
}
```

- Testfälle:
  - `hammingDistance(null, null);`
  - `hammingDistance("len", "length");`
  - `hammingDistance("test", "task");`

# Testarten (Überblick)

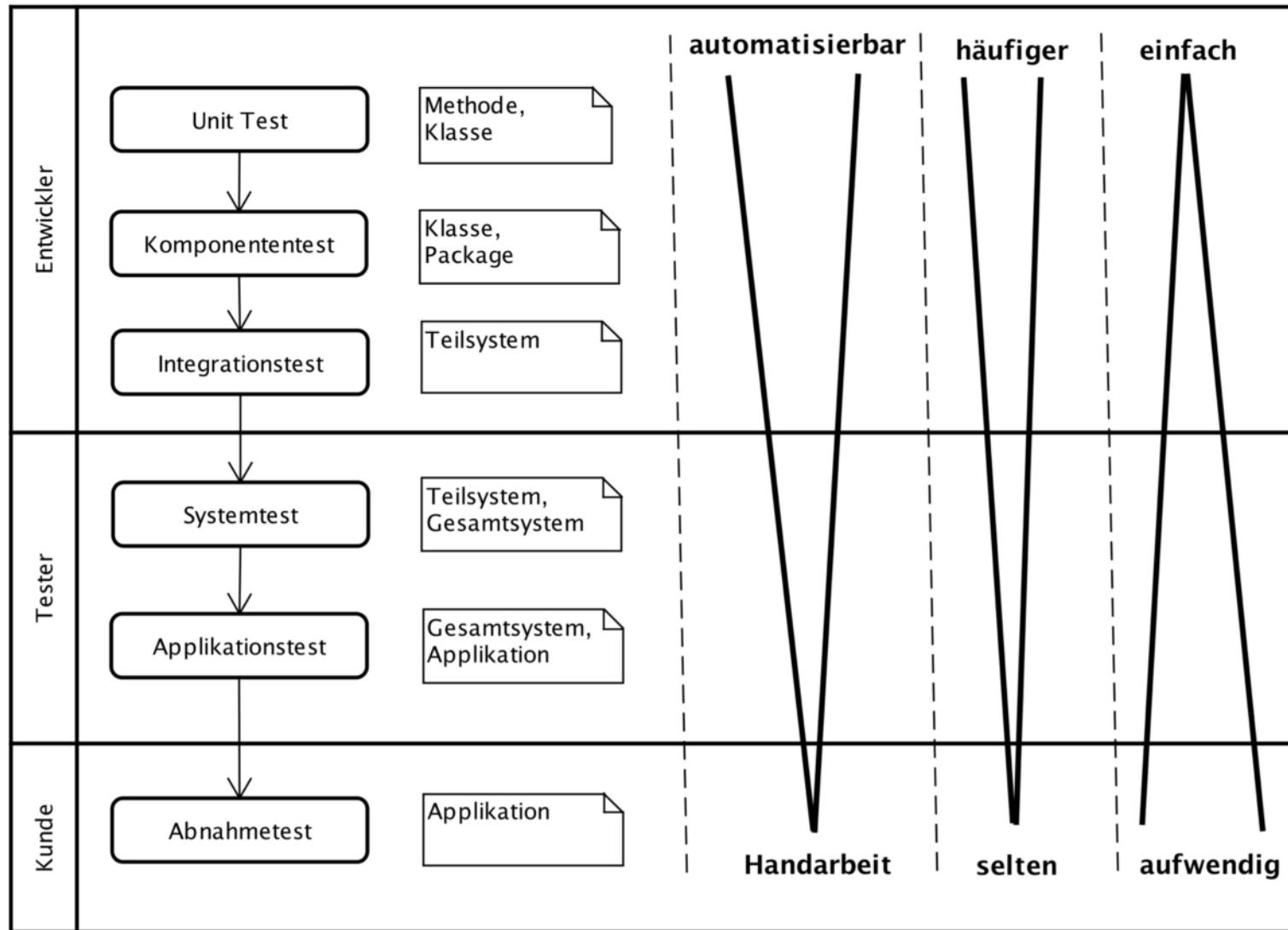


Abbildung entnommen aus „Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung.“, Michael Inden

# Abgrenzung zur Verifikation

- Verifikation = formaler Korrektheitsbeweis
  - Es wird versucht, mit formalen Methoden den Nachweis zu führen, dass ein Programm nur richtige Ergebnisse produzieren kann.
  - Liefert eine endgültige Aussage zur Korrektheit.
  - Schon für sehr kleine Programme kann dieser Ansatz sehr aufwendig sein!
- Testen = systematisches Ausprobieren
  - Es wird eine bestimmte Anzahl von Tests konstruiert, mit denen das Programm „probeweise“ ausgeführt wird.
  - Mit Testen kann aber nur die Anwesenheit von Fehlern nachgewiesen werden, nicht aber deren Abwesenheit!
  - In der Regel wird eine sehr große Anzahl von Testfällen benötigt um ein Programm ausführlich zu testen.
    - Das verursacht aber meist sehr hohe Kosten.
    - Endgültiger Korrektheitsbeweis ist damit auch nicht möglich.

# **Unit-Tests mit JUnit**

# Unit-Tests (1)

- Beim Unit-Test wird eine kleine Einheit eines Programms betrachtet und auf mögliche Fehler untersucht.
- Die Einheit wird isoliert vom Rest des Gesamtsystems betrachtet.
  - Testen eines ausgewählten Softwarebausteins (meist Methoden)
- Unit-Tests können automatisiert werden.
  - Testfälle werden direkt als Quellcode entwickelt.
- Testen sollte immer ein wichtiger Bestandteil des Entwicklungsprozesses sein.
- Es gibt Programmiertechniken, bei denen dieses Testen ein elementarer Teil des Entwicklungsprozesses ist.
  - Extreme Programming (XP)
  - Test-Driven Development (TDD)

# Unit-Tests (2)

- FIRST-Prinzipien

- Fast: schnell ablaufen (sonst werden sie seltener ausgeführt).
- Independent: keine Abhängigkeiten zwischen Tests.
- Repeatable: wiederholt in jeder Umgebung ausführbar.
- Self-Validating: durch Assertions (keine manuellen Überprüfungen).
- Timely: zeitnah geschrieben (kurz vor oder nach Produktionscode).

# Test-Driven Development

- Schritte:
  1. Zuerst werden die Ergebnisse in Form von Testfällen implementiert.
    - Damit die Testfälle übersetzt werden können, werden zunächst nur leere Methoden oder Methoden mit einer einzigen return-Anweisung bereitgestellt.
