

# KLASSENMODELLIERUNG 2: SICHTBARKEIT UND VERERBUNG

**Komplexe Zusammenhänge durch Klassenhierarchien darstellen.**

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt.  
Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am  
Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo  
erstellt und können unter der [MIT-Lizenz](#)  verwendet werden.

# AKTUELLER SEMESTERFORTSCHRITT (WOCHE 6)

Kursabschnitt	Themen		
Grundlagen	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife	
Klassenmodellierung	Grundlagen der Klassenmodellierung		
Interaktive Anwendungen	Event-basierten Programmierung		
Datenstrukturen	Listen, Maps & die Collections		
Software Engineering	Planhaftes Vorgehen bei der Softwareentwicklung		
	Qualitätsaspekte von Quellcode		

# PINGO-QUIZ



<https://pingo.coactum.de/567100> 

# DAS PROGRAMM FÜR HEUTE

- 5-Minuten-Themen: Objekte als Parameter verwenden und die Dokumentation von Klassen
- Klassen und Vererbung
- Klassen und Komposition

# OBJEKTE ALS PARAMETER

# CAVEAT: OBJEKTE ALS PARAMETER

Objekte verhalten sich anders als primitive Datentypen, wenn Sie als (Übergabe-)Parameter eingesetzt werden:

```
1 private void modifyStudent(Student student, int lp) {  
2     System.out.println("Modifying Students LPs");  
3     student.addLP(lp);  
4 }
```

**Hinweis:** Objekte werden **nicht** als Kopien weitergegeben sondern als Verweise (Referenzen) auf die *Originale*. Wenn wir ein Objekt (Parameter) in einer Methode ändern, ändern wir das Original. Technisch gesehen wird auch hier konkreter Wert, nämlich die "Adresse" des übergebenen, Java arbeitet immer mit dem *Call by Value*-Prinzip.

## KONSEQUENZEN DES *CALL BY VALUE*-ANSATZES IN JAVA (1/2)

- Wird ein Objekt an eine Methode übergeben, befindet sich in der Parameter-Variable die Adresse (im Speicher) an der das Objekt bzw. seine Methoden (nicht wirklich) und Eigenschaften gespeichert (die schon) werden.
- Wenn wir die Eigenschaften des Objekts über die Parameter-Variable ändern, ändern wir das ursprüngliche Objekt.
- Wenn wir die Parameter-Variable überschreiben, bleibt das ursprüngliche unberührt (wir ändern dann nur einen von mehreren der Orte, an denen die Adresse des Objekts gespeichert ist).

## KONSEQUENZEN DES *CALL BY VALUE*-ANSATZES IN JAVA (2/2)

```
1 /**
2  * LPs werden für das äußere Objekt gesetzt, die Initialisierung des neuen Objekts
3  * ist aber auf den Scope beschränkt.
4 */
5 private void modifyStudent(Student student, int lp) {
6     System.out.println("Modifying Students LPs");
7     student.addLP(lp);
8     student = new Student("A new User", 4.0);
9 }
```

**Hinweis:** Wir greifen dieses Thema im späteren Verlauf der Vorlesung beim Thema *Speicherverwaltung* noch einmal auf.

# KLASSEN MIT JAVADOC KOMMENTIEREN UND DOKUMENTIEREN

# KOMMENTARE IN JAVA

Kommentare dienen der zusätzlichen Beschreibung von Quelltext. Sie haben keine funktionale sondern eine dokumentierende Bedeutung:

```
1
2 /**
3  * Diese Methode erzeugt ein zufälliges Circle-Objekte (zufällige Position und Farbe) und
4  * gibt die Instanz als Rückgabe wert zurück.
5 */
6 public Circle createRandomCircle() {
7     // createRandomNumber gibt eine zufällige Ganzzahl zwischen 0 und dem übergebenen
8     // Parameter zurück
9     int xPos = createRandomNumber(SCREEN_WIDTH);
10    int yPos = createRandomNumber(SCREEN_HEIGHT);
11    Color circleColor = Colors.getRandomcolors();
12    // Erzeugen des Circle-Instanz (CIRCLE_RADIUS ist eine Konstante auf Klassenebene)
13    Circle circle = new Circle(xPos, yPos, CIRCLE_RADIUS, circleColor);
14    return circle;
15 }
```

