

VII. Einführung in die objektorientierte Programmierung (OOP)

- 1. Konzepte der objektorientierten Programmierung**
- 2. Objektorientierung am Beispiel von Java**
- 3. Struktur und Funktionalität von Java-Programmen**

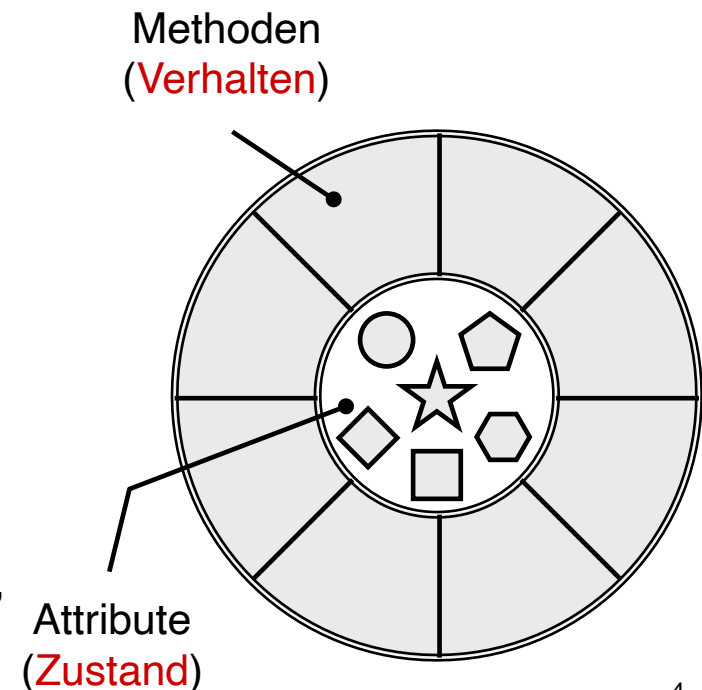
1. Konzepte der objektorientierten Programmierung

- Einführung in die Objektorientierung
- Elementare Grundlagen der Objektorientierung
- Objektorientierung – Analyse, Entwurf, Programmierung

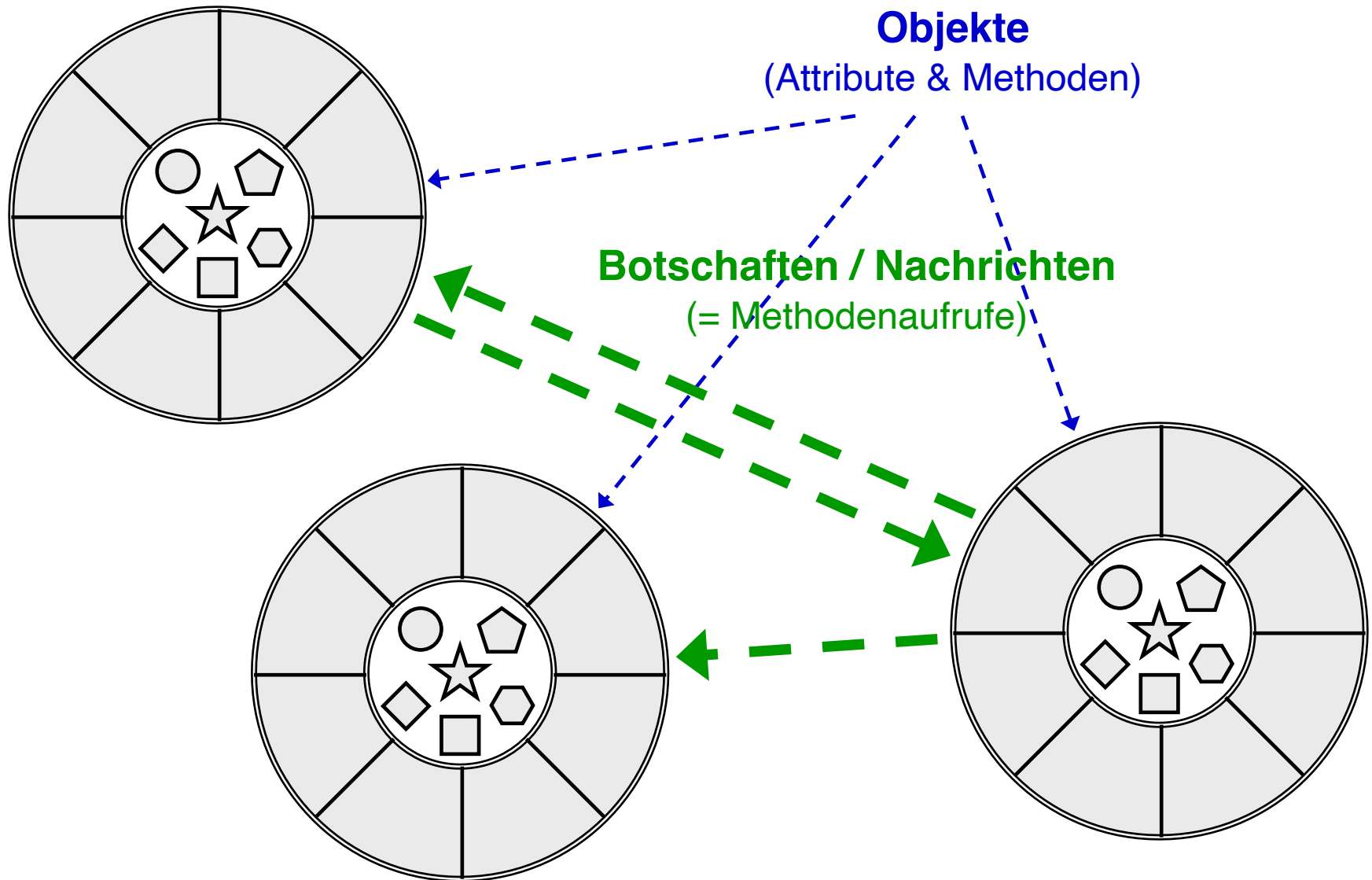
Einführung in die Objektorientierung

Unterschiede zur imperativen Programmierung

- Traditionelle (imperative) Programmiersprachen (z.B. Pascal, MODULA-2, C)
 - Programme sind Folgen von Anweisungen und Aufrufen von Unterprogrammen
 - Es gibt eine Trennung von Daten und Unterprogrammen / Funktionen
 - Der Programmablauf wird zentral gesteuert bzw. koordiniert (Hauptprogramm)
- Paradigmenwechsel durch Objektorientierung
 - Daten (= Attribute) und Funktionen (= Methoden) werden als Einheit betrachtet; sie werden zu **Objekten** zusammengefasst
 - Daten werden nach außen hin verborgen (d.h. „abgeschirmt“) und können bei strenger Auslegung nur über objektspezifische Methoden gelesen bzw. geändert werden (**Kapselung**)
 - Die Kommunikation der Objekte untereinander, mithilfe von (wechselseitigen) Methodenaufrufen, stellt das **Programm** dar



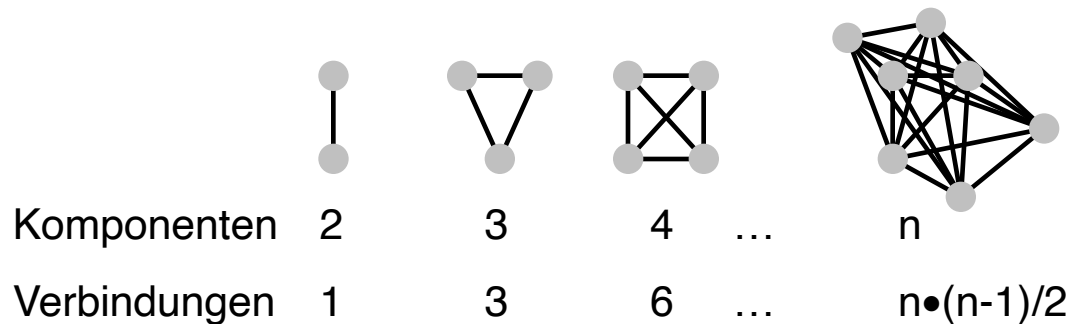
- Objekte kommunizieren über **Nachrichten**



Wozu objektorientierte Programmierung?

Vorteile und allgemeine Zielsetzungen

- Bessere Abbildbarkeit der Realwelt und deren Objekte
- Reduktion der Komplexität von Softwareprodukten
 - Programmsystem: Vielzahl von **Einzelteilen** (Unterprogramme, Daten, Dateien), zwischen denen **verschiedene Beziehungen** bestehen
 - Anzahl der mögl. Verbindungen wächst quadratisch mit der Anzahl der Komponenten



Folge: Ein doppelt so großes Programm verursacht einen 4× so hohen Analyse- und Implementierungsaufwand

Lösungsansatz: Modularisierung \Rightarrow Reduzierung der möglichen Verbindungen

z.B. 16 Komponenten \rightarrow 4 Module à 4 Komponenten

$$= 16 \cdot 15 / 2 = \mathbf{120} \quad = 4 \cdot (4 \cdot 3 / 2) + 4 \cdot 3 / 2 = \mathbf{30}$$

- Wiederverwendung existierender Softwarebausteine
 - Entlastung von Routinetätigkeiten und Konzentration auf anwendungsspezifische Probleme
 - Wiederverwendung beruht auf **Bibliotheken** bekannter Lösungen
 - Wenn bestehende Komponenten nicht genau passen, entsteht Aufwand für die Anpassung an die aktuelle Problemstellung
 1. Neuentwicklung
 2. Anpassung
 3. Erweiterung
 - **Objektorientierung**: **Hierarchien** und **Vererbung** als Technik
- Flexibilität und Kompatibilität
 - Programme sind Sammlungen miteinander kommunizierender Objekte
 - Datenobjekte sind nicht mehr passiv, sondern können selbst Aktionen ausführen
- Unterstützung der Entwicklung von großer Software im Team
- Bessere Wartbarkeit und einfache Erweiterbarkeit von Programmen durch strenge Modularisierung (Aufteilung des Programmcodes auf **gekapselte Objekte**)

Hürden beim Umstieg auf Objektorientierung

- Umdenken bzgl. **Analyse**, **Entwurf** und **Umsetzung** nötig
- Höhere Anforderungen an die **Rechenkapazität**
- Umfangreiche **Klassenbibliotheken** erhöhen zunächst die Komplexität und erfordern mehr Einarbeitung

Bsp.:

```
// In Java ein Wort auf dem Bildschirm anzeigen
public class Output {

    public static void main(String[] args) {
        System.out.println("Hallo ...");
    } // end main

} // end class Output
```

Python:

```
% cat helloworld.py
print("HelloWorld")
% python helloworld.py
HelloWorld
```

Hauptziel der Objektorientierung

- Bessere **Abbildung der Realität** durch objektorientiertes Modell
- Die Programmelemente korrespondieren mit **realen Objekten aus Alltagssituationen**



**Objekt
Haus**

Realität



Abstraktion

Modell

Elementare Grundlagen der Objektorientierung

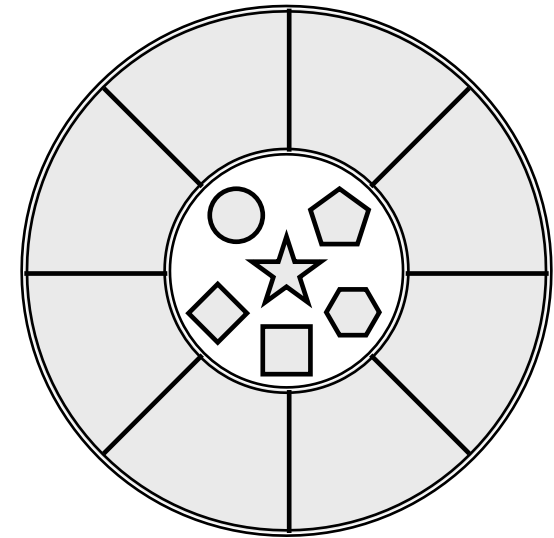
Grundelemente der objektorientierten Programmierung

- Eine objektorientierte Sprache (hier Java) verwendet grundlegende Konzepte und Elemente die mit der realen Welt in Beziehung stehen und deren Begrifflichkeit an Situationen der Realwelt anlehnt
- **Elemente** bzw. Begriffe:
 - Objekte
 - Klassen
 - Instanzen
 - Attribute
 - Methoden
 - Nachrichten

Objekte

Konzept

- **Objekte** repräsentieren Dinge der realen Welt
- Objekte besitzen ...
 - **Eigenschaften** (**Attribute**), die den **Objektzustand** beschreiben
 - **Funktionen** (**Methoden**) für die Manipulation des Objektzustands und damit der Charakterisierung des **Verhaltens** des Objekts – hierzu empfangen und senden Objekte sogenannte **Botschaften**



Objekte vereinen somit Daten und Methoden zum Lesen / Ändern von Daten

- Ein Objekt wird charakterisiert durch:
 - **Objekt-Zustand**: Aktuelle Werte aller Attribute des Objekts
 - **Objekt-Verhalten**: Was machen die Methoden des Objekts als Reaktion auf eintreffende Botschaften?
 - **Objekt-Identität**: Eindeutige Identifikation des Objekts

➡ Möglichkeit der
**graphischen
Darstellung**

Beispiele für Realwelt-Objekte

Autos:

Zustand

Auto
<ul style="list-style-type: none"> • Kennzeichen • Aktuelle Lenkrichtung • Aktueller Gang • Aktuelle Geschwindigkeit
<ul style="list-style-type: none"> • Lenkrichtung ändern • Gang ändern (schalten) • Bremsen • Beschleunigen

Verhalten (Aktionen)

Nachtisch- lampen:

Nachttischlampe
<ul style="list-style-type: none"> • Eingeschaltet • Ausgeschaltet
<ul style="list-style-type: none"> • Einschalten • Ausschalten

Fahrräder:

Fahrrad
<ul style="list-style-type: none"> • Seriennummer • Aktuelle Lenkrichtung • Aktueller Gang • Aktuelle Geschwindigkeit
<ul style="list-style-type: none"> • Lenkrichtung ändern • Bremsen • Beschleunigen

Radios:

Radio
<ul style="list-style-type: none"> • Eingeschaltet • Ausgeschaltet • Aktuelle Lautstärke • Aktueller Sender
<ul style="list-style-type: none"> • Einschalten • Ausschalten • Lautstärke erhöhen • Lautstärke verringern • Sender suchen • Feinabstimmen (<i>tuning</i>)

Softwareobjekte

- Software-Objekte sind konzeptuell ähnlich zu Objekten der Realwelt
- Objekte speichern ihren **Zustand** in **Attributen** (*attributes, fields*);
Objekte zeigen ihr **Verhalten** durch **Methoden** (Funktionen), die
 - auf dem internen Zustand eines Objekts operieren und
 - als primärer Mechanismus der Objekt-zu-Objekt Kommunikation dienen
- Konzept der **Datenkapselung**: Der interne Zustand eines Objekts wird verborgen und die Interaktionen werden über die Methoden des Objekts realisiert
- Vorteile für die Softwareentwicklung
 - **Modularität**: Quellcode für ein Objekt kann geschrieben und unabhängig von anderen Objekten gehalten werden
 - **Informationsverdeckung** (*information hiding*): Durch Interaktion ausschließlich mittels der Methoden werden die Details der Implementierung von außen verborgen
 - **Wiederverwendung** von Programmcode (*bottom-up* Entwurf): Ein bereits existierendes Objekt kann in einem eigenen, neuen Programm genutzt werden
 - **Komposition, Robustheit und Wartbarkeit**: Ist ein Objekt fehlerhaft, so kann es einfach ausgetauscht und durch ein anderes Objekt ersetzt werden

Klassen

Konzept

- **Objekte mit gleicher Struktur** (= Attributen + Methoden) werden zu **Klassen** zusammengefasst;

Klassen stellen eine Art **Schablone** dar, die die Vorlage für konkrete Objekte darstellen

Bsp.: **Objekt** `Client` mit

- **Attributen** `name`, `address`, `bankAccount`, ...
- Die Menge aller Kunden eines Unternehmens bildet eine **Klasse**
- Auf Elemente dieser Klasse [= Objekte; Instanzen] anwendbare **Methoden**: `makeNewClient`, `displayClientState`, ...

- **Klassen** sind demnach **abstrakte Konstrukte**, aus denen **konkrete Objekte** abgeleitet werden
- Wir werden später weiterführende Konzepte zur (hierarchischen) Organisation von Klassen mit ähnlicher Struktur kennen lernen (*Generalisierung* und *Spezialisierung*)

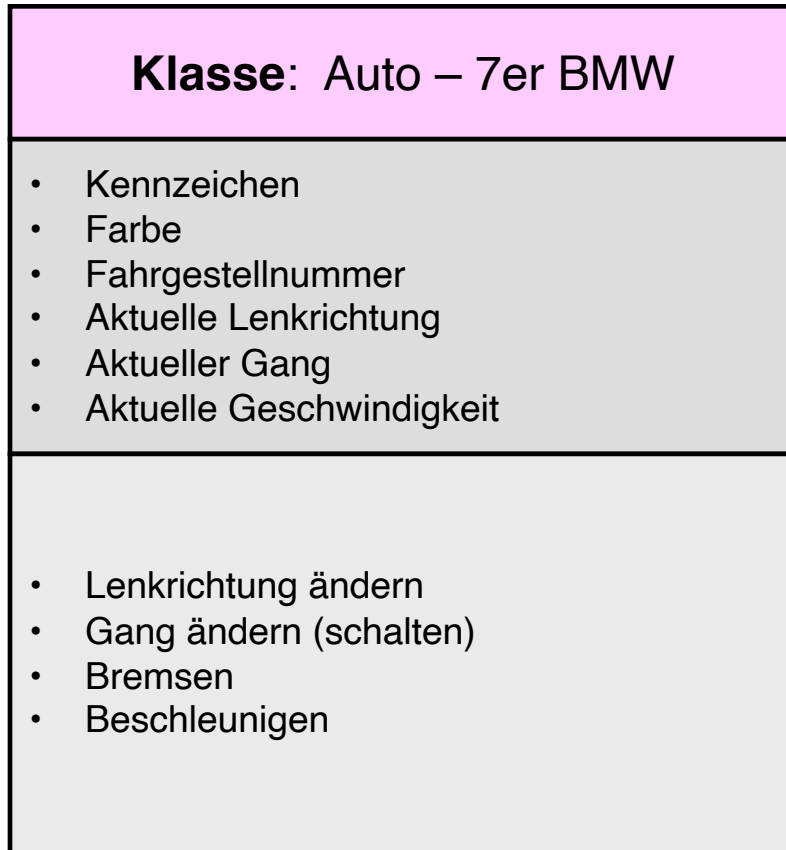
Instanz

Einordnung

- Eine **Klasse** bildet die **Schablone für mehrere gleichartige Objekte**
 - Sie legt die grundsätzlichen **Eigenschaften** und das **Verhalten** ihrer Objekte fest
 - Diese **Eigenschaften** werden nur **abstrakt** angelegt, d.h. die Klasse definiert, welche Attribute ihre Objekte haben, aber **nicht** wie diese **konkret** ausgeprägt sind!
 - Jedes **Objekt** füllt die durch die Klasse definierten Eigenschaften **individuell** aus, d.h. jedes Objekt hat **konkrete Ausprägungen** dieser Eigenschaften
- Ein Objekt, das nach diesen Maßgaben aus einer Klasse erzeugt wird, heißt **Instanz dieser Klasse**
- Auf programmiersprachlicher Ebene werden Instanzen auch **Laufzeitobjekte** genannt:
 - Es handelt sich um Objekte im Hauptspeicher des Rechners, welche eine **Ausprägung der Klasse** darstellen und **konkret verändert** werden können
 - Nur **ein Laufzeitobjekt kann seine Zustände ändern**; wir werden später noch sehen, wie wir solche Instanzen persistent machen (z.B. auf Festplatte speichern) können

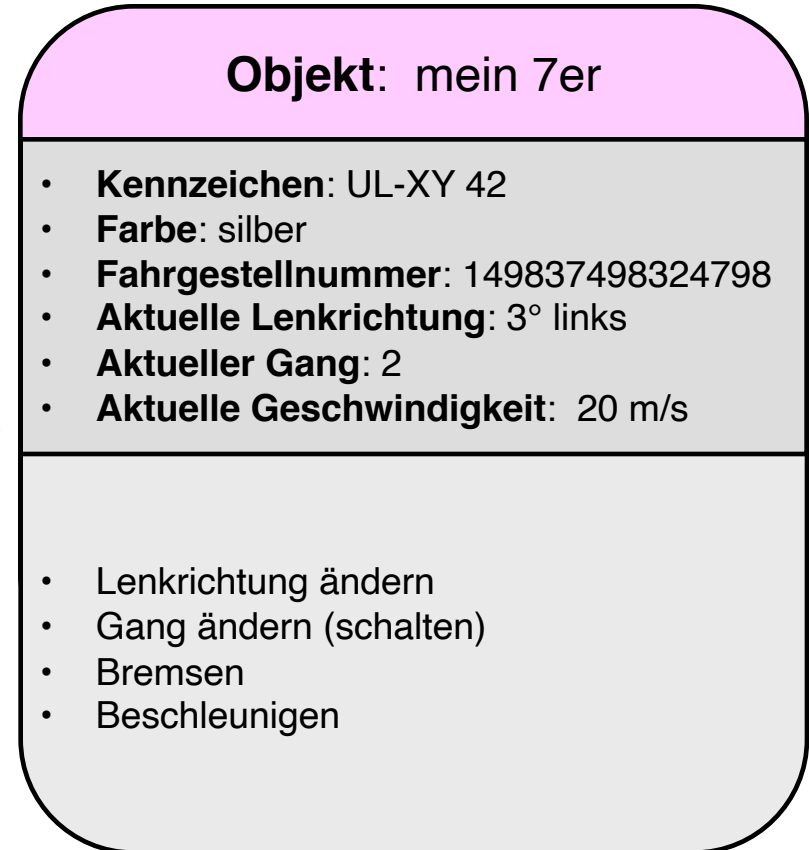
Beispiel zu Klassen und Instanzen

Schablone (abstrakt)



Instanziierung

Instanz (konkret)



Attribute

- Attribute stellen **Eigenschaften von Objekten** dar

Bsp.: Das Attribut `alter` ist eine Eigenschaft des Objektes `Person`

- Die konkreten Werte, d.h. Ausprägungen, von Attributen definieren den **aktuellen Objektzustand**

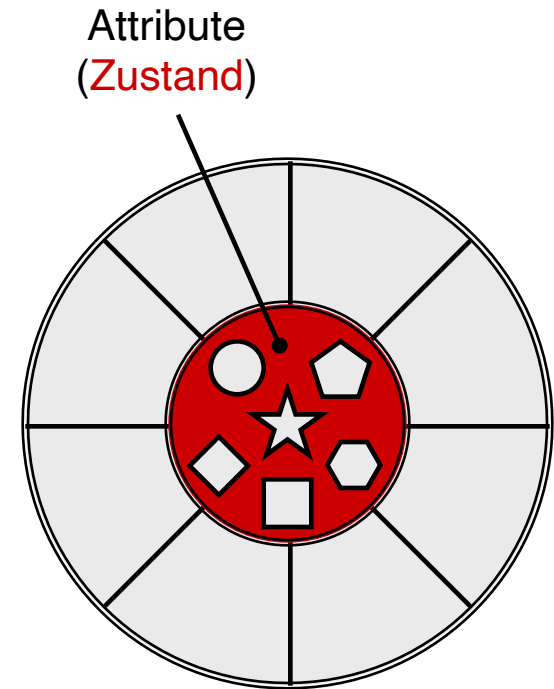
Bsp.: Der aktuelle Zustand eines Objekts `Person` wird durch sein `alter` sowie weitere Attribute beschrieben (`gewicht`, `groesse`, usw.)