      - Übersetzung funktioniert, Tests schlagen noch alle fehl.
  2. Danach wird der Code entwickelt.
    - Die Methodenrümpfe werden schrittweise vervollständigt, bis am Ende alle Tests fehlerfrei durchlaufen werden.
  3. Räume den Anwendungscode auf (Refactoring)
    - Beispielsweise Code-Duplikate entfernen, Code-Richtlinien einhalten usw.
    - Nach dem Refactoring laufen alle Tests immer noch fehlerfrei durch. Beginne wieder bei Schritt 1.
- Vorteile bei diesem Vorgehen
  - Tests werden zuerst erstellt.
  - Tests können nicht vernachlässigt werden.
  - Tests geben notwendige Funktionalität vor.

# JUnit

- JUnit bietet ein einheitliches Framework zur Organisation und systematischen Durchführung von Unit-Tests in Java.
- Aktuelle Version 5.8.2 (November 2021)
- Weit verbreitetes Unit-Testing-Framework
- Die Bibliothek ist nicht Teil des JDK sondern ist unter <https://junit.org/junit5/> erhältlich und dokumentiert.
  - Installation am einfachsten direkt über die IDE.
- Testfälle werden direkt als Java-Programme erstellt.

# Exkurs: Annotationen

- Anführen von Metadaten im Quelltext (seit Java 1.5), nicht Teil des eigentlichen Quellcodes.
- Annotationen starten immer mit einem @-Zeichen.
- Annotationen beziehen sich auf das folgende Code-Element (z.B. Klasse, Methode, Feld).
- Java API Beispiele:
  - `@Override`
    - Annotierte Methode überschreibt eine Methode einer Superklasse bzw. eines Interfaces.
  - `@Deprecated`
    - Markiert veraltete Code-Elemente.
- JUnit-Beispiele:
  - `@Test`
    - Markiert eine Methode als Testmethode.
  - `@DisplayName`
    - Deklaration einer benutzerdefinierten Testbeschreibung.
  - `@ParameterizedTest`
    - Markiert eine Methode als parametrisierte Testmethode.
  - `@Disabled`
    - Markiert eine Testklasse oder Testmethode als deaktiviert.

# Testklasse

- Es gibt eine Top-Level-Testklasse für jede zu testende Klasse.
- Die Testklasse darf nicht abstrakt sein.
- Die Testklasse muss genau einen Konstruktor haben.
- Die Testklasse importiert benötigte JUnit-Klassen und Methoden.
  - Für das Importieren der JUnit-Methoden werden meist statische Imports verwendet.
    - Beispiel:  
`import static org.junit.jupiter.api.Assertions.*;`
- Die Testklasse definiert Testmethoden und Verwaltungsmethoden.
  - Diese Methoden dürfen weder abstrakt noch privat sein.
  - Der Rückgabewert der Methoden muss void sein.