# KOMMENTARE MIT JAVADOC STRUKTURIEREN

```
1 /**
2  * Diese Methode erzeugt ein zufälliges Circle-Objekt innerhalb der
3  * Zeichenfläche, dessen Radius im übergebenen Wertebereich liegt.
4  *
5  * @param minRadius Minimaler Radius des zu erstellenden Kreises (inklusive)
6  * @param maxRadius Minimaler Radius des zu erstellenden Kreises (exklusive)
7  * @return Die erzeugte Circle-Instanz
8 */
9 public Circle createRandomCircleWithColor(int minRadius, int maxRadius) {
10     int xPos = createRandomNumber(SCREEN_WIDTH);
11     int yPos = createRandomNumber(SCREEN_HEIGHT);
12     int radius = minRadius + createRandomNumber(maxRadius - minRadius);
13     return new Circle(xPos, yPos, radius, Colors.getRandomcolors());
14 }
```

**Hinweis:** Das **JavaDoc-Format** erlaubt die gezielte Beschreibung einzelner Code-Bestandteile (z.B. Parameter oder Rückgabewerte). Auf Basis der Kommentare kann eine Übersicht (Dokumentation) des Programms erstellt werden kann (Vgl.: **GraphicsApp-Dokumentation** .

# RÜCKBLICK: KLASSENMODELLIERUNG AM BEISPIEL STUDIERENDER

# WIR MODELLIEREN STUDIERENDE ALS JAVA-KLASSEN

Die Klasse Student soll Gemeinsamkeiten aller Studierende abbilden. Ein möglicher Einsatzzweck ist ein Computersystem zur Verwaltung aller Studieren einer Universität. Wir modellieren Klassen (in der Regel) spezifisch für einen Anwendungsfall. D.h. wir beachten und beschreiben die Eigenschaften der Objekte, die relevant für den Problemkontext sind. Unsere Student-Klasse umfasst z.B. Name, Matrikelnummer, Durchschnittsnote und aktuelle Anzahl an Leistungspunkten, da diese im Kontext *Universität*, z.B. für die Prüfungsverwaltung relevant sind.

Wir beschreiben **nicht** alle denkbaren, strukturellen Übereinstimmungen von Studierenden (*Hobbies, Sehstärke, Lieblingsessen, ...*).

## INTERNER BEREICH (PRIVATE-SICHTBARKEIT)

Private Methoden, Konstanten und Instanzvariablen: Nur die Instanzen der Klassen selbst haben Zugriff.

- Instanzvariable für den Namen
- Instanzvariable für die aktuelle Durchschnittsnote
- Instanzvariable für eine ID, z.B. die Matrikelnummer
- Instanzvariable für die aktuelle Anzahl an Leistungspunkten

## EXTERNE SCHNITTSTELLE (PUBLIC-SICHTBARKEIT)

Private Methoden, Konstanten und Instanzvariablen: Nur die Instanzen der Klassen selbst haben Zugriff.