- Ein Attribut besitzt (i) den **Wert eines primitiven Datentyps** oder (ii) repräsentiert ein **Objekt**

- Attribute deren Wert einem **bestimmten primitiven Datentyp** genügt:
Zahl, Zeichen, Wahrheitswert, usw.

- Attribute, die **Objekte** repräsentieren (Beispiele):

Objekt `Auto` besitzt ein Attribut `Reifen`;
`Reifen` ist eigenes Objekt mit Attributen `farbe`, `felgentyp` usw.

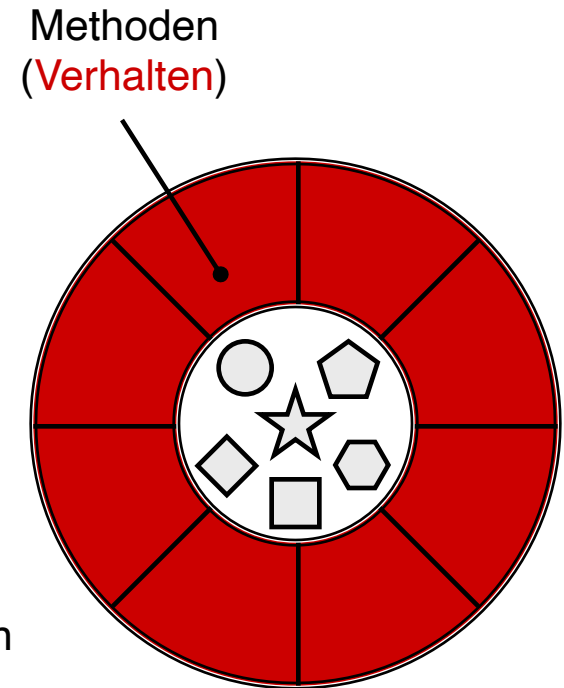


Methoden

- Durch Methoden (**Funktionen** in Programmiersprachen – mit und ohne Ergebniswerte) kann der **Zustand eines Objektes verändert** werden
- Methoden stellen somit **Operationen auf Objekten** dar, die beispielsweise die Attribute verändern

Bsp.: Der **aktuelle Zustand** eines Objekts `Person` mit dem Attribut `alter` kann mit einer **Methode** `incrAlter` (Erhöhung des Alters) verändert werden

- In der **(strengen) Objektorientierung** ist der **Zugriff auf die Attribute eines Objekts nur über die objekteigenen Methoden** möglich (Informationsverdeckung, *information hiding*)



Nachrichten (*messages*)

Eigenschaften

- Nachrichten werden auch Botschaften genannt
- Objekte treten miteinander in Beziehung (kommunizieren miteinander), indem sie Nachrichten (Botschaften) austauschen
- Diese Botschaften repräsentieren Methoden-Aufrufe (Aufrufe von Unterprogrammen) und die hieraus resultierenden Ergebnisse
- Eine Nachricht (Botschaft) enthält somit Eingabeparameter bzw. Rückgabewerte gerufener Methoden;

die aufgerufene Methode des Objekts übernimmt die Eingabeparameter (aktuelle Parameter des Unterprogramm), führt eine Verarbeitung durch und liefert ggf. das Ergebnis an das aufrufende Objekt zurück (vgl. **Teil VI** mit den Parameterübergabe-Mechanismen bei Unterprogrammen bzw. Funktionen)

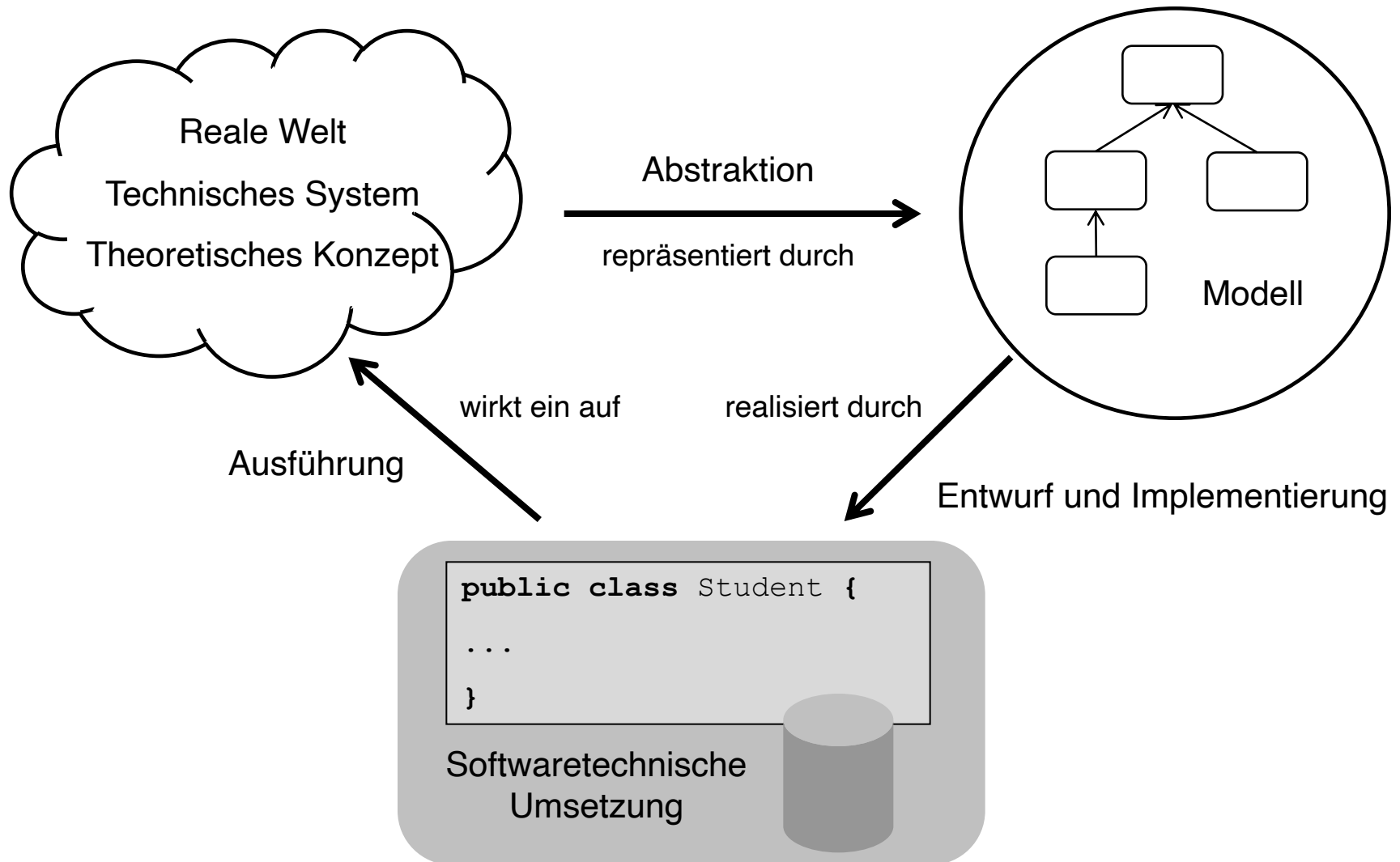
Objektorientierung – Analyse, Entwurf, Programmierung

Vom Realweltproblem zur softwaretechnischen Umsetzung

Vorüberlegungen

- **Objektorientierung** soll helfen, Ausschnitte der Realwelt besser durch eine Programmiersprache abzubilden
- Meist geht der programmiersprachlichen Umsetzung eine **Abstraktion** durch Bildung von **Modellen** voraus
 - **Modelle** (meist in graphischer Form) werden benötigt, da Programmiersprachen meist keine ausreichenden **Abstraktionsebenen** bieten – hauptsächlich liegt das am Fehlen von sich einander ergänzenden **Sichten**
 - Man behilft sich daher mit dem **Abbilden der Realwelt durch Modelle**, welche dann als Grundlage für die programmiersprachliche Umsetzung dienen

Realweltprobleme, ihre Abstraktion und softwaretechnische Realisierung

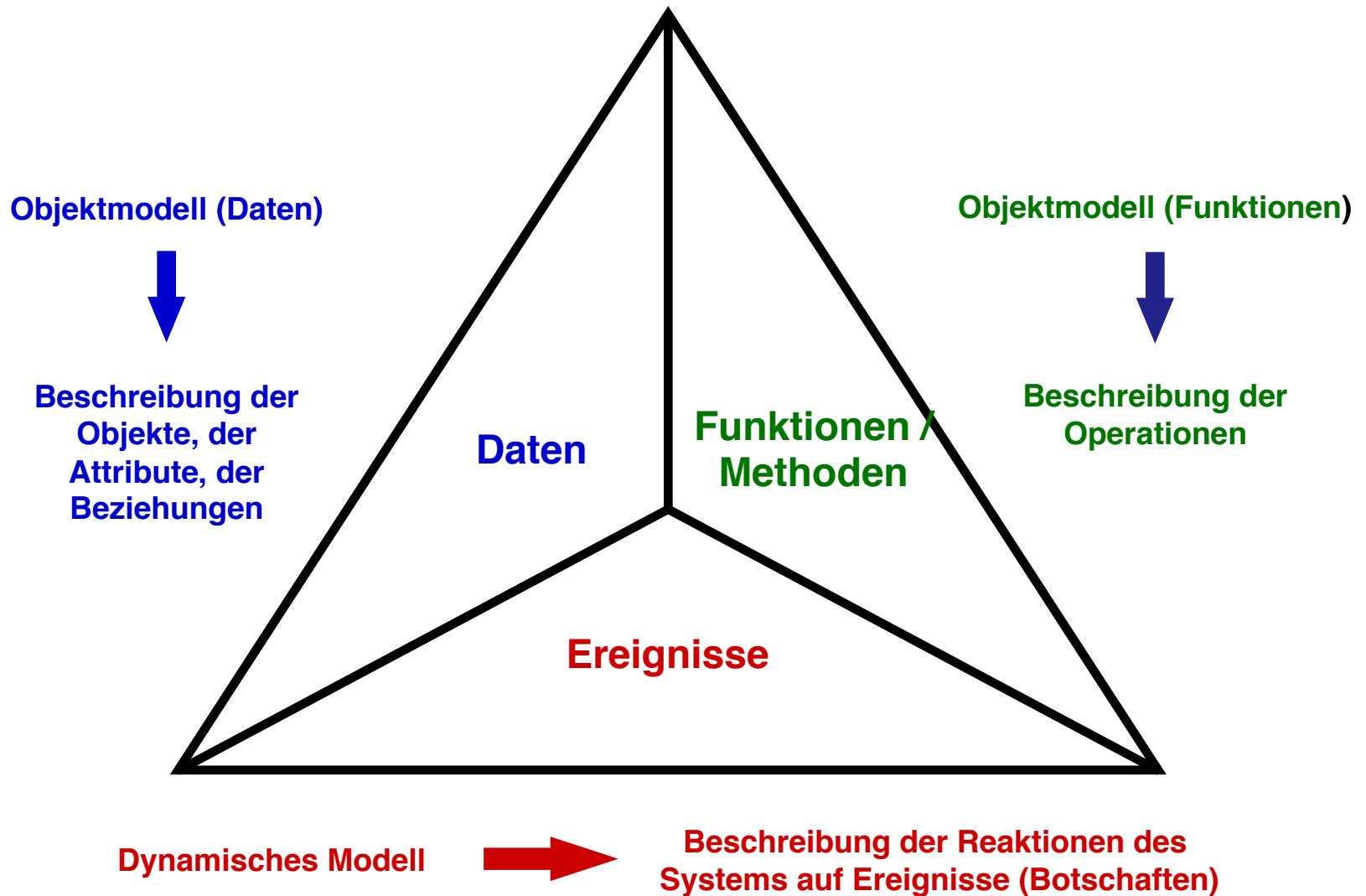


Objektorientierter Softwareentwurf

Einordnung

- **Herausforderung:** Beobachten, Identifizieren und Beschreiben von Objekten der Realwelt, so dass diese sinnvoll durch Klassen implementiert werden können (vgl. die Skizze der Abstraktion und Charakterisierung von Objekten, ihrer Eigenschaften und Verhaltensweisen)
- **Methodik: Objektorientierter Softwareentwurf** bestehend aus drei Phasen
 - **Phase 1: Objektorientierte Analyse (OOA)**
Grundlegende Analyse der für das Problem relevanten Eigenschaften und Verhaltensweisen
 - **Phase 2: Objektorientierter Entwurf (objektorientiertes *Design*, OOD)**
Modellierung der identifizierten Eigenschaften (Attribute) und Verhalten (Methoden)
 - **Phase 3: Objektorientierte Programmierung (OOP)**
Umsetzung der in dem Modell (aus dem OOD) spezifizierten Komponenten in einer Programmiersprache; wir verwenden hier Java und werden die ersten „Gehversuche“ in diesem Kapitel unternehmen ...

Objektorientierte Analyse (OOA)



Objektorientierter Entwurf (objektorientiertes *Design*, OOD)

- **Modellierung verschiedener Perspektiven** des zu realisierenden Software-Systems (z.B. Struktur und Verhalten von Objekten, Interaktionen)
- Für die Modellierung dieser Perspektiven verwendet man verschiedenartige Diagramme (die *Unified Modeling Language*, UML)

Hinweis: Die bisher gezeigten Darstellungen der Klassen und Objekte entspricht z.B. der Darstellung in UML-Klassen- bzw. UML-Objektdiagrammen (Bsp.: Klassendiagramm für Fahrräder – ohne Werte für Attribute, die bei Instanzen hinzu kommen)
- UML umfasst zahlreiche **wichtige Modellierungstechniken** für OOD (UML stellt einen universellen Rahmen zur Beschreibung basierend auf Zustandsdiagrammen dar; ein kurzer Überblick folgt in **Teil X**)

Fahrrad (Bicycle)
<ul style="list-style-type: none"> • Seriennummer • Farbe • Aktuelle Lenkrichtung • Aktueller Gang • Aktuelle Geschwindigkeit
<ul style="list-style-type: none"> • Lenkrichtung ändern • Bremsen • Beschleunigen

Objektorientierte Programmierung (OOP)

- Ausgehend von den Ergebnissen des objektorientierten Entwurfs (z.B. für ein Fahrrad-Objekt) müssen für die **konkrete Implementierung** eine Reihe von Fragen beantwortet werden
- Grundsätzliche **Fragen** sind:
 - *Wie definiere ich eine **neue Klasse** innerhalb eines objektorientierten Programms (hier Java)?*
 - *Wie definiere ich die verschiedenen **Attribute**?*
 - *Wie werden **Methoden** und ihre Parameter definiert?*
 - *Wie erzeuge ich **Laufzeitobjekte** (d.h. **Instanzen**) einer neu definierten Klasse?*
- Im anschließenden Abschnitt werden entlang eines ausgewählten Beispiels zur Modellierung von Klassen (Fahrräder) die Konzepte in Java-Programmen umgesetzt und erste allgemeine Betrachtung zur Verwaltung von Objekten durchgeführt

Fahrrad (Bicycle)
<ul style="list-style-type: none"> • Seriennummer • Farbe • Aktuelle Lenkrichtung • Aktueller Gang • Aktuelle Geschwindigkeit
<ul style="list-style-type: none"> • Lenkrichtung ändern • Bremsen • Beschleunigen

Konzepte der Objektorientierung

Einordnung

- Beim objektorientierten Entwurf (Ergebnis der objektorientierten Analyse) werden die **Attribute** sowie die **Methoden** (Zustandsänderung) identifiziert
- Wesentliche **Strukturierungsmechanismen** sind:
 - **Abstraktion**
 - **Kapselung**

Hinweis: Diese Konzepte werden **in diesem Kapitel** behandelt

Weitere Konzepte sind:

- **Vererbung**
- **Dynamische Bindung und Polymorphie**
- **Überladen und Überdeckung von Methoden**

Hinweis: Diese Konzepte werden in **Kapitel X** behandelt

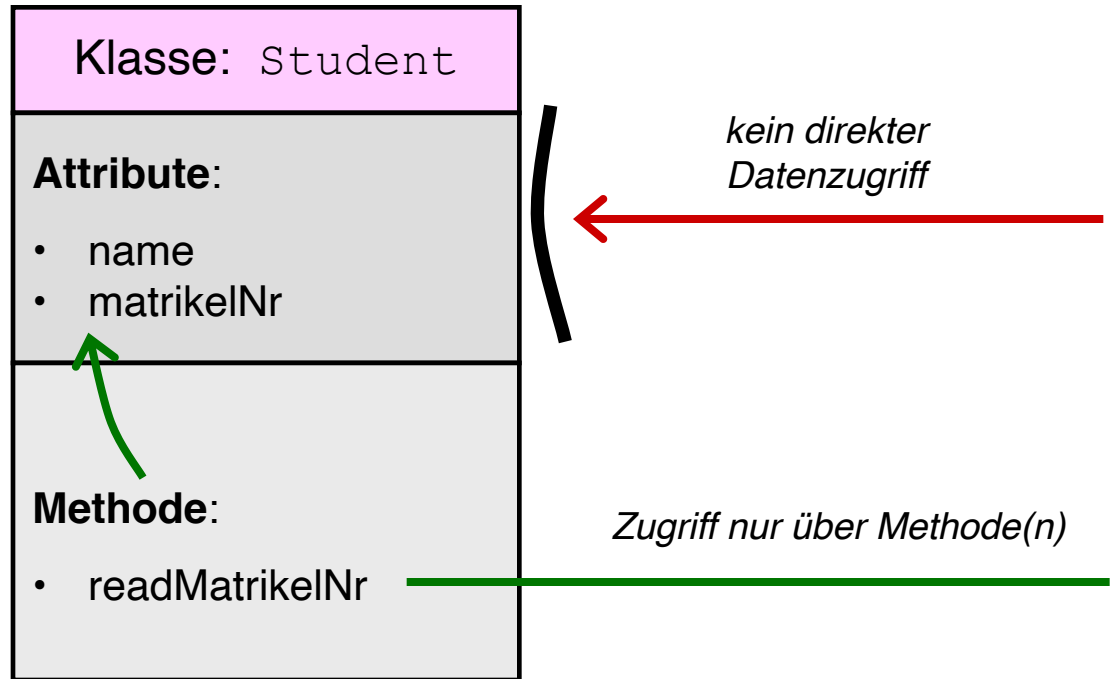
- Objektorientierte Sprachen, wie z.B. Java, bieten verschiedene Mechanismen an, mit denen diese Konzepte umgesetzt werden können