# Testmethoden

- Testmethoden müssen mit `@Test`, `@ParameterizedTest`, `@RepeatedTest`, `@TestFactory` oder `@TestTemplate` gekennzeichnet werden.
- Jede Testmethode sollte jeweils nur eine Funktionalität überprüfen (meist ein Assert pro Testmethode).
  - Der Methodename ist frei wählbar (sollte den Testfall beschreiben)
- Üblicherweise besteht ein Test aus den drei Teilen Given, When, Then (GWT-Stil).
  - Given – Voraussetzungen für den Testfall werden aufgestellt.
  - When – Aktionen die im Testfall überprüft werden sollen.
  - Then – Abgleich der erwarteten Ergebnisse mit den berechneten Werten.

# Verwaltungsmethoden

- Zweck der Verwaltungsmethode wird festgelegt mit einer Annotation.
  - `@BeforeEach`
    - Wird vor jedem einzelnen Test aufgerufen.
  - `@AfterEach`
    - Wird nach jedem einzelnen Test aufgerufen.
  - `@BeforeAll`
    - Wird einmal vor allen Tests aufgerufen (muss static sein).
  - `@AfterAll`
    - Wird einmal nach allen Tests aufgerufen (muss static sein).
- Diese Verwaltungsmethoden können für die Initialisierung und das Zurücksetzen von Daten hilfreich sein.
- Allerdings können diese Verwaltungsmethoden die Lesbarkeit einzelner Testfälle stark einschränken.

# Assertions-Klasse (1)

- In der Assertions-Klasse bietet JUnit verschiedene Methoden an, um Annahmen im Test zu überprüfen.
- Methoden für den Vergleich von Wahrheitswerten:
  - `assertFalse(boolean condition)`
  - `assertTrue(boolean condition)`
  - ...
- Methoden für den Vergleich von Werten `expected` und `actual`:
  - `assertEquals(long expected, long actual)`
  - `assertEquals(double expected, double actual)`
  - `assertEquals(double expected, double actual, double delta)`
  - `assertEquals(Object expected, Object actual)`
  - ...

# Assertions-Klasse (2)

- Methoden für den inhaltlichen Vergleich von Arrays:
  - `assertArrayEquals (int[] expected, int[] actual)`
  - ...
- Methoden zur Überprüfung, ob eine erwartete Exception geworfen wird:
  - `assertThrows(Class<T> expectedType, Executable executable)`
  - ...
- Methoden zur Überprüfung, ob eine variable Anzahl von Assertions stimmen:
  - `assertAll(Executable... executables)`
  - ...
- Viele weitere, siehe [API-Dokumentation!](#)
- Alle Vergleichsmethoden sind überladen mit zusätzlichem Parameter `message`.
  - Text wird beim Fehlschlagen des Tests ausgegeben.
    - `assertFalse(boolean condition, String message)`

# Beispiel Assertions