- Getter-Methode für Zugriff auf den Namen
- Getter-Methode für Zugriff auf die aktuelle Durchschnittsnote
- Getter-Methode für Zugriff auf die ID
- Getter-Methode für Zugriff auf die aktuelle Anzahl an Leistungspunkten
- Setter-Methode zum Hinzufügen neuer Leistungspunkte

# DAS ERGEBNIS: EIGENSCHAFTEN UND KONSTRUKTOR DER KLASSE

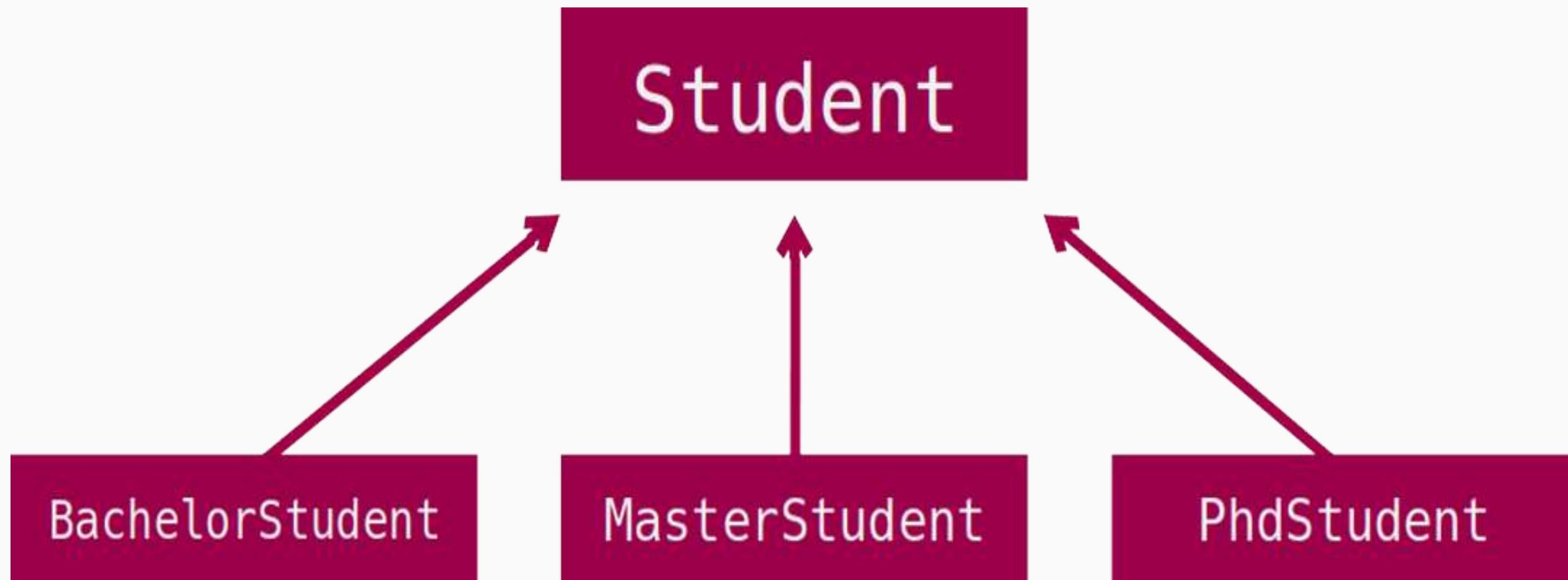
```
1 public class Student {  
2     private String name;  
3     private double grade;  
4     private int id;  
5     private int lp;  
6  
7     private static int nextID = 1;  
8  
9     public Student(String name, double grade) {  
10        this.name = name;  
11        this.grade = grade;  
12        this.lp = 0;  
13        this.id = nextID;  
14        nextID++;  
15    }  
16  
17    // Siehe nächste Folie  
18 }
```

# DAS ERGEBNIS: ÖFFENTLICHE SCHNITTSTELLE DER KLASSE

```
1 public class Student {  
2  
3     // siehe vorherige Folie  
4  
5     public String getName() {  
6         return name;  
7     }  
8     public double getGrade() {  
9         return grade;  
10    }  
11    public int getID() {  
12        return id;  
13    }  
14    public int getLP() {  
15        return lp;  
16    }  
17    public void addLP(int lp) {  
18        this.lp += lp;  
19    }  
20 }
```

# KLASSEN UND VERERBUNG

# KLASSENHIERARCHIEN



Der Vererbungsmechanismus erlaubt uns das hierarchische Gliedern von Klassen und die Modellierung von Gemeinsamkeiten und Spezialisierungen.