Abstraktion

- Ein **fundamentales Prinzip der Informatik** im Allgemeinen sowie der Objektorientierung im Besonderen ist die **Abstraktion**, d.h. die Abbildung eines Ausschnitts der Realwelt in ein Modell, in dem nur die wichtigen Aspekte des realen Gegenstands berücksichtigt werden
- **Modellierung** eines Ausschnitts der Realwelt:
 - Beschränkung der Modellbildung auf die für die jeweilige Fragestellung oder Anwendung relevanten Elemente unter Vernachlässigung anderer Details der realen Objekte
 - Es wird also nicht versucht, ein vollständiges Modell des untersuchten Ausschnitts der Realwelt aufzubauen, sondern es werden lediglich die tatsächlich benötigten Teilaspekte modelliert (entspricht der klassischen Vorgehensweise in den Naturwissenschaften)
- **Nutzen** (für den Softwareentwurf): Reduzierte Komplexität und beschleunigte Modellbildung; dadurch auch weniger Fehler bei der Umsetzung

Kapselung

- Zusammenfassung von **Name**, **Zustand** und **Verhalten** eines Objekts (über den Namen der Klasse, Attribute, Methode(n))
- Daten bzw. Attribute eines Objekts können **nur über dessen Methoden** verändert werden
- Vorteile:
 - Durch eine **Kapselung** wird eine ungewollte Datenmanipulation verhindert; Programme laufen dadurch sicherer und stabiler
 - Der innere Aufbau, die Struktur, und die **Realisierung bleibt vor der äußeren Umgebung verborgen** (*information hiding*) und das Repertoire der Operationen (über die Methoden) bleibt in der Klasse verankert; Programme können einfacher gewartet werden, da Implementierungsdetails nicht sichtbar sind und die Funktionalität nur über Schnittstellen definiert ist (Bsp.: neue Methode `changeMatrNr`)



2. Objektorientierung am Beispiel von Java

- Einführendes Beispiel
- Grundlegendes über Objekte
- Referenz-Variablen und Zugriff auf Objekte
- Statische und nicht-statische Methoden und Attribute
- Objekte und *Arrays*
- Konstruktoren, Objektinitialisierung und Freigabe

Einführendes Beispiel

Einordnung: Entwurf einer Fahrrad-Klasse

Objektorientierte Analyse (OOA) für ein Fahrrad – Abstraktion

■ Daten

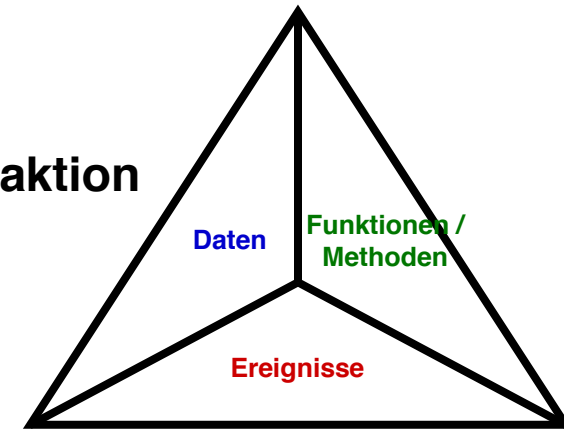
- Es sollen Fahrräder modelliert werden
- Dabei ist für die **Verhaltensmodellierung** die Trittfrequenz, die Geschwindigkeit und der Gang wichtig
- Eigenschaften, wie Farbe, Sattelhöhe, etc. sind für die Fragestellung unwichtig – und werden daher ignoriert (Abstraktionsschritt)

■ Methoden

- Die Eigenschaften sollen initialisiert werden können
- Die relevanten Daten (**Attribute**) sollen gezielt verändert werden können, da sie den **Zustand eines aktuellen Objektes** definieren – der sich ändern darf
- Von einer Klasse sollen **konkrete Objekte erzeugt** werden können

■ Dynamisches Modell

- Es gibt nur eine Klasse, von der Objekte als Instanzen gebildet werden
- Ein Objekt kann seinen Zustand durch äußere Einwirkung ändern und der Zustand kann angezeigt werden



Objektorientierter Entwurf (OOD)

- Einige grundlegende Fragen ...

- *Wie soll die Klasse für das Fahrrad-Objekt heißen (Benennung)?*

Wir nennen sie **Bicycle** (engl. Fahrrad)! Nameskonventionen gibt es wenige, allerdings sollte der Name einer Klasse kurz und prägnant sein sowie die durch die Klasse repräsentierten Objekte gut charakterisieren. Oft wird für Klassennamen in Java der sog. *CamelCase* verwendet.

- *Welche Attribute sollen für das Objekt der Klasse Bicycle verwaltet werden?*

Das hängt von der Aufgabe und Zielsetzung ab, also was mit den Objekten gemacht werden soll. Aus der objektorientierten Analyse hat sich ergeben, dass die **Trittfrequenz** (*cadence*), **Geschwindigkeit** (*speed*) und der **Gang** (*gear*) hier relevant sind

- *Welche Methoden sollen für ein Objekt der Klasse Bicycle verfügbar sein?*

Die objektorientierte Analyse hat Methoden zur Initialisierung, zur Zustandsänderung und der Zustandsanzeige vorgeschlagen. Für den Zugriff auf Attribute kommen oft sogenannte *Getter und Setter Methoden* zum Einsatz, d.h.

Attribut X → **getX()**; **setX(value)**;

- Die Klasse *Bicycle* kann in UML graphisch in Form eines Diagramms dargestellt werden (siehe **Teil X**)

Name

Bicycle

Attribute

cadence (Trittfrequenz)
speed (Geschwindigkeit)
gear (Gang)

Methoden

initBicycle
changeCadence
changeGear
speedUp
slowDown
printState

(initialisiere Attribute)
(ändere Trittfrequenz)
(ändere Gang)
(beschleunige)
(bremse)
(gib Zustand aus)

Objektorientierte Programmierung (OOP)

Es sind eine Reihe implementierungsrelevanter Fragen zu diskutieren ...

- *Welche Datentypen wählen wir sinnvoller Weise für die Attribute?*
 - Die Trittfrequenz (*cadence*) und der Gang (*gear*) seien diskrete Werte, diese können vom Datentyp `int` sein
 - Die Geschwindigkeit (*speed*) kann eine Gleitkommazahl sein, daher kann man sie als `float` repräsentieren

- *Welche Ergebnisse sollen die Methoden liefern?*

Das dynamische Modell sieht hier nur die Steuerung und Veränderung des Zustands eines Objektes vor; der aktuelle Zustand eines Objektes soll nur angezeigt, nicht jedoch geliefert werden; daher sind die Methoden (Funktionen) ohne Rückgabewert (`void`)

- *Welche Parameter sollen die Methoden (Funktionen) erhalten?*
 - Die Methode zur Initialisierung des Zustands weist den Attributen einen Anfangswert zu, die Parameter müssen mit deren Datentyp kompatibel sein
 - Die Methoden zur Veränderung des Zustands benötigen Parameter, die mit dem jeweiligen Datentyp des Attributs korrespondieren
 - Die Zustandsanzeige benötigt keine Parameter – sie liest die aktuellen Attributwerte

Implementierung der Klasse *Bicycle* in Java

Schematische Vorgehensweise

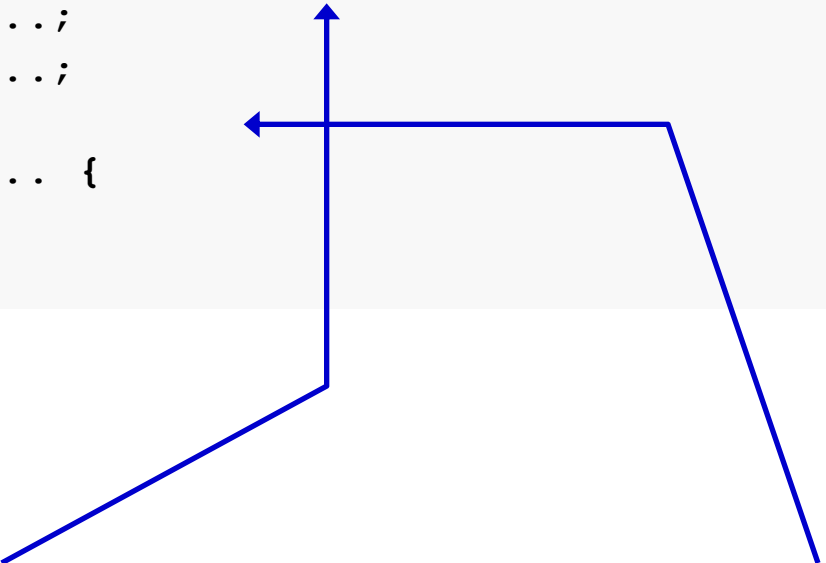
- Generell kann man in vier Schritten vorgehen:
 1. **Klasse erstellen**
 2. **Attribute**, d.h. Eigenschaften, in Form von Variablen einfügen
 3. **Konstruktor** einfügen
(spezielle Anweisung, mit der eine Instanz der Klasse erzeugt wird)
 4. **Methoden** einfügen, die in Form von Funktionen definiert sind, eine (möglicherweise leere) Parameterliste besitzen und einen (möglicherweise leeren, `void`) Rückgabewert haben

- Für die anschließende Verwendung der Klasse wird noch eine Klasse mit dem Testprogramm benötigt, in dem die `main()` Methode spezifiziert ist (im Anschluss)

Eine Klasse erstellen

Festlegung des Rahmens ...

```
public class Bicycle {  
    ...;  
    ...;  
  
    ... {  
    }  
}
```



Name der Klasse

(nochmals zur Erinnerung: Die **Datei** mit dieser Klassendefinition/-implementierung **muss** `Bicycle.java` heißen)

Hier folgen ...

Attribute, **Konstruktor(en)** und **Methoden** in einem Block

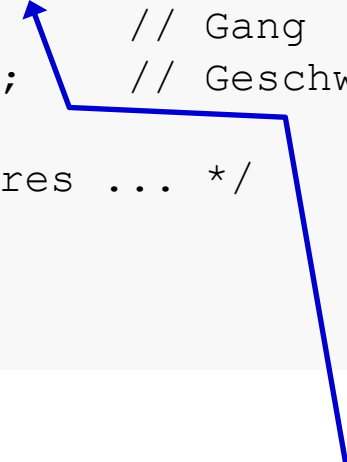
Implementierung der Klasse *Bicycle*:

1. **Klasse erstellen**
2. **Attribute / Eigenschaften einfügen**
3. **Konstruktor(en) einfügen**
4. **Methoden einfügen**

Attribute (Eigenschaften) einfügen

Attribute werden für die Definition des Zustands eines Objekts verwendet

```
public class Bicycle {  
    /* -- Attribute */  
    int cadence, // Trittfrequenz  
    gear; // Gang  
    float speed; // Geschwindigkeit  
  
    /* -- weiteres ... */  
    ...  
    ...  
}
```



Implementierung der Klasse *Bicycle*:

1. Klasse erstellen
2. **Attribute / Eigenschaften einfügen**
3. Konstruktor(en) einfügen
4. Methoden einfügen

Attribute werden definiert nach dem Muster

[<Modifizierer>] <Datentyp> <Name>

- so wie bisher bei der Deklaration von Daten einfacher (primitiver) Datentypen
- Modifizierer ändern die Sichtbarkeit z.B. public

Konstruktor einfügen

- **Wichtig:** Ein **Konstruktor** (mit **demselben Namen wie die Klasse**) dient zur **Erzeugung einer neuen Instanz der Klasse** (Details folgen im Anschluss)

```
public class Bicycle {
    /* -- Attribute */
    ...
    Attribute auf der vorherigen Folie

    /* -- Konstruktor */
    Bicycle(int _cadence,
           float _speed,
           int _gear) {
        cadence = _cadence; // Initialisieren
        speed    = _speed;  // der Attribute
        gear     = _gear;   // des Objekts
    }
    ...
}
```

Implementierung der Klasse *Bicycle*:

1. Klasse erstellen
2. Attribute / Eigenschaften einfügen
3. **Konstruktor(en) einfügen**
4. Methoden einfügen

Wichtig: Ein **Konstruktor** ist **keine Methode** (er besitzt keinen Ergebniswert und kann nur mit **new** aufgerufen werden)

Parameter des Konstruktors (werden häufig durch '_' besonders gekennzeichnet)

- Zur Erzeugung von Objekten (Instanzen einer Klasse, Laufzeitobjekte)
 - Die Erzeugung eines Objektes erfolgt durch den **Operator new**
 - Im **Arbeitsspeicher** (*heap*) wird Speicherplatz für das Objekt (und seine Attribute mit deren Datentypen) angefordert und allokiert (vgl. die Erzeugung von *Arrays*, **Teil V**)
 - Zur Erzeugung wird ein Konstruktor aufgerufen und ausgeführt (ein Konstruktor entspricht von der Form seiner Deklaration einer speziellen Methode, die **denselben Namen wie die Klasse** tragen muss; die Definition eines Konstruktors sieht **keinen Ergebniswert** vor)
 - Den **Objektattributen** werden Werte zugeordnet, die den **Initialzustand** des Objektes definieren
 - Entweder *Default*-Werte (Zahlen: 0, Zeiger: `null`, etc.) bei Aufruf des *Default*-Konstruktors der Klasse oder
 - Festgelegt durch einen speziellen Konstruktor, der in der Klasse definiert wurde (s.o.)
- Hinweis: Zu Konstruktoren und der Objekt-Initialisierung gibt es nachfolgend einen eigenen Abschnitt!

Implementierung der Klasse *Bicycle*:

1. **Klasse erstellen**
2. **Attribute / Eigenschaften einfügen**
3. **Konstruktor(en) einfügen**
4. **Methoden einfügen**

Methoden einfügen

Mithilfe von **Methoden** werden die **Zustände eines Objektes** (repräsentiert durch die Attribute) **verändert oder gelesen**

```
public class Bicycle {
    /* -- Attribute */
    ...
    /* -- Konstruktor */
    ...
    /* -- Methoden */
    void changeCadence(int newValue) { ... }
    void changeGear(int newValue) { ... }
    void speedUp(float incrValue) { ... }
    void slowDown(float decrValue) { ... }
    void printState() { ... }
}
```

Implementierung der Klasse *Bicycle*:

1. Klasse erstellen
2. Attribute / Eigenschaften einfügen
3. Konstruktor(en) einfügen
4. **Methoden einfügen**

Die Methoden in unserem Beispiel hier liefern allesamt keinen **Ergebniswert** (**void**) (entsprechend der Festlegung im Entwurf der Implementierungsdetails, S.34)

Den Methoden werden **Parameter** übergeben oder die Liste ist leer

Klasse *Bicycle* für Fahrrad-Objekte (Demo: [Bicycle.java](#) sowie [BicycleDemo.java](#))

```

class Bicycle {
    /* -- Attribute */
    int    cadence; // Trittfrequenz
    double speed;   // Geschwindigkeit
    int    gear;    // Gang

    /* -- Konstruktor (mit Initialisierung der Attribute) */
    Bicycle(int _cadence, double _speed, int _gear) {
        cadence = _cadence;
        speed    = _speed;
        gear     = _gear;
    } // end constructor Bicycle

    /* -- Methoden */
    void changeCadence(int newValue) {           // aendere Trittfrequenz
        cadence = newValue;
    } // end changeCadence

    void changeGear(int newValue) {              // aendere Gang
        gear = newValue;
    } // end changeGear

    void speedUp(double incrementValue) {        // erhoehe Geschwindigkeit
        speed = speed + incrementValue;
    } // end speedUp

    void slowDown(double decrementValue) {       // bremse Geschwindigkeit ab
        speed = speed - decrementValue;
    } // end slowDown

    void printState() {                         // zeige aktuellen Zustand an
        System.out.println(" |Cadence: " + cadence +
                           " |Speed:  " + speed +
                           " |Gear:   " + gear);
    } // end printState
} // end class Bicycle

```

Grundlegendes über Objekte

Datentypen, Referenzvariablen und Objektallokation

Datentypen und Variablen

- In Java wird eine **Klasse** als (neuer) **Datentyp** behandelt

Hinweis: Dies entspricht dem Umgang mit einfachen (primitiven) Typen `int`, `boolean`, `double`, `char`, etc. sowie auch `String`, die von Java zwar wie ein primitiver Datentyp zur Verfügung gestellt werden, streng genommen jedoch kein primitiver Typ sind (vgl. auch die verschiedenen Funktionen, die auf `Strings` definiert sind)

- Ein **Klassenname** kann verwendet werden, um den Typ einer
 - (**Zeiger-) Variablen** zu deklarieren, oder
 - eines **formalen Parameters**, oder
 - eines **Rückgabewerts** (`return`) einer Methode

festzulegen

Bsp.: Stark vereinfachte Version einer **Klasse** `Student`, die verwendet werden kann, um Informationen über Studierende abzulegen, die eine Veranstaltung besuchen

```
public class Student {
    String name;    // Name der Studentin
    double test1,
           test2,
           test3;  // Testresultate

    double getAverage() {
        // berechne das durchschnittliche Ergebnis
        return ((test1 + test2 + test3) / 3.0);
    }
}
```

Deklaration einer Referenz- (Zeiger-) Variablen des (neuen) Typs `Student`:

```
Student <Objekt-Name>;
```

Hinweis: Die **Deklaration der Variablen** mit dem Bezeichner `<Objekt-Name>` generiert selbst noch kein Objekt; in Java enthält eine **Variable niemals ein Objekt**, sondern kann **ausschließlich eine Referenz (Zeiger) auf ein Objekt** enthalten (vgl. zur Erinnerung den **Umgang mit Arrays, Teil V**)

Erzeugung von Objekten

- Objekte werden mit dem **Operator new** generiert (vgl. **Teil V** mit der Generierung von *Arrays*)

Bsp.:

```
Student stud;  
  
stud = new Student();
```

Erläuterung:

- Die Anweisung mit der Zuweisung erzeugt ein **neues Objekt** (Laufzeitobjekt), das eine Instanz der Klasse `Student` ist
- Eine **Referenz auf dieses Objekt** im Arbeitsspeicher (*heap*) wird in der Referenz-/Zeiger-Variablen `stud` abgelegt (die Variable hält die notwendige Information, um das Objekt und seine Komponenten im Speicher zu finden; **Dereferenzierung**)
- Wenn die Variable nicht auf ein generiertes Objekt (Instanz einer Klasse) zeigt, dann enthält sie einen Null-Zeiger (`null`)

- Klassenvariable des selben Typs (also derselben Klasse) können einander zugewiesen werden (wie bei einfachen Datentypen)

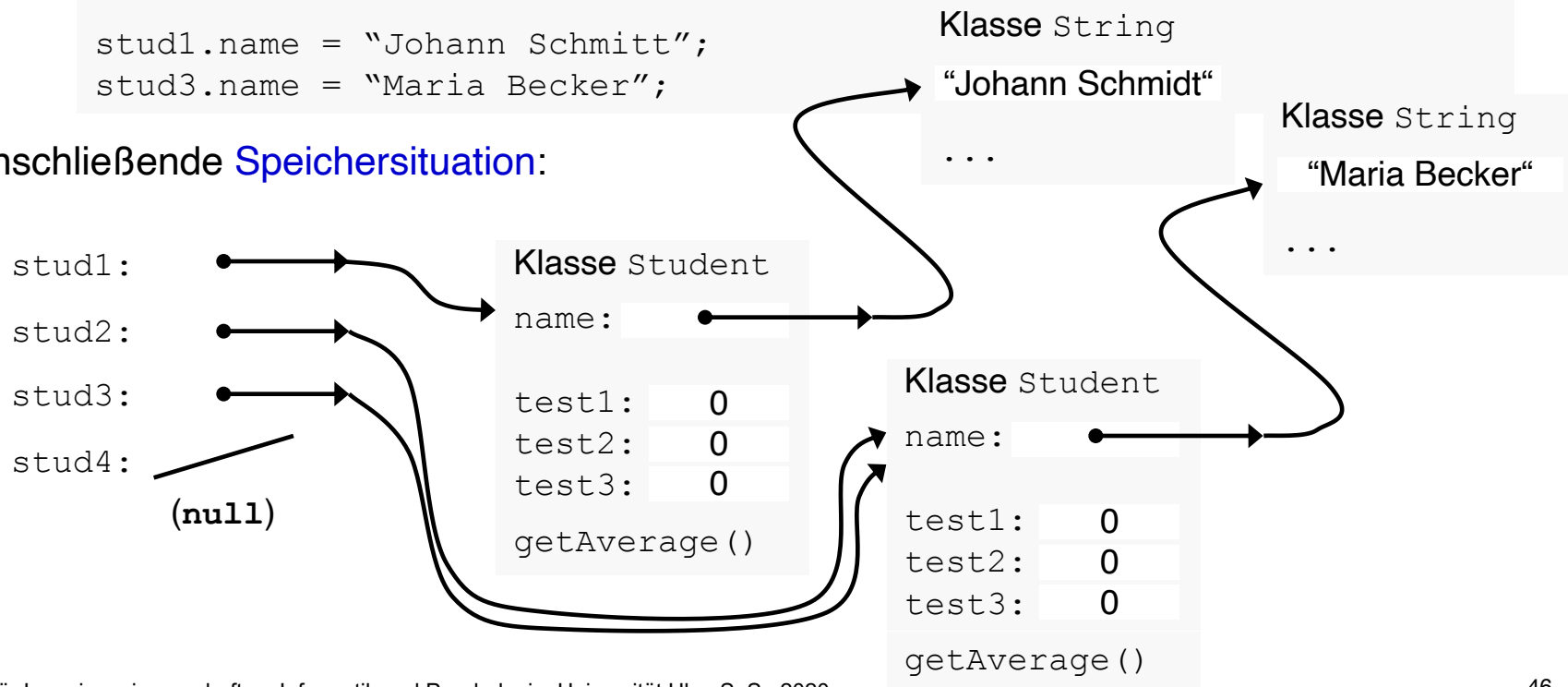
Bsp.: `Student stud1, stud2, stud3, stud4;`

```
stud1 = new Student();
stud2 = new Student();
```

```
stud3 = stud2;
stud4 = null;
```