```
@Test  
public void sizeEmpty() {  
    final ArrayStack stack = new ArrayStack();  
  
    final int size = stack.size();  
  
    assertTrue(0 == size);  
}
```



```
public class ArrayStack implements Stack {  
  
    private final String[] data;  
    private int position = 0;  
    ...  
    public int size() {  
        return position;  
    }  
    ...  
}
```





# Use Meaningful Assertions

```
@Test  
public void sizeEmpty() {  
    final ArrayStack stack = new ArrayStack();  
  
    final int size = stack.size();  
  
    assertEquals(0, size);  
}
```



- Vorher:
  - Vergleich ist über assertTrue bzw. assertFalse durchaus möglich.
  - Im Fehlerfall geht allerdings Information verloren.
  - Feedback wenig aussagekräftig (z.B. AssertionFailedError: expected: <true> but was: <false>)
- Nachher:
  - assert-Methode ist auf Vergleich abgestimmt
  - Feedback ist detaillierter (z.B. AssertionFailedError: expected: <0> but was: <3>)



# Describe Your Tests

```
@Test  
@DisplayName("an empty stack should have size 0")  
public void sizeEmpty() {  
    final ArrayStack stack = new ArrayStack();  
  
    final int size = stack.size();  
  
    assertEquals(0, size);  
}
```



- Vorher:
  - Bei den Testergebnissen werden die Methodennamen angezeigt.
- Nachher:
  - Bei den Testergebnissen wird die Beschreibung aus DisplayName angezeigt.
  - Der Testfall ist dokumentiert.



# Exceptions

- Testen des Normalfalls:
  - Werden Methoden, welche Checked Exceptions werfen, getestet, können die aufgetretenen Exceptions in der Testmethode weitergereicht werden.
- Testen des Ausnahmefalls:
  - Überprüfung, ob erwartete Exception wirklich geworfen wird.
  - Test ist erfolgreich, wenn die erwartete Exception von der getesteten Methode geworfen wird.
  - Test scheitert, wenn die Exception nicht geworfen wird.

# Testen des Ausnahmefalls

```
@Test  
@DisplayName("pop on empty stack")  
public void popEmpty() {  
    final ArrayStack stack = new ArrayStack();  
  
    try {  
        stack.pop();  
        fail("Test should fail since stack is empty.");  
    } catch (EmptyStackException ignored) {  
    }  
}
```



# < > Let Framework Handle Exceptions

```
@Test  
@DisplayName("pop on empty stack")  
public void popEmpty() {  
    final ArrayStack stack = new ArrayStack();  
    Executable when = () -> stack.pop();  
    assertThrows(EmptyStackException.class, when);  
}
```

Executable aus org.junit.jupiter.api.function

Executable when = () -> stack.pop(); Lambda-Ausdruck (mehr dazu in der VO zu funktionaler Programmierung)



- Vorher:
  - Beim Lesen muss die Bedeutung (Semantik) der Variablen selbst herausgefunden werden.
  - Variablen geben keinerlei Auskunft über Inhalt.
- Nachher:
  - Lesbarer Code



# Parametrisierte Tests

- Ausführen desselben Tests mit unterschiedlichen Daten, welche als Argument der Testmethode übergeben werden.
- Diese Tests werden mithilfe der Annotation `@ParameterizedTest` realisiert.
- Zusätzlich muss eine Quelle für die Argumente festgelegt werden.
- Beispiele möglicher Quellen:
  - `@ValueSource`
    - Array aus Literalen
  - `@CsvSource`
    - Liste aus Comma-separated-values

# Beispiel parametrisierte Tests (1)

```
public static boolean isPrime(final int primeCandidate) {  
    if (primeCandidate <= 1) {  
        return false;  
    }  
    for (int divisor = 2; divisor * divisor <= primeCandidate; ++divisor) {  
        if (primeCandidate % divisor == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
@ParameterizedTest(name = "isPrime({0}) => true")  
@ValueSource(ints = {2, 3, 5, 7, 11, 13, 17, 19, 15601})  
void isPrime(int value) {  
    assertTrue(MathUtils.isPrime(value));  
}
```



# Beispiel parametrisierte Tests (2)

```
public static long fibonacciNumber(final int n) {  
    if (n < 0) {  
        throw new IllegalArgumentException(  
            String.format("Expected non-negative integer but got %d", n));  
    }  
    long previous = 0;  
    long current = 1;  
    if (n <= 1) {  
        return n;  
    }  
    for (int i = 2; i <= n; ++i) {  
        final long currentTmp = current;  
        current = Math.addExact(current, previous);  
        previous = currentTmp;  
    }  
    return current;  
}
```