# SUPER- UND SUBKLASSEN

Student ist die **Superklasse** (auch *Oberklasse*). BachelorStudent, MasterStudent und PhdStudent erben von dieser Klasse: Sie sind **Subklassen** (auch *Unterklassen*) von Student. Dabei übernehmen Sie die Eigenschaften und Methoden der *Superklassen*. In den *Subklassen* können (öffentliche) Methoden der *Superklasse* überschrieben werden und neue Methoden und Eigenschaften ergänzt werden.

**Hinweis:** Eine Subklasse ist eine Erweiterung oder **Spezialisierung** der Superklasse. Umgekehrt kann man eine Oberklasse als **Generalisierung** ihrer Unterklassen ansehen.

## VERERBUNG IM CODE EINLEITEN

```
1 public class BachelorStudent extends Student {  
2  
3     // Spezialisierte Implementierung für Bachelor-Studierende  
4  
5 }
```

Über das Schlüsselwort `extends` wird die Vererbung eingeleitet (Vgl. `GraphicsApp` und `BouncerApp`). Im *Body* der Klasse kann dann auf alle **öffentlichen** Variablen und Methoden der Superklasse zugegriffen werden.

**Hinweis:** *Private* Methoden und Eigenschaften werden nicht vererbt. Vererbte, öffentliche Methoden, die auf solche Bereiche zugreifen, funktionieren auch in der UnterkLASSE. Der direkte Zugriff auf z.B. eine *private int*-Variable der *Superklasse* in der UnterkLASSE **nicht** möglich.

# IMPLIZITE VERERBUNG IN JAVA (1/2)

Alle Klassen in Java (auch unsere eigenen) werden automatisch in eine bestehende Klassenhierarchie eingebunden. Deren Ausgangspunkt ist die Klasse `Object`. Lassen wir unsere Klassen nicht explizit von einer anderen Klasse erben, erstellen wir eine direkte Unterklasse von `Object`. Da Vererbung auch über mehrere Hierarchieebenen funktioniert, steht am Anfang einer beliebig langen Vererbungskette immer die `Object`-Klasse: "*Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.*" (Quelle: Oracle-Dokumentation (Java 10)).

Dadurch verfügt jede Klasse bzw. jedes Objekt über bestimmte vorgegebene Methoden.

# IMPLIZITE VERERBUNG IN JAVA (2/2)

## Method Summary

All Methods	Instance Methods	Concrete Methods	Deprecated Methods
Modifier and Type	Method	Description	
protected Object	<code>clone()</code>	Creates and returns a copy of this object.	
boolean	<code>equals(Object obj)</code>	Indicates whether some other object is "equal to" this one.	
protected void	<code>finalize()</code>	<b>Deprecated.</b> The finalization mechanism is inherently problematic.	
<code>Class&lt;?&gt;</code>	<code>getClass()</code>	Returns the runtime class of this Object.	
int	<code>hashCode()</code>	Returns a hash code value for the object.	
void	<code>notify()</code>	Wakes up a single thread that is waiting on this object's monitor.	
void	<code>notifyAll()</code>	Wakes up all threads that are waiting on this object's monitor.	
<code>String</code>	<code>toString()</code>	Returns a string representation of the object.	
void	<code>wait()</code>	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> .	
void	<code>wait(long timeout)</code>	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed.	
void	<code>wait(long timeout, int nanos)</code>	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed.	

# VERERBUNG ALS MITTEL DER SPEZIALISIERUNG

Oft wollen Sie eine vorhandene Klasse mit deren öffentlicher Schnittstelle als Grundlage für eine neue, spezialisierte Variante nutzen.