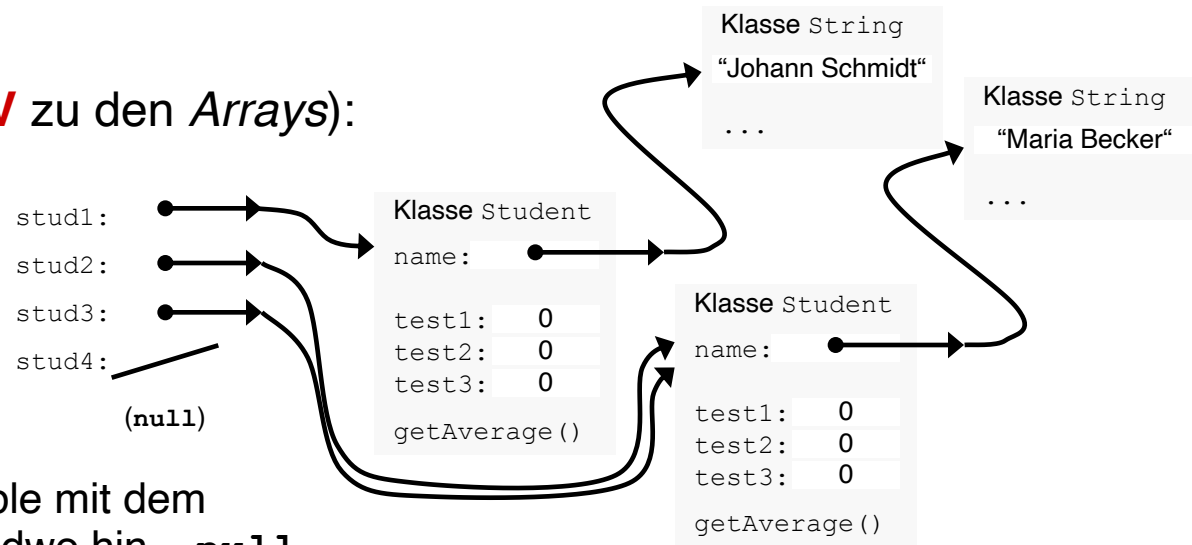
```
stud1.name = "Johann Schmitt";
stud3.name = "Maria Becker";
```

Anschließende **Speichersituation:**



Erläuterungen (vgl. **Teil V** zu den *Arrays*):

- Wenn eine Variable die Referenz auf ein Objekt enthält, wird dies in Form eines Zeigers dargestellt
- Eine (Referenz-) Variable mit dem Wert `null` zeigt nirgendwo hin – `null` bedeutet explizit, dass die Variable momentan auf kein Objekt verweist
- Die Zeiger der Variablen `stud2` und `stud3` referenzieren beide dasselbe Objekt im Speicher
- Zeichenketten (`Strings`) sind eigentlich Objekte, die erzeugt werden – ohne dass wir das selber explizit anweisen müssen ...



Wichtig:

- Es gilt weiterhin (wie bei den *Arrays*): Wenn eine **Zeigervariable** (die auf ein Objekt verweist) an eine andere Zeigervariable zugewiesen wird, dann wird nur die Referenz kopiert; das **referenzierte Objekt** selbst (im Speicher) wird **nicht kopiert**
- Die **Datentypen** bzw. die Klassen der Zeigervariablen, die zugewiesen werden, müssen **identisch** sein

Die Rolle von Methoden

- Die **Methoden** beschreiben zusammen mit den **Attributen** die abstrakten **Eigenschaften von Objekten**
 - Sie ermöglichen es, die Attribute bzw. Eigenschaften von Objekten (Instanzen einer Klasse) zu verändern
 - Durch Aufruf von Methoden erhält man Zugriff auf die Attribute von Instanzen bzw. kann man darauf basierende Berechnungen vornehmen
- Methoden dienen (auch) der **Abstraktion** und sind ein wichtiges Mittel zur **modularen Gliederung** von Programmen
- **Ziel:** **Datenkapselung**, d.h. gekapselter Zugriff auf Objektattribute über Methoden
 - Erhöhung der Beherrschbarkeit der Komplexität beim Programmieren, da die Details der Repräsentation der Attribute von Außen nicht sichtbar sein müssen
 - Weiterhin reduziert ein ausschließlich über Methoden realisierter Zugriff auf den Zustand eines Objekts (in Form seiner Attribute) die Fehlerhäufigkeit erheblich

Zugriff auf die Elemente einer Instanz

- Eine **Instanz** enthält aktuelle Objektkomponenten (**Attribute**), die den Zustand des Objekts definieren; weiterhin sind die **Methoden** verfügbar, die auf die Attribute zugreifen und den Zustand verändern (vgl. S.48)
- Auf die **Komponenten einer Instanz** (Attribute, Methoden) kann mittels eines **Selektors** `'.'` zugegriffen werden

Allgemein: `<Referenz-Variable>.<Komponente>`

Bsp.: Es sei eine **Instanz** der Klasse *Bicycle* generiert worden (Allokation von Speicherplatz mittels eines klassenspezifischen **Konstruktors**)

```
Bicycle bike = new Bicycle(0, 0.0, 1);  
                // Verwendung eines klassenspezifischen  
                // Konstruktors mit Uebergabe von Parametern  
                // zur Initialisierung der Instanz
```

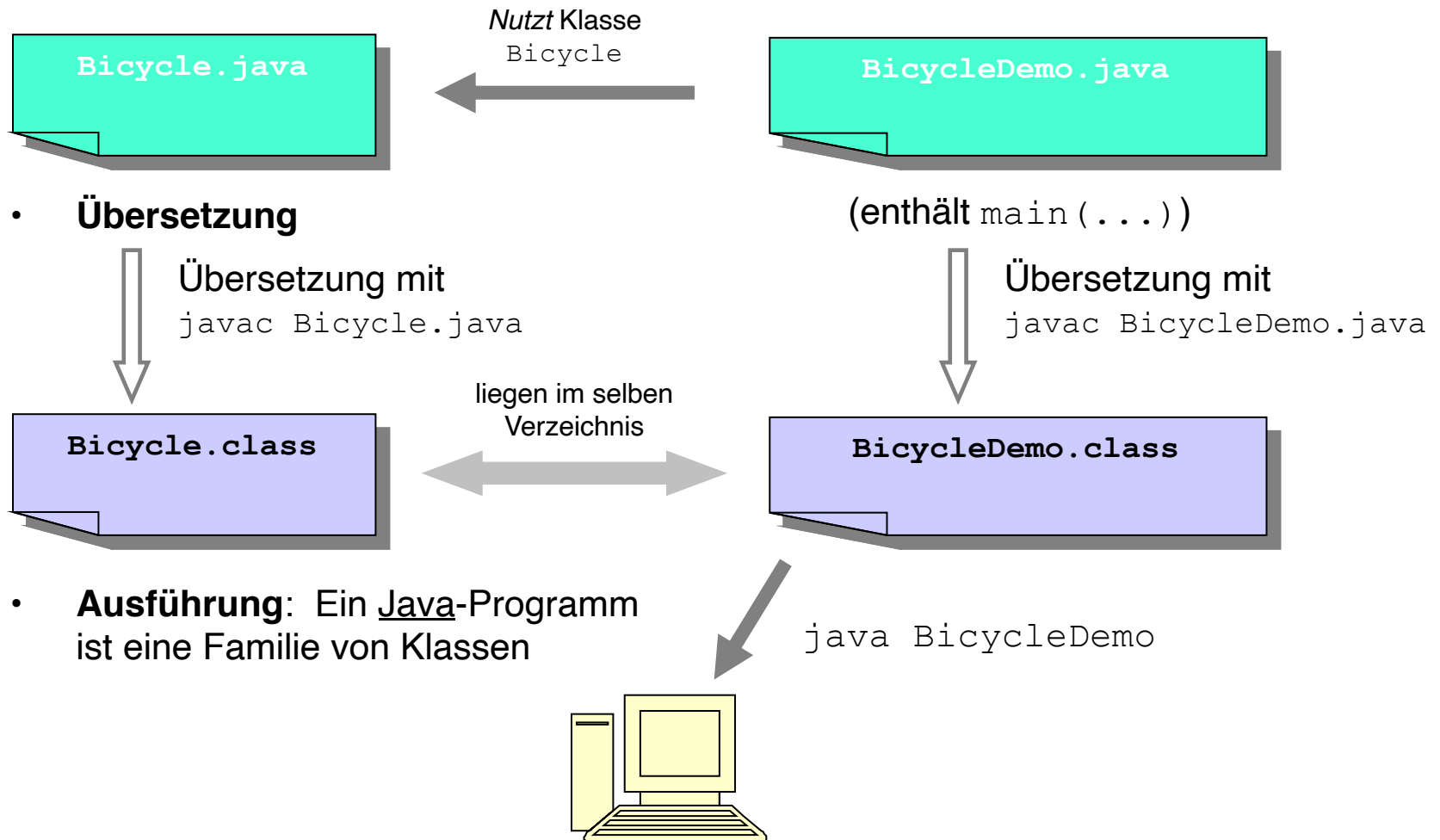
Auf die **Geschwindigkeit** (Attribut) der Instanz kann zugegriffen werden; eine Beschleunigung um 10.0 (mit der entsprechenden Methode) wird ausgeführt:

```
s = bike.speed;  
bike.speedUp(10.0);
```

Zusammenspiel der Klassen und Objekte im Hauptspeicher

lediglich eine
Hilfsklasse,
definiert kein Objekt

- Zusammenspiel der Klassen *Bicycle* und *BicycleDemo*



Selbstreferenz – die Variable `this`

- Ein Element einer Klasse hat einen **Namen** als Bezeichner, der nur **innerhalb der Klassendefinition** verwendet werden kann
- **Außerhalb der Klasse** hat das Element des Objekts den vollständigen Namen der Form

```
<Objekt-Name>.<Name>
```

Problem: *Wie kann eine Methode definiert werden, die auf die Instanz des Objekts selbst zugreift in der sie definiert wird?*

Elegante Lösung: Java stellt eine definierte Variable `this` (entsprechend “this object“!) zum Zugriff auf das aktuelle Objekt zur Verfügung:

Bsp.1:

```
public class Student {  
    private String name;  
  
    public Student(String name) {  
        this.name = name;  
    }  
    ...  
} // end class Student
```

Der aktuelle Parameter (die lokale Variable)
überdeckt die Instanz-Variable

Bsp.2: Für zwei Fahrrad-Objekte sollen die Zustände synchronisiert werden

```

public class Bicycle {
    int    cadence;
    double speed;
    int    gear;

    ...
    void synchronize(Bicycle bikeTwo) {
        this.cadence = bikeTwo.cadence;
        this.speed   = bikeTwo.speed;
        this.gear    = bikeTwo.gear;
    }
    ...
} // end class Bicycle

```

Hier würden auch folgende
Zuweisungen erlaubt sein:

```

cadence = bikeTwo.cadence;
speed   = bikeTwo.speed;
gear    = bikeTwo.gear;

```

Die Referenzierung über **this** ist
jedoch auch hier die strukturiere
Form

Aufruf mit (vgl. mit Kopierfunktionen auf **Strings**):

```

Bicycles bike1, bike2;
bike1.synchronize(bike2); // Synchronisation der Attribute

```

- Weitere Verwendungsmöglichkeiten für **this**
 - Übergabe des aktuellen Objekts, das eine Variable oder Methode enthält, an ein Unterprogramm (als aktueller Parameter)
 - Zuweisung des Wertes von **this** an eine Referenz-Variable

Methoden als Instrument der Kapselung

Analyse einer Klasse *Rectangle*

- Die Klasse *Rectangle* ist mit zwei Methoden definiert

```
class Rectangle {
    public double height,    // Hoehe
                  width;     // Breite des Rechtecks
    public int     lineSize; // Liniendicke

    ... // Konstruktor ...

    public double maxExtent(double d1, double d2) {
        return (d1 >= d2) ? d1 : d2;
    } // end maxExtent

    public Rectangle coverBox(Rectangle other) {
        double    nh, nw;
        Rectangle rr;

        nh = maxExtent(height, other.height);
        nw = maxExtent(width, other.width);

        rr = new Rectangle();
        rr.height = nh;
        rr.width  = nw;
        return rr;
    } // end coverBox
} // end class Rectangle
```

- Analyse der Klasse *Rectangle*

- *Rectangle* hat zwei Methoden:
 1. Eine Methode `coverBox` berechnet für 2 übergebene Rechtecke das umschließende Rechteck
 2. Die andere Methode `maxExtent` berechnet den Maximalwert zweier übergebener Seitenlängen (Hilfsfunktion für die Berechnung des umschließenden Rechtecks)
- *Rectangle* definiert lediglich Methoden für einen bestimmten Anwendungsfall (hier: Ermittlung des umschließenden Rechtecks)
- Im vorliegenden Beispiel gibt es für den Zugriff auf einzelne Attribute (`height`, `width`, `lineSize`) keine speziellen Methoden; auf die **Attribute wird direkt zugegriffen** – auch von außen (... spart häufig Schreibaufwand)
- Solche **Direktzugriffe auf Attribute** – insbesondere von außerhalb der definierenden Klasse – **widersprechen** dem Prinzip der **Kapselung**

Mechanismen der Kapselung

- **Kapselung**: Ein Objekt sollte in seinen Methoden alle Operationen lesender und verändernder Art umfassen, die auf ihnen möglich sind
- Hierzu gehören **lesende und verändernde Zugriffe** auf einzelne **Attribute** des Objekts (*getter* und *setter Methoden*)

Bsp.: Erweiterung der Klasse *Rectangle* um *getter* und *setter Methoden*

```
class Rectangle {
    private double height,    // Hoehe
                  width;      // Breite des Rechtecks
    private int    lineSize;  // Liniendicke

    ...
    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public int getLineSize() {
        return lineSize;
    }

    public void setLineSize(int lineSize) {
        this.lineSize = lineSize;
    }

    ...
} // end class Rectangle
```

- *getter / setter* Methoden und **private**-Modifizierer
 - Auf ein als **public** deklariertes Attribut kann beliebig zugegriffen werden, auch von anderen Klassen
 - Auf ein als **private** deklariertes Attribut kann nur innerhalb der definierenden Klasse zugegriffen werden; ein Zugriff von anderen Klassen (bzw. Instanzen anderer Klassen) erfolgt über **Zugriffsmethoden**, die als **public** definiert sind
 - **Lesemethoden** (*getter, accessor*)

Bsp.:

```
private String title;

public String getTitle() {
    return title;
}
```

- **Schreibmethoden** (*setter, mutator*)

Bsp.:

```
public void setTitle(String newTitle) {
    this.title = newTitle;
}
```

- Nutzen:
 - Funktionen können eine umfangreichere Operationalität haben, z.B.
 - die Anzahl der Zugriffe zählen oder
 - legale Zugriffe prüfen
 - **Änderungen an der Klasse** haben **keine Auswirkungen** auf die Klassen, die die Zugriffsmethoden verwenden, d.h. die Attribute (und ihre Repräsentation) sind vollständig gekapselt und können vollständig objektintern verwaltet werden

Bsp.: In der Klasse *Bicycle2* werden die **Attribute** als **private** deklariert; die bisherigen Methoden haben bereits *getter* bzw. *setter* Eigenschaften

```
public class Bicycle2 {
    /* -- Attribute */
    private int    cadence;    // Trittfrequenz
    private double speed;     // Geschwindigkeit
    private int    gear;      // Gang

    /* -- mehrere Konstruktoren (overloading) */
    public Bicycle2() {
    } // end constructor Bicycle2

    public Bicycle2(int cadence, double speed, int gear) {
        this.cadence = cadence;
        this.speed    = speed;
        this.gear     = gear;
    } // end constructor Bicycle2
}
```

Revidierte Klasse *Bicycle2* (Fortsetzung)

```

/* -- Methoden */
public void changeCadence(int newValue) {           // aendere Trittfrequenz
    cadence = newValue;
} // end changeCadence

public void changeGear(int newValue) {              // aendere Gang
    gear = newValue;
} // end changeGear

public void speedUp(double incrementValue) {        // erhoehe Geschwindigkeit
    speed = speed + incrementValue;
} // end speedUp

public void slowDown(double decrementValue) {       // bremse ab
    speed = speed - decrementValue;
} // end slowDown

public void syncBikes(Bicycle2 bikeTwo) { // synchronisiert die Zustände
    this.cadence = bikeTwo.cadence;
    this.speed   = bikeTwo.speed;
    this.gear    = bikeTwo.gear;
} // end syncBikes

public void printState() {                          // zeige den aktuellen Zustand an
    System.out.println(" |Cadence: " + cadence +
                       " |Speed:   " + speed +
                       " |Gear:    " + gear);
}
}

```

Vorteile der Kapselung von Attributen

- Die **Implementierung objektinterner Details** ist unabhängig von den Methoden, welche die Schnittstelle zur Umgebung des Objektes bilden
- **Prinzip der Abstraktion**: Ein Objekt kann durch **Methodenaufrufe** benutzt werden, ohne dass die objektinterne Datenstruktur verstanden werden muss
- **Flexibilität der Programmierung**
 - Erweist sich eine Datenstruktur als ungeeignet oder ihre Bearbeitung durch Methoden als ineffizient, kann die **Implementierung der objektinternen Details** geändert werden, ohne dass sich die Schnittstelle zur Umgebung ändert
 - Die Methodenaufrufe und ihre Parametrisierung bleiben dabei unverändert bestehen
- Eventuelle Ineffizienzen bei Attributzugriffen mittels Methoden werden durch Java-Compiler „wegoptimiert“ ...