```
@ParameterizedTest(name = "fibonacciNumber({0}) => {1}")  
@CsvSource({"0, 0", "1, 1", "2, 1", "3, 2", "4, 3", "5, 5", "6, 8"})  
void fibonacciNumberInput(final int input, final int expectedOutput) {  
    assertEquals(expectedOutput, MathUtils.fibonacciNumber(input));  
}
```



# Objekte mit Abhangigkeiten (1)

- Bei einem Element, welches getestet werden soll, wird von object-under-test oder system-under-test (SUT) gesprochen.
- Ein Kollaborateur ist ein Element, welches durch das SUT aufgerufen wird.
- SUT und Kollaborateure konnen sich beeinflussen.
  - Indirekte Eingabe
    - Kollaborateure beeinflussen das SUT beispielsweise uber Ruckgabewerte von Methoden, Verndern des Zustands eines Parameters oder Werfen einer Ausnahme.
  - Indirekte Ausgabe
    - SUT beeinflusst den Zustand eines Kollaborateurs.
- Eine Einheit soll bei Unit-Tests isoliert vom Rest des Gesamtsystems betrachtet werden.
- Was wird als eine Einheit betrachtet?
  - Klassischer bzw. Detroit-Style: Eine Klasse inklusive der Abhangigkeiten
  - London-Style: Eine Klasse

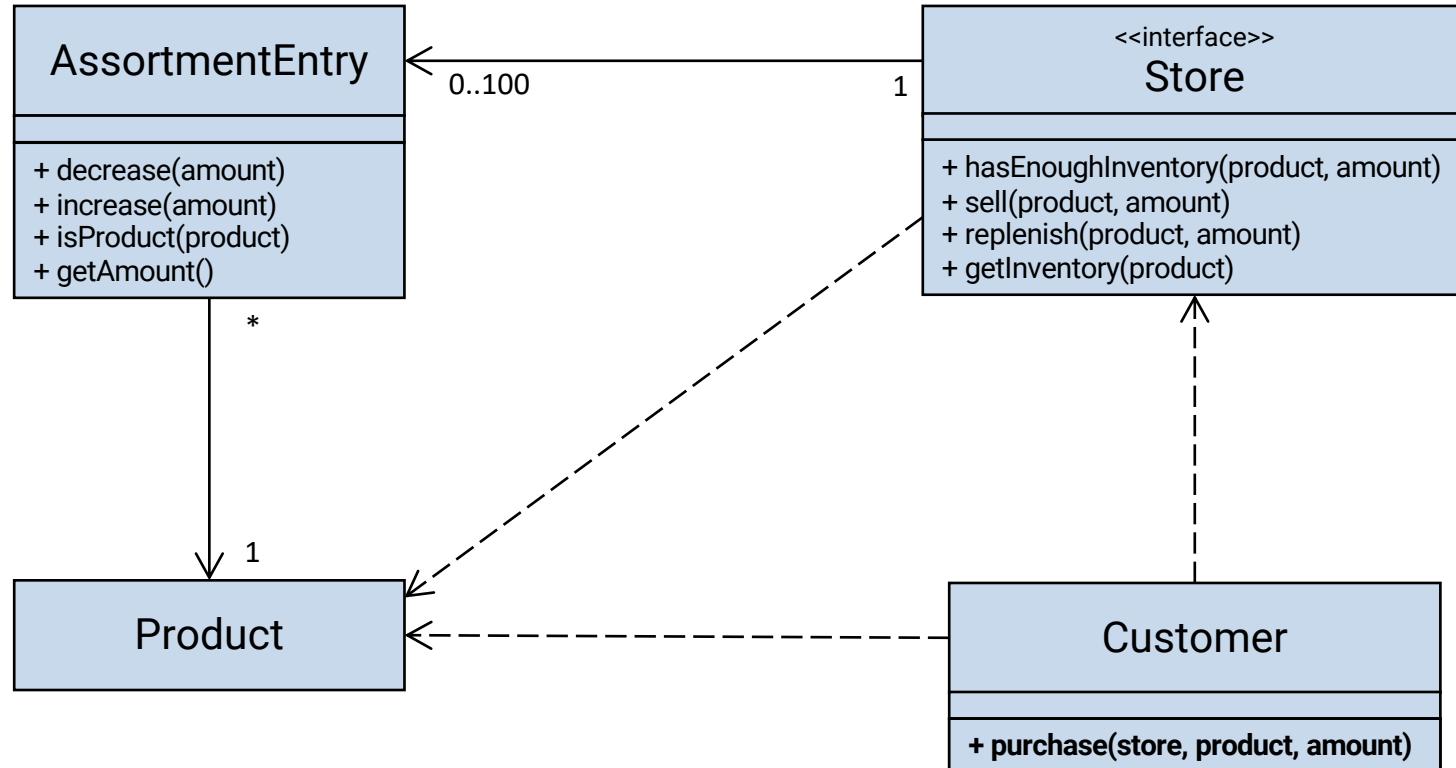
# Objekte mit Abhängigkeiten (2)

- Klassischer bzw. Detroit-Style:
  - Verwendung von realen Objekten, wenn möglich.
  - Tests dürfen keine Abhängigkeiten teilen, welche das Ergebnis anderer Tests oder die Wiederholbarkeit beeinflussen.
    - Dateisystem
    - Datenbank
    - Datum
    - Zeit
    - ...
  - Test-Doubles werden eingeführt, um solche Abhängigkeiten zu isolieren.
- London-Style:
  - Verwendung eines Mocks für jedes Objekt mit relevantem Verhalten.

# Test-Doubles

- Dummy
  - Objekt, das als Parameter übergeben aber nicht verwendet wird.
- Fake
  - Alternative und vereinfachte Implementierung mit abgewandelter Funktionsweise.
- Stub
  - Minimalistische Implementierung mit vordefinierten Rückgabewerten für Methoden, welche während des Tests aufgerufen werden.
- Mock
  - Objekt, welches das vordefinierte Verhalten mit erwarteten Methodenaufrufen überprüft.
  - Dynamische Generierung durch Frameworks wie beispielsweise [Mockito](#), [EasyMock](#) oder [jMock](#).

# Beispiel Objekte mit Abhängigkeiten (1)



# Beispiel Objekte mit Abhängigkeiten (2)