## Beispiel: Master-Studierende

Wir möchten eine spezielle Klasse für die Verwaltung von Master-StudentInnen erstellen:

- Master-StudentInnen soll alles können, was StudentInnen können
- Über 180 Leistungspunkte aus dem BA-Studium verfügen
- In der `toString()`-Methode angeben, dass es sich um einen bzw. eine Master-StudentIn handelt

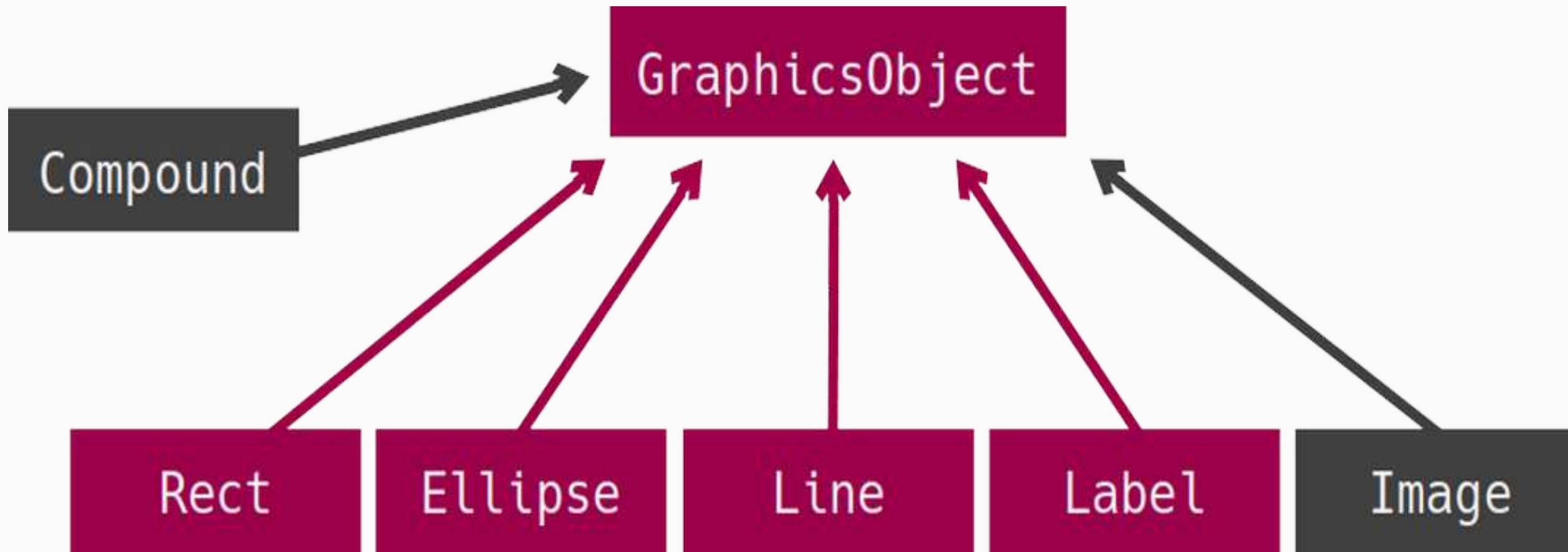
**Hinweis:** Bei der Frage, ob eine neue Komponente in eine bestehende Klassenhierarchie eingegliedert werden soll, hilft der "is a"-Test: *MasterStudent is a Student* (Unterklasse *is a* Oberklasse)? Wenn die Frage mit *ja* beantwortet werden kann, ergibt die Spezialisierung hier Sinn.