Aufbau von Programmen

- Es gibt **Klassen**, die **Objekte** (Schablonen hiervon) definieren und solche, die als **Hilfs-, Test- bzw. Nutzklassen** dieser Objekte dienen
- Es werden **Konstruktoren** benötigt, um Instanzen von Klassen zu erzeugen (jedoch nicht von den Nutzklassen, von denen werden keine Instanzen erzeugt – sie enthalten beispielsweise die Methode `main()`)
- Neue **Instanzen** werden mittels des **Operators `new`** erzeugt

Bsp.:

```
Bicycle bike;  
  
bike = new Bicycle(0, 0.0, 1);
```

- Weitere Beobachtungen
 - Eine vereinbarte **Klasse** `class Bicycle { ... }` wird in Java als **neuer Datentyp** (hier: *Bicycle*) behandelt
 - **Zugriffe** auf einzelne Komponenten (Eigenschaften), d.h. Methoden und Attribute, erfolgen mittels eines **Selektors** `'.'`

Bsp.:

```
value = bike.speed; // bezieht sich auf Def. der Klasse 'Bicycle'  
bike.printState();
```

Ausblick

Ausgewählte Aspekte werden nachfolgend im Detail betrachtet und vertieft

```
public class BicycleDemo {  
  
    public static void main(String[] args) {  
        double s;  
        Bicycle bike = new Bicycle(0, 0.0, 1);  
  
        ...  
  
        s = bike.getSpeed();  
        bike.speedUp();  
  
        ...  
    }  
}
```

Gültigkeitsbereich von
Bezeichnern

Referenz-/Zeiger-Variablen
und Objektzugriffe

Konstruktor (vgl. Vorstellung weiter hinten)

Informationsverdeckung und Geheimnisprinzip
(basiert auf der Def. der Klasse 'Bicycle2')

Methoden und
ihre Parameter

Referenz-Variablen und Zugriff auf Objekte

Einordnung

- Auf **Objekte**, d.h. ihre Attribute und Methoden, kann über **Referenzen** zugegriffen werden
- Eine **Referenz-Variable** enthält einen **Zeiger** auf die **Adresse im Hauptspeicher** unter der das Objekt abgelegt worden ist
- Objekte, auf die **keine gültige Referenz-Variable** zeigt, können nicht mehr weiter verwendet werden; damit durch immer mehr nicht mehr referenzierbare Objekte im Hauptspeicher dieser nicht immer weiter aufgefüllt wird, sorgt die Speicherverwaltung dafür, dass der Speicher nicht mehr erreichbarer Objekte frei gegeben wird (***Garbage collection***)
- Ein **leerer Zeiger** wird mit dem **Schlüsselwort** (Literal) **null** gekennzeichnet; dieser Wert kann jeder Referenz-Variablen, unabhängig vom jeweiligen Datentyp, zugewiesen werden

Weiteres Beispiel – Rechtecke

Darstellung und Manipulation von Rechteck-Objekten in einem Programm

- Ein **Rechteck** wird hier durch die **Attribute** Breite (*width*), Höhe (*height*) und Strichstärke (*lineSize*) charakterisiert
- Für ein Objekt (Instanz der Klasse) soll die **Fläche** mittels einer **Methode** `computeArea()` berechnet werden
- Die **Klasse *Rectangle*** kann in **UML-Notation** dargestellt werden
- Es werden vier **Objekte** (als Instanzen der Klasse) erzeugt und verwaltet

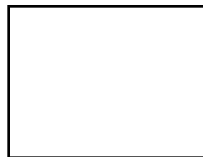
Name der Klasse

Rectangle

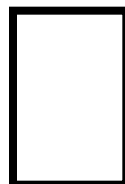
Attribute: Breite,
Höhe, Strichstärkewidth
height
lineSize

Methoden: Fläche

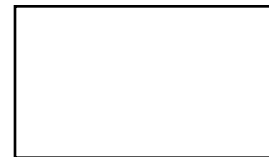
computeArea()



height: 2.0
width: 2.5
lineSize: 1



2.5
1.5
3



2.0
3.5
1



2.5
3.5
2

Definition der Klasse (vier Schritte)

- Klasse erstellen
(Datei heißt `Rectangle.java`)

```
public class Rectangle {
    ...
}
```

- Attribute (Eigenschaften) einfügen

```
public class Rectangle {
    /* -- Attribute */
    private double height;
    private double width;
    public int lineSize;
}
```

- Konstruktor einfügen

```
public class Rectangle {
    /* -- Attribute */
    /* -- Konstruktor(en) */
    public Rectangle(double height, double width) {
        this.height = height;
        this.width = width;
        lineSize = 1;          // hier immer Strichstärke 1
    }
}
```

- Methoden einfügen

```
public class Rectangle {
    /* -- Attribute */
    /* -- Konstruktor(en) */
    /* -- Methoden */
    public double computeArea() {
        return (height * width);
    }
}
```

- Fertige Klasse

```
public class Rectangle {  
  
    /* -- Attribute */  
    private double height;  
    private double width;  
    public int lineSize;  
  
    /* -- Konstruktor der Klasse */  
    public Rectangle(double height, double width) {  
        this.height = height;  
        this.width = width;  
        lineSize = 1;           // hier immer Strichstaerke 1  
    }  
  
    public double computeArea() {  
        return (height * width);  
    }  
}
```

Referenz-Variablen (in Demo-Klasse) und Generierung von Objekten

- Es sollen zur Laufzeit **vier Rechteck-Objekte** als Instanzen der **Klasse *Rectangle*** generiert werden
- In der **Klasse *RectangleDemo*** werden die Referenz-Variablen `rect1`, `rect2`, `rect3`, `rect4` **deklariert**

```
public class RectangleDemo {  
  
    public static void main(String[] args) {  
        double area1;  
        Rectangle rect1 = new Rectangle(2.0, 2.5),  
            rect2 = new Rectangle(2.5, 1.5),  
            rect3 = new Rectangle(2.0, 3.5),  
            rect4 = new Rectangle(2.5, 3.5);  
  
        ...  
        rect2.lineSize = 3; // schreibender Zugriff auf Attribut  
        rect4.lineSize = 2; // dto.  
  
        area1 = rect1.computeArea(); // Aufruf der Methode  
        ...  
    }  
}
```

Zugriffe auf Objekte über Referenz-Variable

Einfaches Beispiel – Student

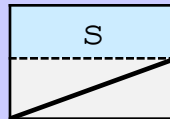
- Referenz, Instanz, Zuweisungen

```
class Student { ... }
```

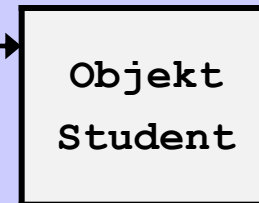
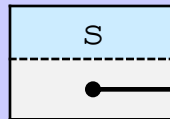
Student.java

Klasse Student

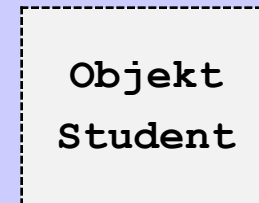
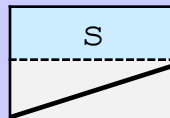
```
Student s;
```



```
s = new Student();
```



```
s = null;
```



Objekt **nicht mehr verwendbar** (keine Referenzen auf den Speicherbereich)

- **Details:** **Referenz-Variablen** haben einen **Typ** (wie andere Variablen auch):
Referenz-Variable *s* ist vom Typ *Student*; genauer gesagt *s* kann auf ein Objekt vom Typ *Student* zeigen

Bsp.:

Typ der
Referenzvariablen

```
Rectangle rect1 = new Rectangle(0.0, 0.0);  
...
```

Bezeichner

```
rect1.width = 5.0;
```

Anlegen eines neuen Objekts
durch den Konstruktor (s. S. 71)

Zugriff auf Attribut *width* desjenigen
Objekts der Klasse *Rectangle* auf
das *rect1* verweist

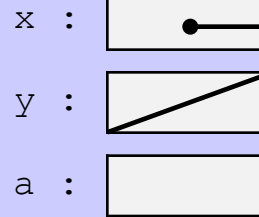
Operationen auf Rechteck-Objekten

■ Anweisungsfolgen

Konstruktor entsprechend S.71

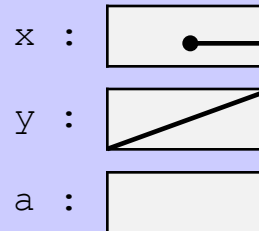
```
Rectangle x, y;  
double    a;
```

```
x = new Rectangle(0.0, 0.0);  
y = null;
```



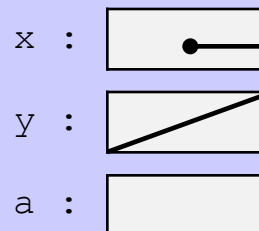
Object Rectangle
height: 0.0
width : 0.0
lineSize: 1
area()

```
x.width = 5.3d;
```



Object Rectangle
height: 0.0
width : 5.3
lineSize: 1
area()

```
x.height = x.width + 1.0d;  
x.lineSize = 2;
```

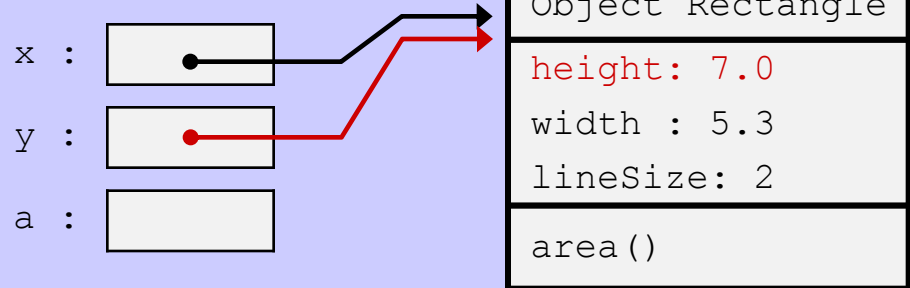


Object Rectangle
height: 6.3
width : 5.3
lineSize: 2
area()

■ Anweisungsfolgen (Fortsetzung)

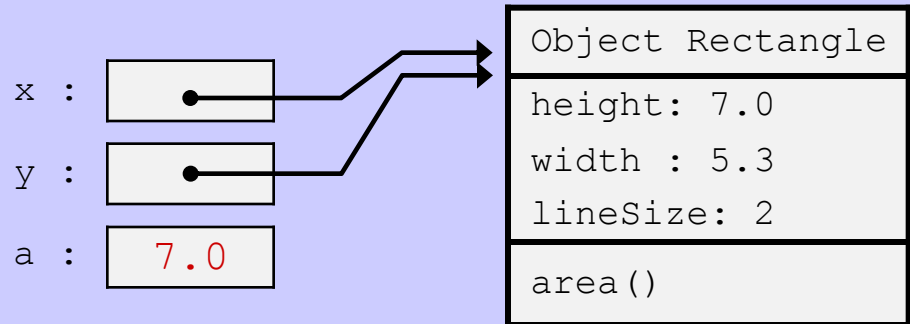
```
y = x;
y.height = 7.0d;
```

Frage: Was würde passieren, wenn stattdessen die Zuweisung `x = y;` angegeben würde?



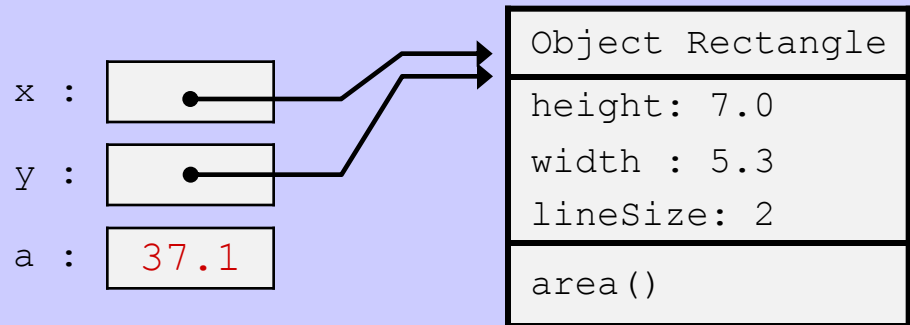
Antwort: Da `y == null`, würde `x == null` und die zuvor generierte Instanz würde nicht mehr erreichbar sein (\rightarrow *Garbage collector*)

```
a = x.height;
```



\rightarrow `a = y.computeArea();`

$\text{height} \times \text{width} = 7.0 \cdot 5.3$
 $= 37.1$



Seiteneffekte bei multiplen Referenzen

- Im Zusammenhang mit der **Zuweisung von Referenzvariablen** (Zeiger; z.B. in der Form $y = x$) können **Seiteneffekte** auftreten, d.h. der schreibende Zugriff auf Attribute des Objektes über Referenzvariable x ändert auch entsprechende Zugriffe über y
- Seiteneffekte führen leicht zu **Fehlern**; diese – über Zuweisungen von Referenzvariablen/Zeigern erzeugten Seiteneffekte – **erschweren** zudem die **Verständlichkeit von Programmen** (als Konsequenz sollten derartige Seiteneffekte mit Bedacht eingesetzt werden)
- In manchen Fällen ist es jedoch zweckmäßig, dass zumindest temporär mehrere Referenzvariablen auf ein und dasselbe Objekt zeigen (beispielsweise basieren **Operationen auf dynamischen Datenstrukturen**, z.B. **Listen**, auf der Zuweisung von Zeigern) – es ist daher wichtig zu verstehen,
 - welche Seiteneffekte in Verbindung mit Referenzvariablen auftreten und
 - wie diese kontrolliert werden können

Bsp. für Seiteneffekt

```
class Person {
    private String name;
    private ValuePair v;

    public Person(
        String name, int x, int y)
    {
        this.name = name;
        v = new ValuePair(x,y);
    }

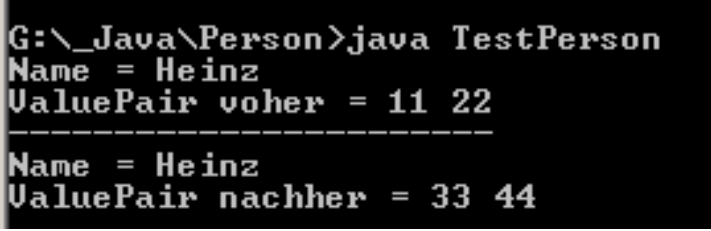
    public String getName() {
        return name;
    }

    public ValuePair getValuePair() {
        return v;
    }
}

class ValuePair {
    int x;
    int y;

    ValuePair(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
class TestPerson {
    public static void main(String[] args) {
        Person p = new Person("Heinz", 11, 22);
        String n1 = p.getName();
        ValuePair xy1 = p.getValuePair();
        System.out.println("Name = " + n1);
        System.out.println("ValuePair vorher = "
            + xy1.x + " " + xy1.y);
        System.out.println("-----");
        n1 = "Hugo";
        String n2 = p.getName();
        System.out.println("Name = " + n2);
        xy1.x = 33;
        xy1.y = 44;
        ValuePair xy2 = p.getValuePair();
        System.out.println("ValuePair nachher = "
            + xy2.x + " " + xy2.y);
    }
}
```



```
G:\_Java\Person>java TestPerson
Name = Heinz
ValuePair vorher = 11 22
-----
Name = Heinz
ValuePair nachher = 33 44
```

Effekt? Die Instanzvariablen von Objekt p.v wurden verändert!

Statische und nicht-statische Methoden und Attribute

Einleitung – Statische Methoden und Attribute

- **Statische Attribute** (statische Variablen) definieren globale Variablen, z.B. in einer Klasse, auf die von verschiedenen Methoden zugegriffen wird; statische Elemente werden durch den Modifizierer **static** gekennzeichnet

Bsp.:

```
public class Program {  
    static int counter = 0; // globale Variable  
    ...  
    public static void main(String[] args) {  
        ...  
    }  
  
    static double readPosFloat(int count) {  
        ...  
    }  
}
```

■ Einordnung:

- Methoden und Attribute, die **statisch** deklariert sind, **existieren während der Ausführung des Programms nur einmal**
- Werden mehrere **Objekte** (einer Klasse) **instanziiert**, so **existieren die statischen Elemente nur einmal** als Beschreibung des Zustands der definierten Klasse
 - gültig für alle Objekte der Klasse (global)
 - für den (statischen) Methodenaufruf ist kein Objekt (als Instanz der Klasse) notwendig

Erläuterung:

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

Die `main()` Methode wird vom System aufgerufen, obwohl **keine Instanz der zugehörigen Klasse** erzeugt wurde

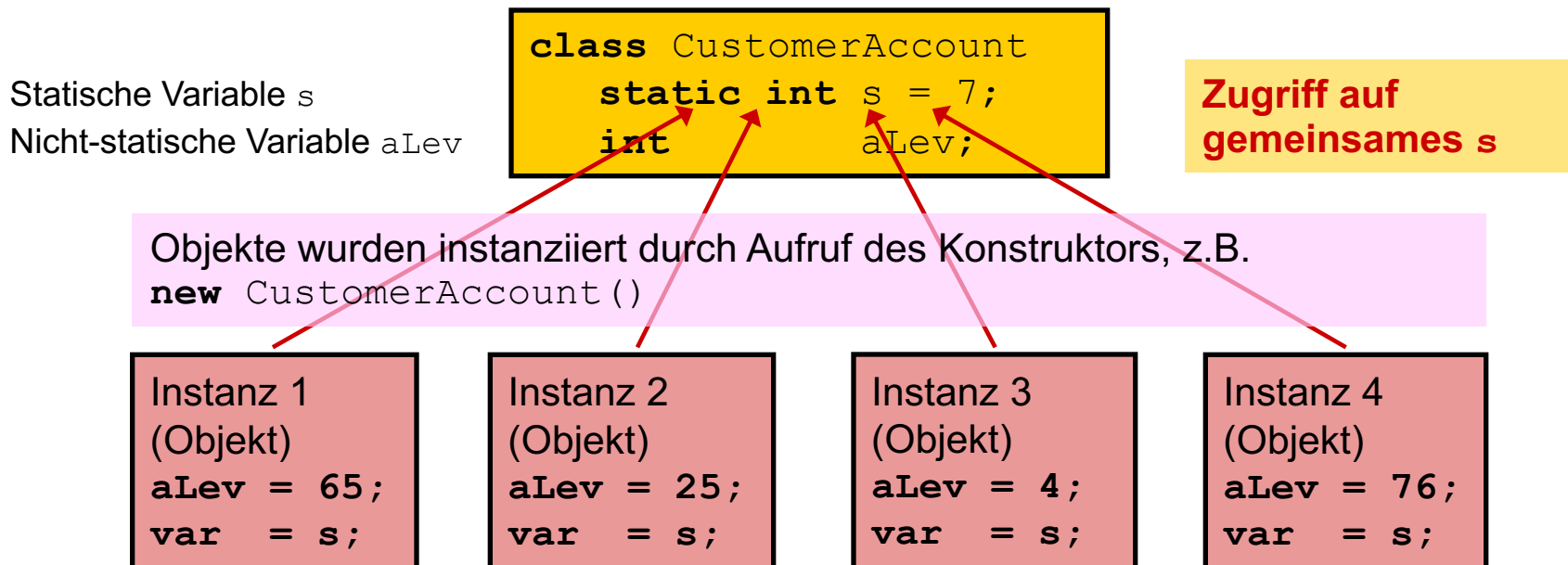
- ```
public static double sin(double x) { ... }
```

 - die Funktion `sin` ist eine Methode der Klasse `Math`
 - die Methode `sin` wird **nicht für eine Instanz** aufgerufen, sondern „einfach so“, z.B. `double d = Math.sin(2.0);` (Aufruf mit vorangestelltem Klassennamen `Math`)

Statische und nicht-statische (dynamische) Attribute und Methoden

Attribute

- Wenn innerhalb von Methoden **lokale Variablen** definiert werden (auch innerhalb einzelner separater Blöcke), dann sind diese Variablen **nicht-statisch**, da sie nur in diesem Block existieren und nach Beendigung der Ausführung des Blocks eliminiert und bei neuerlichem Aufruf wieder neu instanziiert werden
- Gleiches gilt für **Klassenvariable** und **Methoden** einer Klasse, die **nicht statisch** deklariert wurden und die bei einer Objektinstanziierung neu generiert werden
- Beispiel – instanz-spezifische Ausprägungen von (nicht statischen) Attributen



■ Implementierung des Beispiels für statische und nicht-statische Variable

```

class CustomerAccount {
    static int s;      // statisches Attribut
    int      aLev;     // nicht-statisches Attribut

    void manageAccount() {
        aLev = 4;      // nicht-statisches Attribut (instanzspezifisch)
        s     = 77;     // statisches Attribut (klassenspezifisch)
    }
}

// Zugriffe ausserhalb der Klasse:
CustomerAccount.s = 40; // Zugriff auf statisches Attribut s
                       // es existiert noch keine Instanz der Klasse!

CustomerAccount cAcc1 = new CustomerAccount(),
                cAcc2 = new CustomerAccount();

cAcc1.aLev = 65;      // Zuweisung möglich, da cAcc1 auf zuvor erzeugtes
                       // Objekt der Klasse CustomerAccount verweist

cAcc1.s      = 64;     // hat denselben Effekt wie "CustomerAccount.s = 64;"
                       // im Anschluss gilt ebenfalls: cAcc2.s == 64

```

Methoden

- **Nicht-statische Methoden** werden jeweils bezogen auf eine bestimmte Instanz (Objekt) aufgerufen

Zugriff auf alle Attribute bzw. Attributwerte der betreffenden Instanz möglich

- **Statische Methoden** werden bezogen auf eine Klasse, und nicht für eine einzelne Instanz aufgerufen

Zugriff:

- **kein Zugriff auf nicht-statische Attribute von Instanzen** möglich
- Zugriff auf statische Attribute der Klasse möglich

- Aufrufe **nicht-statischer** Methoden: `rObj.method1()`

Erläuterung: `rObj` ist eine **Referenzvariable** (Zeiger) die auf ein entsprechendes Objekt verweist, das wiederum zuvor mittels **new** und Konstruktor der Klasse generiert wurde