```
public class Customer {  
  
    public boolean purchase(Store store, Product product, int amount) {  
        if (!store.hasEnoughInventory(product, amount)) {  
            return false;  
        }  
        store.sell(product, amount);  
        return true;  
    }  
}
```

# Beispiel Objekte mit Abhängigkeiten (3)

- Klassischer Ansatz: Es werden keine Test-Doubles verwendet, da es keine geteilten Abhängigkeiten gibt.

```
@Test
@DisplayName("purchase some of the available products")
public void purchaseSomeProducts() {
    Customer customer = new Customer();
    Store store = new GroceryStore();
    Product product = new Product("Milk", 129);
    store.replenish(product, 10);

    boolean success = customer.purchase(store, product, 7);

    assertAll(
        () -> assertTrue(success),
        () -> assertEquals(3, store.getInventory(product))
    );
}
```

# Exkurs: Mockito

- Mockito ist ein Beispiel eines Mocking-Frameworks.
- Hilfreiche Methoden
  - `mock()`
    - Wird verwendet um einen Mock einer Klasse zu erstellen.
  - `when()` und `thenReturn()`
    - Verhalten bei einem Methodenaufruf beschreiben.
  - `verify()`
    - Überprüfen von einem Methodenaufruf.

# Beispiel Objekte mit Abhängigkeiten (4)

- London Ansatz: Es wird für das Objekt des Interfaces Store ein Mock verwendet.

```
@Test
@DisplayName("purchase some of the available products")
public void purchaseSomeProducts() {
    Customer customer = new Customer();
    Store store = mock(Store.class);
    when(store.hasEnoughInventory(any(), anyInt())).thenReturn(true);
    Product product = new Product("Milk", 129);

    boolean success = customer.purchase(store, product, 7);

    assertTrue(success);
    verify(store, times(1)).sell(product, 7);
}
```

# Quellen

- Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Gerard Meszaros: **xUnit Test Patterns: Refactoring Test Code**, Addison-Wesley, 2007
- Vladimir Khorikov: **Unit Testing: Principles, Practices, and Patterns**, Manning Publications, 2020
- Martin Fowler: **Mocks Aren't Stubs**, besucht am 30.03.2022,  
<http://www.martinfowler.com/articles/mocksArentStubs.html>