## DIE KLASSE MASTERSTUDENT

```
1 // Unsere Klassen erweitert/erbt von die/der Klasse Student
2 public class MasterStudent extends Student {
3
4     public MasterStudent(String name, double grade) {
5         // Wir verwenden den vererbten Konstruktor für die allgemeine Initialisierung
6         super(name, grade);
7         // Dann erfolgt die spezielle Initialisierung
8         addLP(180);
9     }
10
11    // Wir überschreiben die schon vorhandenen Methode toString
12    // @Override ist eine optionale Anmerkung für die Laufzeitumgebung
13    @Override
14    public String toString() {
15        // Das Schlüsselwort super verweist immer auf die Superklasse und
16        // erlaubt Zugriff auf deren Methoden und Eigenschaften
17        return super.toString() + " (MasterStudent)";
18    }
19 }
```

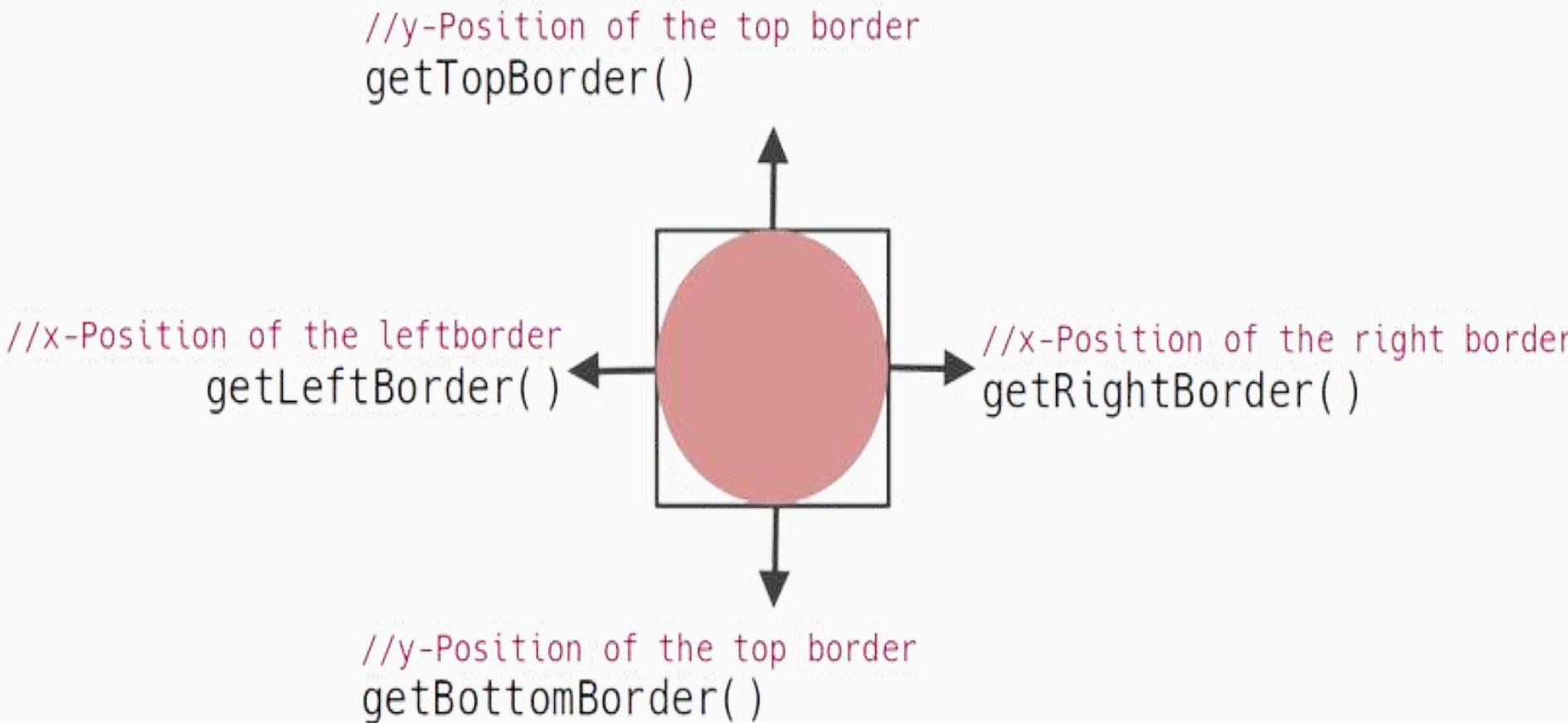
# KLASSEN UND KOMPOSITION

# VERERBUNG IN DER GRAPHICSAPP



Alle Subklassen erben von `GraphicsObject` und nutzen (oder überschreiben) bestehende Methoden dieser *Superklasse*. Der Umgang mit ihnen ist daher ähnlich, da die öffentliche Schnittstelle von der Superklasse übernommen wird: `getXPos()`, `getYPos()`, ...

## GRAPHICSAPP-HINWEISE: BOUNDING BOX



**Hinweis:** Alle *GraphicsObjects* verfügen über Methoden, um die Ränder eines Rechtecks (die *Bounding Box*) zu ermitteln, welches das Objekt umschließt. Diese Box kann sinnvoll für eine (angenäherte) Kollisionsabfrage genutzt werden.

## GRAPHICSAPP-HINWEISE: BILDER

- Bilder werden in der *GraphicsApp* über die Klasse `Image` repräsentiert und werden (Vgl. *Vererbung*) wie die anderen *GraphicsObjects* positioniert und bewegt.
- Beim Erzeugen des Bildes (Konstruktor) muss der relative Pfad zur Ausgangsdatei angegeben werden. Nach dem Erzeugen hat das Bild immer die Größe der Ursprungsdatei, kann aber skaliert werden.
- Die Ausgangsdateien werden in einem Ordner im Projektverzeichnis gespeichert und von dort referenziert: `Image bird = new Image(0, 0, "data/bird.png");`