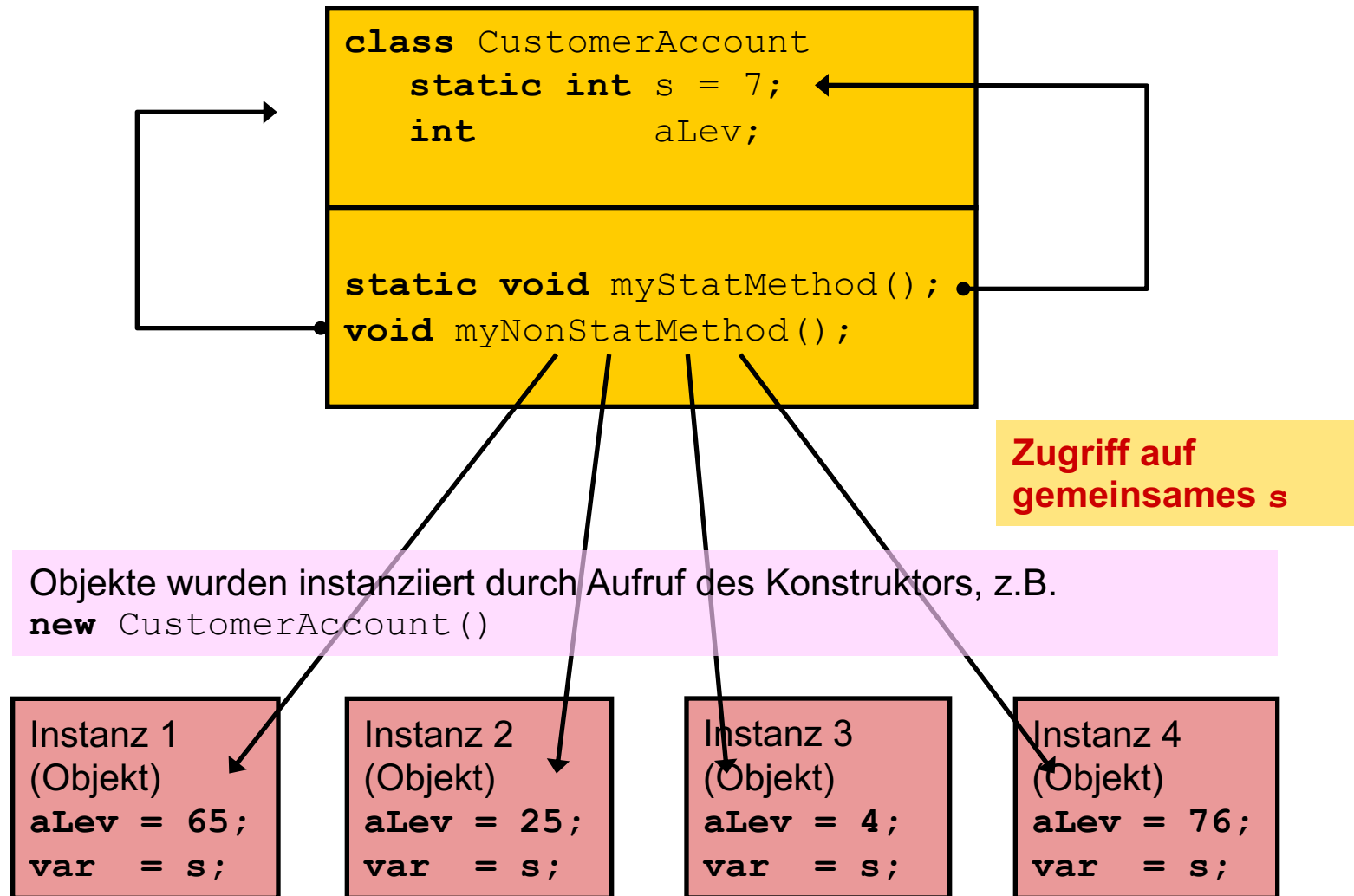
- Aufruf **statischer** Methoden: `<Klassenname>.method2()`

Erläuterung:

- es muss beim Aufruf nicht unbedingt eine Instanz der Klasse existieren
- existiert eine Instanz der Klasse (Objekt), kann (obwohl unschön) die statische Methode auch über `rObj.method2()` aufgerufen werden (`rObj`: Referenzvariable auf das Objekt)

```
class MyClass {
    static int statMethod() {
        ...
    }
}
...
int i = MyClass.statMethod();
```

- Beispiel – statische und nicht-statische Methoden



■ Beispiel für die typische Verwendung von **Klassenvariablen**

```

class Point {
    public static int number; // statisches Attribut zum Zählen der
                             // Instanzen der Klasse
    public int x, y;         // nicht-statisches Attribut

    public Point() { // Konstruktor
        number = number + 1; // zaehlt jede neue Instanz
    }

    public void shiftPoint(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}

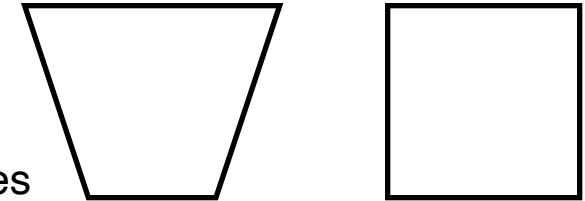
// Zugriffe ausserhalb der Klasse:
System.out.println("Anzahl Punkte: " + Point.number);  Ausgabe: 0
Point p1 = new Point(),
      p2 = new Point();
System.out.println("Anzahl Punkte: " + Point.number);  Ausgabe: 2
p1.shiftPoint(2, 3); // Ausfuehren der shiftPoint Methode des Objekts p1
p2.shiftPoint(1, 5); // Ausfuehren der shiftPoint Methode des Objekts p2
p2.shiftPoint(3, 1); // Ausfuehren der shiftPoint Methode des Objekts p2
System.out.println("Punkt 1 = (" + p1.x + ", " + p1.y + ")");  p1 = (2, 3)
System.out.println("Punkt 2 = (" + p2.x + ", " + p2.y + ")");  p2 = (4, 6)

```

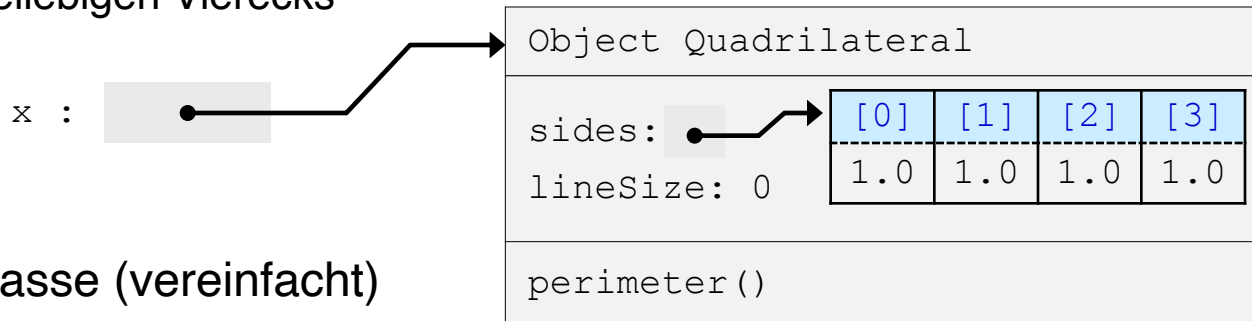
Objekte und Arrays

Arrays als Objekt-Attribute

- Attribute eines Objekts können *Array*-Variablen sein



Bsp.: *Array*-Variable zur Speicherung der **Seitenlängen** eines beliebigen Vierecks



- Java-Klasse (vereinfacht)

```
class Quadrilateral {
    double[] sides; // Referenz auf Array mit Seitenlaengen
    int     lineSize;

    public Quadrilateral() { ... } // Konstruktor

    public double perimeter() {
        return (sides[0] + sides[1] + sides[2] + sides[3]);
    }
}
```

■ Verwendung der Java-Klasse (z.B. in einem Hauptprogramm)

```
Quadrilateral fLeg = new Quadrilateral();
```

1


```
fLeg.sides = new double[4];
```

2

```
for (int i = 0; i < 4; i++) {  
    fLeg.sides[i] = 1.0d;  
}
```

3

fLeg :

```
sides:   
lineSize: 0  
  
perimeter()
```

[0]	[1]	[2]	[3]
0.0	0.0	0.0	0.0

[0]	[1]	[2]	[3]
1.0	1.0	1.0	1.0

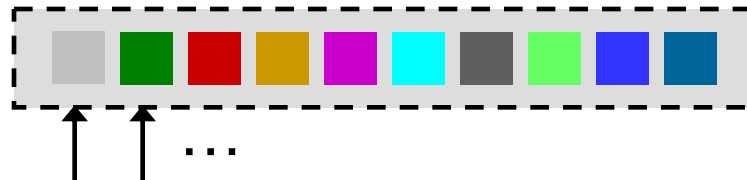
Bemerkungen:

- Für die Instanziierung des *Arrays* muss dessen Größe und der Attributname für die *Array*-Variable außerhalb der Klasse bekannt sein (hier sind die Details der Objektdaten nicht gekapselt!)
- Die Werte des *Arrays* werden automatisch mit 0.0 initialisiert; die Zuweisung der neuen Werte erfolgt hier ebenfalls von außen durch direkten Zugriff auf die Attributelemente

Arrays von Objekt-Referenzen

Noch einmal *Arrays* ...

- Wir haben *Arrays* bereits kennen gelernt, die eine Menge **gleichartiger Elemente** zu einer **Struktur** (oder **Verbund**) zusammenfassen; die Elemente sind **linear angeordnet** und können mittels eines **Indexwerts** adressiert werden (vgl. **Teil V**)



- Array*-Variablen werden in Form von Referenz- (Zeiger-) Variablen deklariert (siehe **Teil V**)

```
<datentyp> [] <Referenz-Variablen-Name>;
```

Erläuterungen:

- In Java werden **Klassen** als **neue Datentypen** behandelt
- Ein **Objekt** (Instanz einer Klasse) wird mittels einer **Referenzvariable** deklariert, die auf das mittels **new** und **Konstruktor** generierte Objekt zeigt

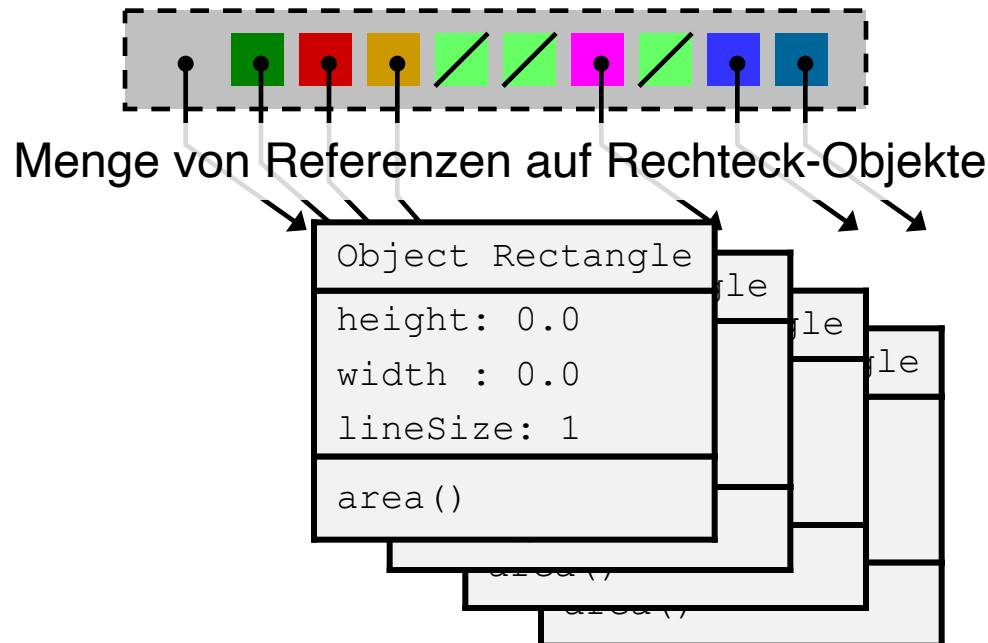
- Ein *Array* kann aus *Elementen* bestehen, die alle vom *selben Referenztyp* sind

Bsp.: Ein *Array*-mit Zeiger auf Objekte (desselben Typs)

```
Rectangle[] R = new Rectangle[10];
```

Die Elemente des *Arrays* werden *zunächst mit null initialisiert* – wir nehmen an, dass *für einzelne Elemente ein Objekt instanziiert und zugewiesen* wird ...

```
R[0] = new Rectangle(0.0, 0.0);
R[1] = new Rectangle(0.5, 1.0);
...
```



- Einordnung und Erläuterungen zu *Arrays* mit Objekt-Referenzen
 - Die **Referenzvariable** `R` verweist auf ein *Array* mit 10 Elementen
 - Das *Array* enthält **Elemente**, die **alle vom selben Datentyp** sind; hier sind dies **Referenzen auf Rechteck-Objekte** `Rectangle` (das *Array* selbst enthält keine Objekte, sondern nur die **Zeiger**)
 - Die Instanziierung (Allokation von Speicher für die Elemente) des *Arrays* erfolgt – ebenso wie die von Objekten (als Klasseninstanzen) – mittels des **Operators** `new`
 - ***Arrays in Java* sind selbst Objekte**, die beispielsweise ein **Attribut** `length` besitzen – allerdings sind *Arrays* **spezielle Objekte**, da für die Klasse der *Arrays* **keine speziellen Konstruktoren** definiert sind!
 - Auf die **Elemente eines instanziierten *Arrays*** kann mittels einer **Index-Variable** mit einem Wert, $0 \leq \text{Index-Wert} < \text{Array-Länge}$, zugegriffen werden

- Ein Beispiel für Rechtecke (Rectangles)

```
public class Rectangle {
    double height,
           width;
    int    lineSize;
    ...
}
```

```
public class RectangleTester {
```

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

```
        Rectangle[] rectangles = new Rectangle[10];
```

```
        for (int i = 0; i < rectangles.length; i++) {
```

```
            rectangles[i] = new Rectangle(0.5, 1.0);
```

```
            rectangles[i].lineSize = 2;
```

```
        }
```

```
        ...
```

```
    }
```

```
}
```

Referenz auf ein *Array* mit Elementen, die auf *Rectangles* zeigen

Allokation eines *Arrays* mit 10 Elementen

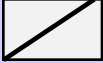
Für jeden Index wird nacheinander ein Objekt der Klasse *Rectangle* instanziiert

Auf die Elemente des *Arrays* kann jetzt über die Index-Variable *i* zugegriffen werden

- Verweis- und Objektstruktur

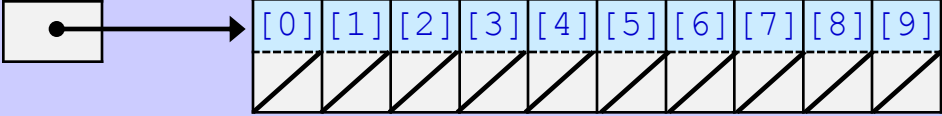
```
Rectangle[] R;
```

1

R : 

```
R = new Rectangle[10];
```

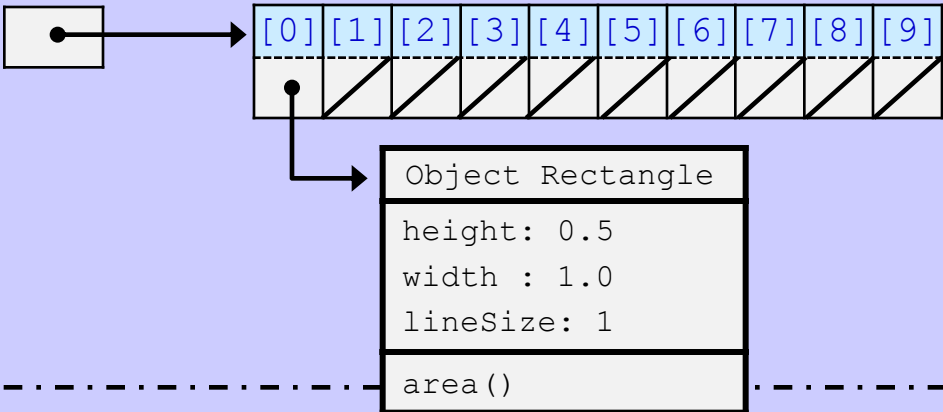
2

R : 

```
R[0] = new Rectangle(
    0.5, 1.0);
```

3

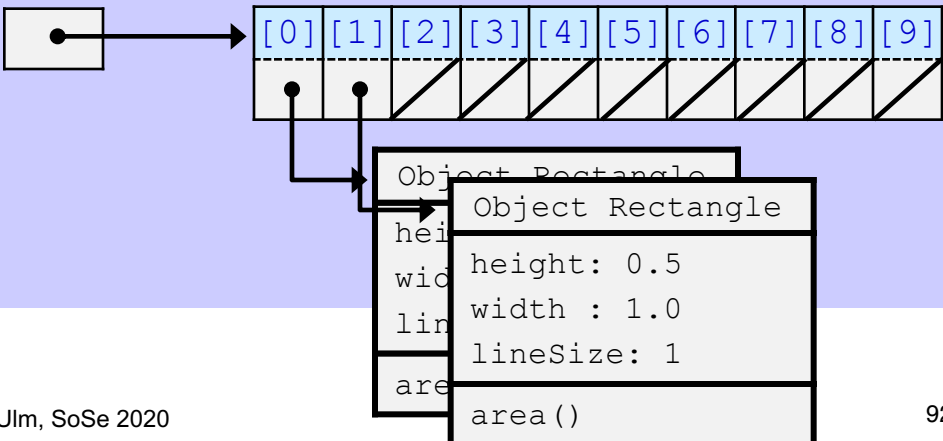
4

R : 

```
R[1] = new Rectangle(
    0.5, 1.0);
```

3

4

R : 

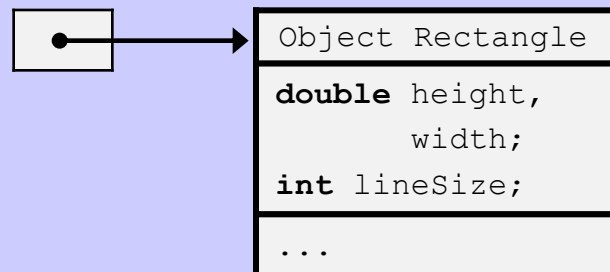
- Die in einem *Array* gespeicherten Referenzen müssen alle vom selben Typ sein, d.h. **Verweise auf verschiedene Objekte** sind nicht erlaubt

Bsp.:

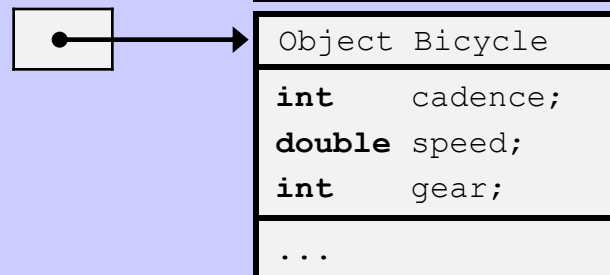
```
Rectangle rect = new Rectangle(0.5, 1.0);
Bicycle bike = new Bicycle(0, 0.0, 1);

Rectangle[] R = new Rectangle[2];
```

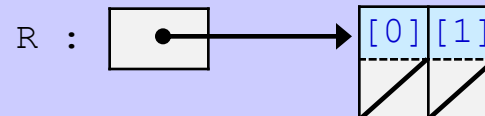
Rectangle rect = **new** Rectangle(0.5, 1.0); rect :



Bicycle bike = **new** Bicycle(0, 0.0, 1); bike :



Rectangle[] R = **new** Rectangle[2];



```
R[0] = rect; // OK!
R[1] = bike; // Fehler: Typen nicht kompatibel,
              // Compiler-Fehler
```

Konstrukturen, Objektinitialisierung und Freigabe

Anforderung von Speicherplatz und Instanzvariablen initialisieren

- **Objekttypen** (aus Klassen) werden in Java anders als primitive Typen behandelt: die **Variablen-Deklaration für ein Objekt** legt nur eine **Referenz** (Zeiger) auf das Objekt an; **Objekte** selbst müssen **explizit erzeugt** werden (Konstruktor)
- **Objekterzeugung**:
 - Bestimmung von ungenutztem Speicherbereich
 - Belegen der Instanz-Variablen (Attribute) mit Werten

... danach wird die **Referenz auf das Objekt** zurück geliefert

Bsp.:

```
public class PairOfDice {
    public int die1 = 3, // Zahl auf dem ersten Wuerfel
              die2 = 4; // Zahl auf dem zweiten Wuerfel

    public void roll() { // zufaellig wuerfeln
        die1 = (int) (Math.random() * 6) + 1;
        die2 = (int) (Math.random() * 6) + 1;
    }
}
```

Hinweis: Werden **keine Initialwerte** angegeben, so werden die **Instanz-Variablen** mit einem **Default-Wert** belegt (vgl. **Teil V**, Initialisierung von *Arrays*)

Konstruktoren

Einordnung und Vorbetrachtungen

- Objekte werden mit dem **Operator new** erzeugt (vgl. **Teil V**)

```
Bsp.:  PairOfDice dice;           // Deklarationen einer Referenz-Variablen  
  
        dice = new PairOfDice(); // konstruiere ein neues Objekt
```

- Der Aufruf von `PairOfDice()` (zusammen mit **new**) erinnert an einen Funktionsaufruf – es ist ein **Konstruktor** (spezielle Sorte von Unterprogramm)

Eigenschaften (vgl. S.39 + 40)

- Ein **Konstruktor** wird im Rumpf der entsprechenden Klasse definiert
- Der **Name eines Kontruktors** ist gleich dem **Klassennamen**
- Ein Konstruktor hat **keinen Rückgabewert** (auch nicht **void**!)
- Der **Rumpf eines Konstruktors** ist wie eine normale Methode (Funktion) aufgebaut, mit einem Block von Anweisungen

- Über eine **Parameterliste** können **Initialisierungswerte** für die Instanz-Variablen (Attribute) übergeben werden; die Parameter können auch anderweitig zu Steuerungszwecken dienen
- Falls für eine Klasse **kein Konstruktor angegeben** wird, legt Java **automatisch** einen sog. **Default-Konstruktor** ohne Parameter an; dieser
 - allokiert bei seinem Aufruf Speicher,
 - initialisiert die Instanz-Variablen und
 - liefert eine Referenz auf das neu generierte Objekt zurück
- **Wichtig:**
 - Wird in der Klassendefinition **ein Konstruktor angegeben**, dann wird kein (zusätzlicher) **Default-Konstruktor** angelegt,
 - Es können **mehrere Konstruktoren** für eine Klasse deklariert werden
- Als **Modifizierer** wird meist **public** verwendet; dies ermöglicht es den anderen Klassen, neue Objekte der geg. Klasse zu erzeugen

Aufruf

- Ein **Konstruktor** kann ausschließlich mithilfe des **Operators new** aufgerufen werden (Konstruktoren sind damit keine Methoden)

Bsp.: `new <Klassen-Name> (<Parameter-Liste>);`

Erläuterung: Die **<Parameter-Liste>** kann **auch leer** sein, abhängig von der Deklaration; beim Aufruf des **Default-Konstruktors** ist die **<Parameter-Liste>** immer leer

- Ablauf** bei Aufruf und Ausführung eines Konstruktors
 - Es wird ein **Block freien Speichers angefordert**, der groß genug ist, ein Objekt der Klasse zu speichern
 - Initialisierung der Instanz-Variablen** des Objekts; wird in der Deklaration einer Instanz-Variablen ein Initialwert angegeben (Ausdruck), so wird der Wert berechnet und abgespeichert; ansonsten wird ein *Default*-Wert abgelegt
 - Die **aktuellen Parameter des Konstruktors** (wenn vorhanden) werden ausgewertet und den **formalen Parametern zugewiesen** (wie bei Unterprogrammen/Funktionen)
 - Die **Anweisungen im Rumpf** des Konstruktors (wenn vorhanden) werden ausgeführt
 - Eine **Referenz (Zeiger)** auf das Objekt wird als **Wert des Konstruktor-Aufrufs zurück geliefert**

Überladen von Konstruktoren

- Es können für eine Klasse **mehrere Konstruktoren** deklariert werden, davon kann auch einer wie der *Default*-Konstruktor aufgebaut sein

Bsp.:

```
public class PairOfDice {
    public int die1, // Zahl auf dem ersten Wuerfel
              die2; // Zahl auf dem zweiten Wuerfel

    public PairOfDice() { // wie Default (leere Parameterliste, leerer Block)
    }

    public PairOfDice(int die1, int die2) {
        this.die1 = die1;
        this.die2 = die2;
    }

    public PairOfDice(boolean flagRandom) {
        if (flagRandom)
            roll(); // Aufruf der roll() Methode
        else {
            die1 = 0;
            die2 = 0;
        }
    }

    public void roll() {
        die1 = (int)(Math.random() * 6) + 1;
        die2 = (int)(Math.random() * 6) + 1;
    }
}
```

Hinweis: Alle Konstruktoren (hier 3) unterscheiden sich in der Parameteranzahl und/oder den Parametertypen (**Signatur**)!

- Beispiel Java-Datentyp `String`, besitzt **mehrere Konstruktoren**
 - `public String()`
 - `public String(String value)`
 - `public String(char[] value)`
 - `public String(char[] value, int offset, int count)`
 - ...

- Insgesamt sind eine Reihe **initialer Aktionen** während der Ausführung eines Konstruktors möglich
 - der **Zustand der Objektvariablen** kann gezielt **initialisiert** werden
 - Parameterisierung des Konstruktors erlaubt die Wahl der **Anfangsbelegung der Instanz-Variablen**
 - Ein Konstruktor kann **zusätzliche Anweisungen** enthalten; diese können weitere Objekte mit einbeziehen oder andere neue Objekte erzeugen
 - **verschiedene Algorithmen** zur Initialisierung sind möglich
 - andere zum Zeitpunkt der Erzeugung existierende Objekte können genutzt und ggf. verändert werden

Freigabe von Objekten (*Garbage collection*)

Allgemeines zur Verwaltung von Objekten – speziell in Java

- Generell: **Objekte**, die **nicht mehr benötigt** werden, sollten gelöscht werden, damit der von ihnen belegte Speicherplatz freigegeben und wieder verwendet werden kann
- Möglichkeiten:
 - Aufruf eines **Löschooperators** und Ausführung einer speziellen Destruktor-Methode
 - Vollautomatische Bereinigung durch das **System**
- Java: Mit **new** werden neue Objekte erzeugt! Java kennt keine Destruktoren^(*), d.h. es gibt keinen zu **new** symmetrischen Operator, der dafür sorgt, dass zuvor erzeugte Objekte explizit gelöscht werden
- Das Java-System erkennt automatisch, wenn ein Objekt nicht mehr von einer Variablen direkt oder indirekt erreichbar ist; **nicht erreichbare Objekte** werden durch den **Garbage collector** gelöscht; **freier Speicherplatz wird kompaktiert**, damit immer größtmögliche freie Plätze entstehen (ähnliches wie bei der Fragmentierung und Defragmentierung von Festplatten)

(*) Es gibt eine Methode `finalize()` ([http://download.oracle.com/javase/6/docs/api/java/lang/Object.html#finalize\(\)](http://download.oracle.com/javase/6/docs/api/java/lang/Object.html#finalize())); um diese muss sich der Benutzer jedoch nicht kümmern

Java *Garbage collector*

- Der Java *Garbage collector* durchsucht von Zeit zu Zeit den gesamten Speicher nach Objekten und gibt den Speicherplatz für diejenigen Objekte wieder frei, auf die keine Referenzvariable mehr zeigt
- Jedoch: Die periodisch durchgeführte *Garbage collection* benötigt nicht unerhebliche Rechenzeit
 - Prüfung aller Referenzvariablen hinsichtlich direkter und indirekter Referenzen auf Objekte
 - Verschiebung der im Speicher verbleibenden Objekte in entstehende Speicherlücken; Freigabe möglichst zusammenhängender Speicherbereiche
 - Belegung der von Verschiebungen betroffenen Referenzvariablen mit neuen Referenzen (Zieladressen)
- **Technik:** Es gibt verschiedene Verfahren zur automatischen Speicherbereinigung, die alle auf einem der drei folgenden Grundverfahren basieren:
 - Referenzzählung
 - *Mark & Sweep*
 - *Stop & Copy*

Java bietet verschiedene *Garbage Collectors* an

- Nach außen hat die automatische *Garbage collection* keinerlei Wirkung; Vorteil ist die **Vermeidung „hässlicher“ Fehlerquellen bei der Programmierung**
- Nachteil: Manchmal erfolgt die *Garbage collection* zu einer unvorhergesehenen Zeit, man weiß nie, wann die „Müllabfuhr“ kommt; wenn die *Garbage collection* läuft, dann können andere Programme unterbrochen bzw. behindert werden
- **Kann man den Vorgang der Bereinigung irgendwie beeinflussen?**

Prinzipiell nein, aber man kann **Objektreferenzen selbst auf `null`** setzen

([http://download.oracle.com/javase/6/docs/api/java/lang/System.html#gc\(\)](http://download.oracle.com/javase/6/docs/api/java/lang/System.html#gc()))

GC bei niedriger Systemauslastung anstoßen: `System.gc();`

- Allgemein gilt:
 - Wenn ein Objekt nicht mehr benötigt wird, die referenzierende Variable aber noch länger gültig bleibt, sollte man diese auf `null` setzen
 - *Garbage collection* kann so den Speicher für das Objekt schon frühzeitig freigeben

```
Person bundeskanzler = new Person();
// (verwende bundeskanzler)
bundeskanzler = null; // Mitteilung: Objekt wird nicht mehr benötigt
// (hier folgen zeitaufwändige Berechnungen)
```

- Das ist jedoch überflüssig, wenn der Gültigkeitsbereich der Variablen ohnehin bald endet, z. B. innerhalb einer kurzen Methode

3. Struktur und Funktionalität von Java-Programmen

- Gültigkeit und Sichtbarkeit von Bezeichnern
- Namensräume und Zugriffskontrolle
- Behandlung von Programmausnahmen (*Exceptions*)

Gültigkeitsbereiche und Sichtbarkeit von Bezeichnern

Bezeichner, Gültigkeitsbereiche und Überdeckungen

Einordnung

- Java-Programme sind hierarchisch strukturiert
 - Pakete (Java-Dateien in einem Verzeichnis)
 - Klassen
 - Methoden
 - Anweisungen
 - ...
- Auf jeder Ebene eines Programms werden **Bezeichner** benötigt, um Dinge, Datenstrukturen, Objekte, etc. zu benennen; aus diesem Grund müssen Bezeichner **eindeutig** sein und es muss an jeder Stelle klar sein, welcher Bezeichner von einem gegebenen Typ gemeint ist (wir haben **Blöcke** bereits in Klassen, Unterprogrammen und in Blöcken (vgl. **Teil III**) verwendet)
- Bezeichner können sich **gegenseitig verdecken** – die Regeln der Sichtbarkeit hängen vom Gegenstand (Datum mit Datentyp, Methode, etc.) ab

Allgemeine Regeln

- Eine **lokale Variable** oder ein **formaler Parameter** kann denselben Bezeichner (Namen) tragen wie eine **Objekt-Variable** oder eine **Klassen-Variable** (statisch); in diesem Fall wird die Objekt- bzw. Klassen-Variable innerhalb des Gültigkeitsbereichs der lokalen Größen **überdeckt**
- Es ist dennoch weiterhin möglich, auf **überdeckte Variablen** zuzugreifen
- **Zugriff** auf **Klassen-Variablen**: **<Klassen-Name>.<Variablen-Name>**

Bsp.: `class Test1 {
 static int count; // Klassen-Variable`

```
    static void doSomething() {  
        int count = 5; // lokale Variable  
        Test1.count = count; // weist der Klassen-Variable den Wert 5 zu  
    }  
}
```

(lokale Variable count überdeckt die globale Variable count)

- **Zugriff** auf **Objekt-Variablen** mittels Selbstreferenz: **this.<Variablen-Name>**

Bsp.: `class Test2 {
 int count; // Objekt-Variable`

```
    void doSomething() {  
        int count = 5; // lokale Variable  
        this.count = count; // weist der Member-Variable den Wert 5 zu  
    }  
}
```


- In Java können die Namen formaler Parameter oder lokaler Variablen **innerhalb des Blocks** der zugehörigen Methode (Unterprogramm) **nicht neu definiert** werden, auch in geschachtelten Blöcken dürfen lokale Variablen **nicht** neu definiert werden

Bsp.:

```
void badExample(int y) {
    int x;

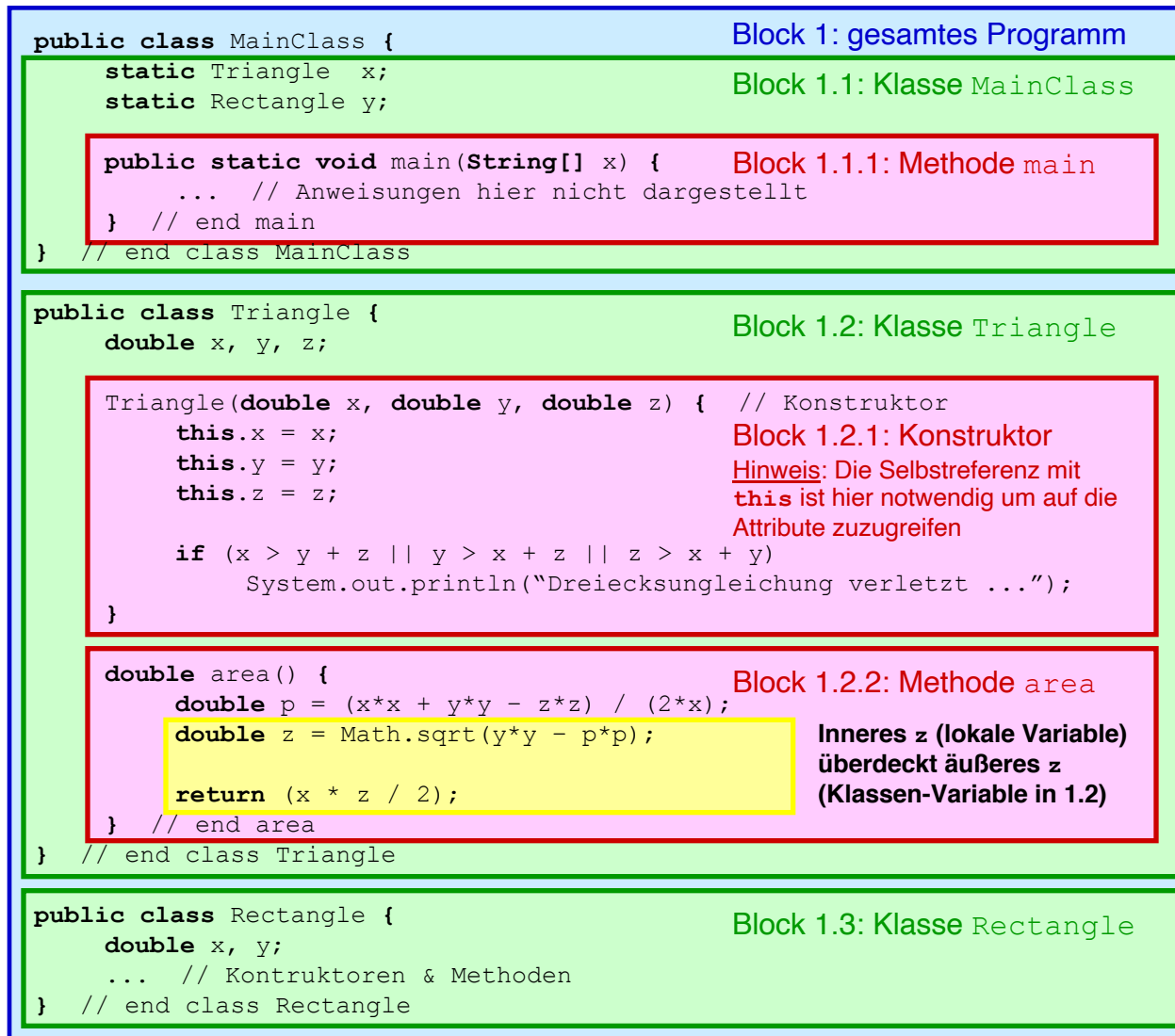
    while (y > 0) {
        int x; // nicht möglich ... Uebersetzungsfehler
        :
    }
}
```

Hinweise:

- In einer Reihe von Programmiersprachen ist es möglich, dass eine lokale Variable x innerhalb einer **while**-Schleife (oder allgemein einer Wiederholungsschleife) die lokale Variable x des Unterprogramms überdeckt, in Java geht das **nicht!**
- Diese Einschränkung gilt auch, wenn die gleichnamigen Bezeichner Daten mit unterschiedlichen Datentypen identifizieren, z.B.

```
double x;
while (...) {
    int x;
    ...}
```

Schachtelung von Blöcken strukturiert Gültigkeitsbereiche von Bezeichnern



Hinweis und Einordnung:

Die Hierarchieebenen bilden Bereiche der Gültigkeit von Bezeichnern

Praktische Überprüfung von Gültigkeitsbereichen

- Eine Variable ist **gültig** ab ihrer Deklaration bis zum Ende des Blockes, der die Deklaration enthält
- Sie ist **sichtbar** in dem Bereich, in dem man ohne vorangestellten Qualifizierer auf sie zugreifen kann
- Der **Sichtbarkeitsbereich** einer Variablen ist also ein Teilbereich ihres **Gültigkeitsbereiches**
Er unterscheidet sich dort vom Gültigkeitsbereich, wo die Variable von einer anderen Variablen überschattet wird

Bestimmung des Sichtbarkeitsbereiches einer Variablen:

- Suche zuerst die Deklaration der Variablen und den dazugehörigen umschließenden Block
- Entferne dort alle inneren Blöcke mit der Deklaration einer Variablen desselben Namens
- Übrig bleibt der gesuchte Sichtbarkeitsbereich

Gültigkeitsbereiche

```

public class Triangle {
    private double x, y, z;

    public Triangle(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
        if (x > y + z || y > x + z || z > x + y)
            System.err.println("Dreiecksungleichung verletzt");
    }

    public double area() {
        double p = (x * x + y * y - z * z) / (2 * x);
        double z = Math.sqrt(y * y - p * p);
        return (x * z / 2);
    }
}

```

Sichtbarkeitsbereiche

Erläuterungen:

- **x**, **y** und **z** sind **Objekt-Variablen**, sie sind im gesamten Bereich der Klasse gültig
- **x**, **y** und **z** sind **formaler Parameter**, sie sind innerhalb des Konstruktors gültig und überschatten dort die Objekt-Variablen gleichen Namens
- **z** ist eine **lokale Variable**, sie ist innerhalb der Methode `area()` gültig (beginnend bei ihrer Deklaration) und überschattet dort die Variable **z**

Erlaubte Namensgleichheiten

Namensgleichheiten innerhalb eines Gültigkeitsbereiches (Block) dürfen auftreten, wenn die Größen aufgrund ihrer syntaktischen Struktur eindeutig unterscheidbar sind

- Eine **Klasse**, eine **Methode** und eine **Variable** dürfen in einem Gültigkeitsbereich gleich benannt werden

Bsp.:

```
class f {
    void f() {
        int f; ...
    }
}
```

- **Methoden** mit **unterschiedlichen Parameterlisten** dürfen gleiche Bezeichner (Namen) haben – das heißt dann „**Überladen**“ von Methoden

Bsp.:

```
func(int x) { ... }
func(float x) { ... }
func(int x, int y) { ... }
```

- Ein unschönes, aber syntaktisch korrektes Beispiel ...

```
class R {
    float x, y;
}

public class TestCatastrophy {
    R R(float x, float y) {
        R r = new R();

        r.x = x;
        r.y = y;

        return r;
    } // end R

    public static void main(String[] args) {
        R[] r = new R[2];

        for (int R = 0; R < 2; R++) {
            r[R] = new R();
            r[R].x = R;
            r[R].y = R;
        }
        System.out.println(r[0].x + ", " + r[0].y);
        System.out.println(r[1].x + ", " + r[1].y);
    } // end main
} // end class TestCatastrophy
```

R ist die Bezeichnung der Klasse

Vereinbarung einer Methode R

Die Methode R liefert ein Resultat vom Referenz-Typ R

Zeiger-Variable r zeigt auf Objekt der Klasse R

Array von Objekten der Klasse R

Deklaration einer Integer Variable R

Ausgabe:

0.0, 0.0

1.0, 1.0

- Die Lesbarkeit von Programmen ist wesentlich durch die Struktur und die Wahl der Bezeichner gegeben:
 - „sprechende“ Namen für Bezeichner wählen
 - konsistente Bezeichnungsweise verwenden
 - vernünftige Längen von Bezeichnern wählen
 - einheitliche sprachliche Namensgebung (engl. ist Standard, vgl. Beispielprogramme)

Namensräume und Zugriffskontrolle

Motivation – Klassen und Pakete in Java

- *Was passiert, wenn zwei Hersteller Klassen mit demselben Namen erstellt haben und beide Klassen in einem Projekt zusammen benutzt werden sollen?*
- *Wie können logisch zusammengehörige Klassen gruppiert werden?*
- Für den Umgang mit diesen Problemstellungen gibt es **Namensräume**:
 - Logische Gruppierung zusammengehöriger Komponenten
 - Unterstützung der Modularisierung großer Programme
- Konzept in Java: **Packages**
 - Klassen sind in Java grundsätzlich in **packages** gruppiert (auch Klassen mit der `main()` Methode)
 - Falls kein **package** angegeben wird, ist dies das *Default-package*
 - **Packages** bilden einen eigenen **Namensraum für Klassen**, Namenskonflikte bei Verwendung von *Packages* verschiedener Hersteller werden vermieden