# KOMPOSITION VS. VERERBUNG (1/3)

Komposition beschreibt den Vorgang, einer Klasse Instanzen anderer Klassen als *Member* bzw. Eigenschaft zur Verfügung zu stellen. Intern kann dann auf die Funktionalität der Klassen zugegriffen werden. Komposition ist häufig eine sinnvollere Alternative zur Vererbung.

***Favor Composition over Inheritance!***

# KOMPOSITION VS. VERERBUNG (2/3)

BouncingBall-Klasse

# KOMPOSITION VS. VERERBUNG (3/3)

Ist Vererbung (Ellipse) hier praktikabel? Nein, da das Zeichnen des Kreises nur eine Teilaufgabe ist. Unser BouncingBall wird als eigene Klasse definiert, die Instanzen von Ellipse, Rectangle und Label nutzt (*Komposition*). Der *has a*-Test zeigt schnell, das Komposition die besser Wahl ist.

# KOMPOSITION IN DER GRAPHICSAPP: COMPOUND

- Das Compound erlaubt es, mehrere graphische Objekte zu kombinieren, sodass sie sich wie ein *GraphicsObject* verhalten und als Einheit manipuliert werden können.
- Objekte werden dem Compound über Methode add() bzw. addRelative() hinzugefügt.
- Die Menge der im Compound gespeicherten Objekte kann damit als einzelnes *GraphicsObject* behandelt werden: z.B. sorgt ein Aufruf der draw-Methode des Compound dafür, dass alle Objekte gezeichnet werden.

# ZUSAMMENFASSUNG

- Objekte verhalten sich anders als primitive Datentypen, wenn Sie als Parameter von Methoden verwendet werden.
- Mittels Vererbung lassen sich bestehende Klassen erweitern – Methoden und Variablen der Superklasse mit Sichtbarkeit public stehen der Subklasse zur Verfügung
- Overriding bezeichnet das überschreiben bestehender Methoden in der Subklasse

## VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo ([alexander.bazo@ur.de](mailto:alexander.bazo@ur.de)). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach ([florin.schwappach@ur.de](mailto:florin.schwappach@ur.de)). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter [mioop@mailman.uni-regensburg.de](mailto:mioop@mailman.uni-regensburg.de).

## QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.