Organisation: Klassen werden in einem **eigenen Verzeichnis** mit dem Namen des **package** abgelegt – das *Package* mit seinen Klassen kann in dem darüber liegenden Verzeichnis (welches das **package**-Verzeichnis enthält) **übersetzt** werden

Packages in Java

Namenskonventionen

- Namenskonventionen für Packages
 - *Packages* der **Standard-Klassenbibliothek** beginnen mit `java` bzw. `javax`, z.B. `java.lang`
 - **Herstellerspezifische *Package*-Namen** beginnen mit dem umgekehrtem Domain-Namen des Herstellers, z.B. `com.sun.javadoc`
- Die Klassen der [Java-Klassenbibliothek](#) sind in *Packages* gruppiert; einige der wichtigsten *Packages* sind:
 - `java.lang` Grundlegende Systemklassen, z.B. `Object`, `String`, `System`, ...
 - `java.util` Hilfsklassen, z.B. `Vector`, `Date`, ...
 - `java.io` Ein-/Ausgabeklassen, z.B. `File`, `FileReader`, `FileWriter`, `InputStream`, `PrintStream`, ...
 - `java.awt` GUI-Klassen (GUI: *Graphical User Interface*)
- Dokumentation: *API Specification* (API: Application programming interface)

Verwendung von *Packages* in Java-Programmen

- Vollständige Angabe von Klassennamen

Bsp.: `java.util.Date startDate = new java.util.Date();`

Klasse	Referenzvariable	Konstruktor zur Generierung eines Objekts
(im Package <code>java.util</code>)		(als Instanz der Klasse <code>Date</code>)

- Klasse importieren (mit **import**-Anweisung zu Beginn des Quelltexts)

Bsp.: `import java.util.Date;`
 `...`
 `class MyClass {`
 `static Date startDate = new Date();`
 `...}`

- Alle Klassen eines *Packages* importieren

Bsp.: `import java.util.*;`
 `import java.awt.*;`

Problem: Kann bei mehrdeutigen Namen zu Übersetzungsfehlern führen

```
List          li1;   // Fehler: mehrdeutig!
java.awt.List li2;   // OK
```

Zugriffskontrolle durch *Scopes*

Allgemeines

- *Was ist bereits bekannt?*
 - Sichtbarkeit von Java-Bezeichnern
 - Logische Gruppierung mittels „*Packaging*“
 - Weitere Verfeinerung durch Klassenbildung – bestehend aus **Methoden** und **Attributen**
- Ausnutzung dieser Hierarchie und Sichtbarkeit für Java-Sicherheitskonzept, das sich auf die Kapselung von Informationen stützt

Konzept der **Scopes**:

- Jeder Name eines Elements einer Klasse (Attribut oder Methode) sowie auch die Klasse selbst haben einen Sichtbarkeitsbereich (**Scope**)
- Zugriffe auf ein Element (Attribut, Methode) unterliegen einer **Zugriffskontrolle** (*access control*)
- Der Name eines Elements (Attribut, Methode) kann nur dort verwendet werden, wo er sichtbar ist

Sichtbarkeitsbereiche

- In Java gibt es 4 **Sichtbarkeitsbereiche**: global, geschützt, Paket und Klasse
- Der Sichtbarkeitsbereich eines Elements wird durch einen der folgenden **Modifizierer** (*access modifiers*) angegeben:
 - **public** (öffentlich, global)
 - **protected** (geschützt; Zugriff nur von Unterklassen aus möglich, **Teil XII**)
 - **private** (Zugriff nur innerhalb der Klasse)

Wird **nichts angegeben**, gilt der Standard-Sichtbarkeitsbereich (*default scope*) *Package*

Bsp.:

```
public class Date { // der Klassen-Name ist public
    private int day;    // Feld/Attribut ist private (nur in der Klasse sichtbar)
    private int month;  // Feld/Attribut ist private (nur in der Klasse sichtbar)
    private int year;   // Feld/Attribut ist private (nur in der Klasse sichtbar)

    public void method1() {           // die Methode ist public
        ...}

    void method2() {                  // die Methode ist im Package sichtbar
        ...}

    private boolean method3() {       // die Methode ist nur in der Klasse sichtbar
        ...}
}
```

Ebene des Zugriffsschutzes

- Generell gilt in Java, dass der Zugriffsschutz auf Klassenebene definiert wird und nicht auf Objektebene

Beispiel: Eine **Methode**, die auf einem **Objekt obj1** der **Klasse A** abläuft, kann auf **alle Attribute** eines anderen Objekts **obj2** vom Typ **A** mit genau denselben Rechten zugreifen, als seien es ihre eigenen

- Die **zugreifbaren Attribute und Methoden** einer Klasse bilden die (Aufruf-) Schnittstelle (*call interface*) bzw. **Signatur der Klasse** nach außen
 - Interne Daten und Methoden bleiben verborgen und können geändert werden, ohne dass extern davon etwas zu merken ist
 - Realisierung eines Geheimnisprinzips (*Information hiding*)
- Die **Syntax und Semantik der Aufrufschnittstelle** bilden den **Vertrag** (*contract*) zwischen den Programmierern der Klasse und ihren Nutzern
 - Solange der Vertrag beachtet wird, funktioniert die Zusammenarbeit zwischen den Nutzern der Klasse und dem Code der Klasse selbst
 - Die Art und Weise, wie die Klasse ihre Seite des Vertrags erfüllt, ist unerheblich und kann wechselnden Erfordernissen angepasst werden

- In bestimmten Fällen kann es nötig werden, weitere Zusicherungen in den Vertrag aufzunehmen, z. B. Leistungsdaten
 - wie die Komplexität von implementierten Algorithmen oder
 - garantierte Reaktionszeiten im Realzeitbereich
- Es ist offensichtlich, dass der Vertrag entsprechend sorgfältig **dokumentiert** werden muss; hierzu gehören insbesondere ...
 - die **Spezifikation der Methoden**
 - eine **Beschreibung der Gesamtleistung der Klasse**
- **Werkzeuge** wie **javadoc** unterstützen sowohl die Spezifikation der Methoden als auch die Beschreibung der Gesamtleistung der Klasse
 - Ein **Dokumentationskommentar** (`/** ... */`), der unmittelbar vor einer Klasse steht, wird von **javadoc** als **Beschreibung der Gesamtleistung** der Klasse interpretiert
 - Da private Felder und Methoden nicht zur Aufrufschnittstelle gehören, werden diese (und deren Dokumentation) von **javadoc** (in der Standardeinstellung) nicht mit in die erzeugte Dokumentation der Klasse aufgenommen

Zusammenfassung der Sichtbarkeitsregeln

■ Übersicht

Spezifikation	Klasse	Unterklasse ^(*)	<i>Package</i>	Alle
private	X			
protected	X	X	X	
public	X	X	X	X
package	X		X	

(*) Unterklassen wurden bisher noch nicht behandelt; diese werden bei der Bildung von Hierarchien in Objektklassen und der Vererbung relevant

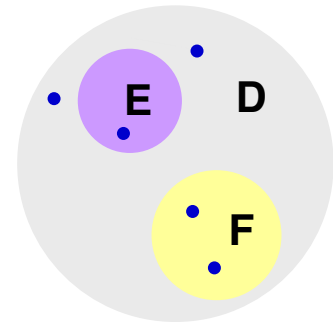
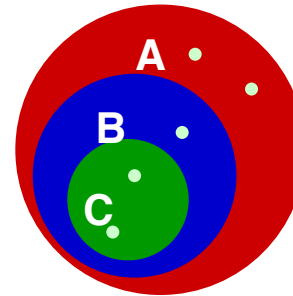
■ Beispiele für die Rolle der Modifizierer:

- Auf ein als **private** deklariertes Attribut kann nur die Klasse (oder deren Exemplare, d.h. Instanzen) zugreifen, in der es definiert wurde
- Die `main()` Methode wird als **public** deklariert, so dass damit das Programm überall aufgerufen und ausgeführt werden kann

Behandlung von Programmausnahmen (*Exceptions*)

Motivation – Fehlerquellen und Fehlerbehandlung

- Zur **Erinnerung** (vgl. **Teil II**): Ein Programm (Algorithmus) ist korrekt, wenn es für jede Eingabe aus dem Definitionsbereich der Funktion die korrekte Ausgabe erzeugt – und dabei im Fehlerfall (Menge **B** – **C**) Eingaben „sinnvoll“ behandelt



- Beispiel:

```
public class MyMath {
    public static int fak(int n) {
        if (n > 0)
            return n * fak(n-1);
        else
            return 1; // falsch, weil n < 0 (nicht def.)
    }

    public static double div(double num, double denom) {
        if (denom != 0.0)
            return num/denom;
        else
            return 0.0; // eigentlich falsch (unendlich)
    }
} // end class MyMath

public class MyComputations {
    public static void main(String[] args) {
        double eingabe = TextIO.getlnDouble(); // Funktion aus TextIO
        System.out.println(fak(eingabe));
        System.out.println(div(eingabe, eingabe-2.0));
    }
} // end class MyComputations
```

■ *Was gibt es für Fehler und wie werden sie behandelt?*

1. „Schwere“ Fehler → Programmabbruch
2. „Leichte“ Fehler → Korrektur

Einordnung: Grad der Schwere des Fehlers ist unter Umständen situationsabhängig!

Bsp.: Division durch 0

- Programmabbruch oder
- Aufforderung zur Neueingabe von Werten

■ Weitere Frage: *Wer soll auf Fehler reagieren?*

- Softwareentwickler einer Methode
- Aufrufende Umgebung einer Methode

■ Fehlerquellen

- Ungültiger *Array*-Index
- Zugriff auf `null`-Objekt
- Arithmetische Fehler
- Kein Speicherplatz mehr vorhanden
- Inkonsistente Werte von Variablen
- Ungültige Parameterwerte
- Falsche Adressen / Dateinamen
- Nicht erfüllte Vorbedingungen (Lesen erst nach Öffnen einer Datei)
- ...

■ *Warum programmseitige Ausnahmebehandlung?*

- Programme sollen robuster gemacht werden gegen fehlerhafte Eingaben, Programmierfehler (z.B. bei Division durch 0), etc.
- **Zielsetzung(en):**
 - **Trennung** zwischen Algorithmus und Fehlerbehandlung
 - Die **Fehlerbehandlung** soll **durch den Compiler** überprüfbar gemacht werden
- Die angebotenen Möglichkeiten zur Ausnahmebehandlung sind stark programmiersprachenabhängig
- Nachfolgend werden einige ausgewählte Konzepte der programmseitigen Ausnahmebehandlung am Beispiel von Java erörtert

Anmerkung: Hier erfolgt zunächst eine sehr stark vereinfachte Darstellung, mehr Details zur Ausnahmebehandlung in Java (z.B. die Definition eigener Fehlerklassen) werden in der Vorlesung „**Programmierung von Systemen**“ vorgestellt

Fehler- und Ausnahmebehandlung in Java

- Zunächst wird „versucht“, die Methode auszuführen

Falls **kein Fehler** auftritt, normal weitermachen ...

Falls **ein Fehler** auftritt, wird ein Ausnahme-/Fehlerobjekt erzeugt,
dieses wird abgefangen und eine geeignete Fehlerbehandlung eingeleitet

- **Explizite Ausnahmen:** eine `throw`-Anweisung im Programmcode erzeugt ein Ausnahme-/Fehlerobjekt

Bsp.:

```
String passwd = TextIO.getln();
...
if (passwd.length() < 5)
    throw new Exception("mind. 5 Zeichen");
...
```

- **Implizite Ausnahmen:** durch die *Java Virtual Machine* (JVM)
 - Division durch 0
 - Zugriff über `null`-Referenz
 - Ungültige Konvertierung

Bsp.:

```
int divideInt(int arg1, int arg2) {
    return arg1 / arg2;
}
```

Aufruf von `divideInt(1, 0)`
resultiert in einer **Ausnahme**

try und catch in Java

Fehlerbehandlungen

- Aufbau von **expliziten Fehlerbehandlungen**

```
try {  
    <Anweisungen>;  
}  
catch (<ExceptionSorte1> <Ausdruck1>) {  
    <Anweisungen-Fehlerbehandlung>;  
}  
catch (<ExceptionSorte2> <Ausdruck2>) {  
    <Anweisungen-Fehlerbehandlung>;  
}  
...  
finally {  
    <Anweisungen>;  
}
```

Normaler Block mit Anweisungen
bzw. Anweisungsfolgen

Nach einem **try**-Block muss
mindestens ein **catch**-Block
folgen!

Erläuterungen:

- Bei **mehreren catch-Blöcken** beginnt die Suche der **Ausnahmebehandlung von oben nach unten**
- Die Angabe eines **finally-Blocks** ist **optional**; wenn er auftritt, dann wird er immer ausgeführt

■ Beispiel in Java (Demo: [Catch.java](#))

```
class Catch {

    public static void main(String[] args) {

        try {
            int i;

            i = Integer.parseInt(args[0]);
            System.out.print("i = " + i);
        }
        catch (ArrayIndexOutOfBoundsException except) {
            System.out.print("Parameter vergessen ... ");
        }
        catch (NumberFormatException except) {
            System.out.print("Kein int-Wert ... ");
        }
        finally {
            System.out.println(" finally !");
        }
    } // end main
} // end class Catch
```

Testaufrufe:

```
java Catch 123
```

Ausgabe: i = 123 finally !

```
java Catch xy
```

Ausgabe: kein int-Wert ... finally !

```
java Catch
```

Ausgabe: Parameter vergessen ... finally !

Verwendung der throw-Klausel

- Beispiel

```
setPassword(String pw) {
    ...
    try {
        if (pw.length() < 5) {
            throw new Exception("Fehler");
        }
    }
    catch (<ExceptionSorte> <Ausdruck>) {
        <Anweisungen>;
    }
}
```

- Alternative: Weitergabe der *Exception* an den Aufrufer

Bsp.:

```
try {
    setPassword(pw);
}
catch (Exception except) {
    <Anweisungen>;
    // Ausnahme behandeln
}
```

```
static setPassword(String pw)
    throws Exception {
    if (pw.length() < 5) {
        throw new Exception("Fehler");
    }
    ...
}
```

■ Beispiel für die Weitergabe von *Exceptions*

```
main(...) {
    ...
    try {
        A(x);
    }
    catch (Exc1 e) {
        ...
        print("main");
    }
    ...
}
```

```
...
A(int x)
    throws Exc1 {
    try {
        B(x);
    }
    catch (Exc2 e) {
        ...
        println("A");
    }
    finally {
        ...
        println("finA");
    }
    ...
}
```

```
...
B(int x)
    throws Exc1, Exc2 {
    try {
        C(x);
    }
    catch (Exc3 e) {
        ...
        println("B");
    }
    finally {
        ...
        println("finB");
    }
    ...
}
```

```
...
C(int x)
    throws Exc1, Exc2, Exc3 {
    try {
        if (x == 1) {
            throw new Exc1("F.");
        }
        else if (x == 2) {
            throw new Exc2();
        }
        else if (x == 3) {
            throw new Exc3();
        }
        else {
            throw new Exc4();
        }
    }
    catch (Exc4 e) {
        ...
        println("C");
    }
}
```

Ergebnisse ...

x = 1

finB
finA
main

x = 2

finB
A
finA

x = 3

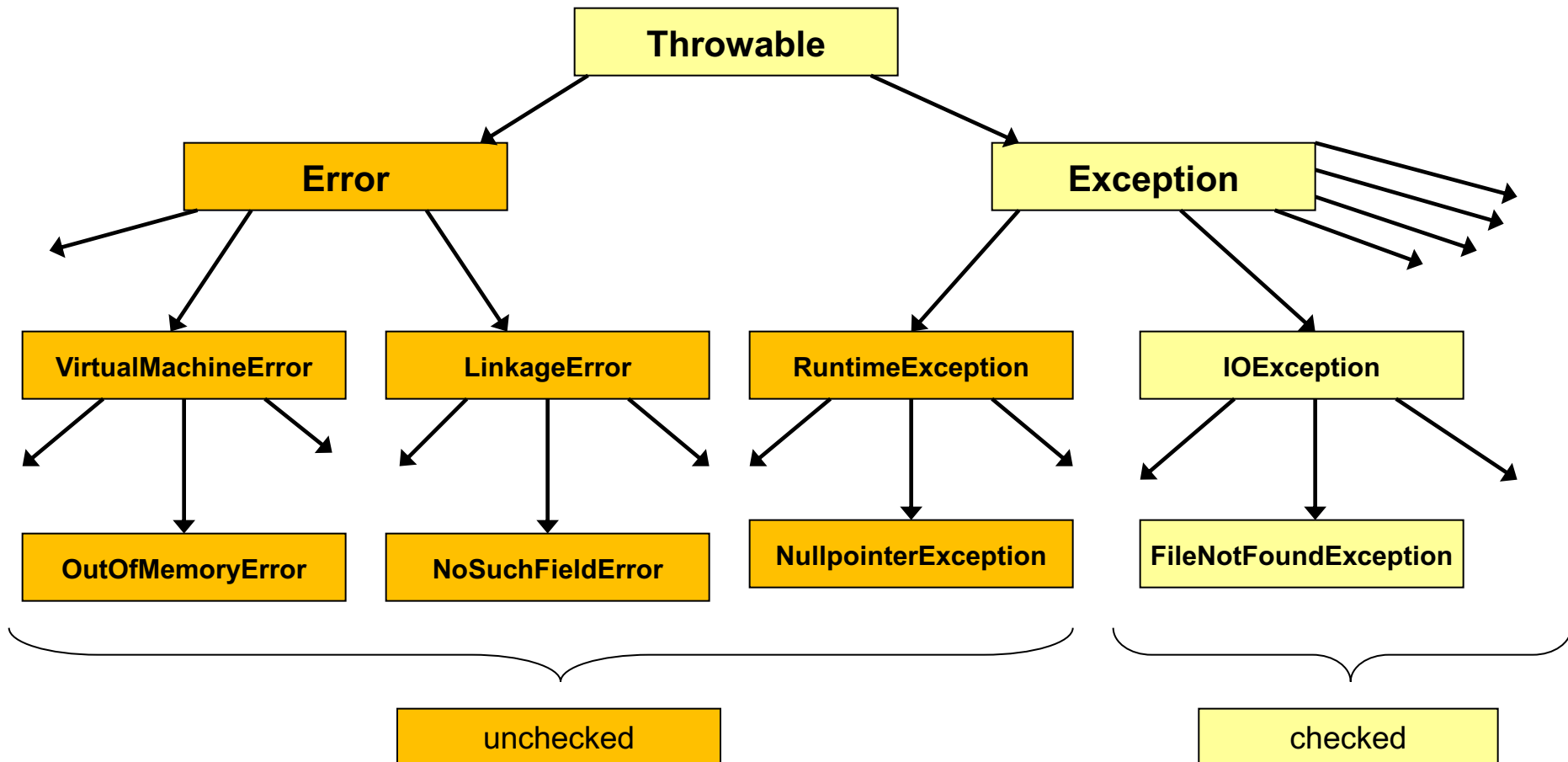
B
finB
finA

x = 4

C
finB
finA

Hierarchien und Fehlerarten

- Übersicht über die Hierarchie der Ausnahmebehandlung in Java



- Die **Error**-Klasse

VirtualMachineError

InternalError

OutOfMemoryError

StackOverflowError

UnknownError

LinkageError

NoSuchMethodError

NoSuchFieldError

AbstractMethodError

ThreadDeath

Ausnahmen des Typs **Error** sind
nicht vernünftig zu behandeln

- Die **Exception**-Klasse

IOException

FileNotFoundException

SocketException

EOFException

RemoteException

UnknownHostException

InterruptedException

NoSuchFieldException

ClassNotFoundException

NoSuchMethodException

Ausnahmen des Typs
Exceptions sollten behandelt
werden

(ausgenommen
RuntimeExceptions)

Zusammenfassung

- Wird eine **außergewöhnliche Situation zur Laufzeit** entdeckt, so wird eine Ausnahme ausgelöst
- Programmseitig werden zur Behandlung von Ausnahmen in Java so genannte **try-catch-Blöcke** benutzt:
 - Wir umgeben ein Codefragment, das eine Ausnahme auslösen könnte, mit `try { . . . }` und
 - stellen zugehörige `catch`-Blöcke zur Verfügung
- Ein **geprüfte Ausnahme** muss bearbeitet werden; wird dies unterlassen, ergibt sich ein Übersetzungsfehler!
- Die **Klassenhierarchie Throwable** fasst die wichtigsten Ausnahmen zusammen, mit denen man es bei der Programmierung von Systemen zu tun hat und zeigt, welcher Klasse sie angehören
 - Ausnahmen innerhalb der Klasse `RuntimeException` sind schwierig zu bearbeiten und können in vielen Fällen bewusst ignoriert werden
 - Alle anderen Ausnahmen unterhalb der Klasse `Exception` werden überprüft und sollten bearbeitet werden