universität innsbruck

Vererbung und Polymorphie

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Vererbung allgemein

Vererbung allgemein

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

Unterklassen und Oberklassen

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

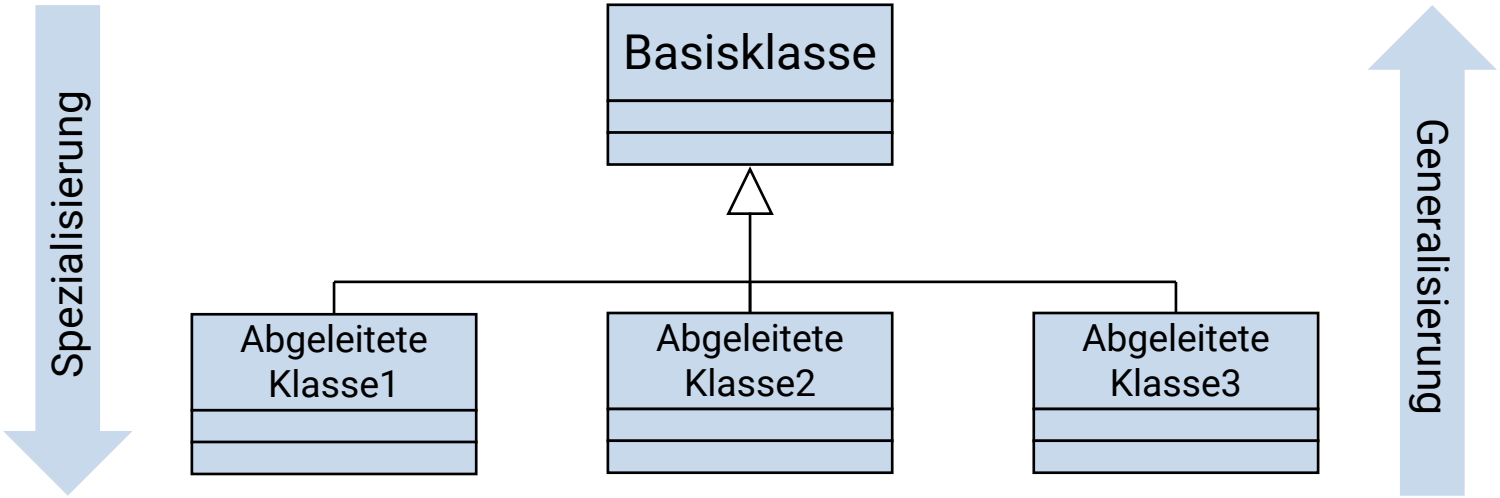
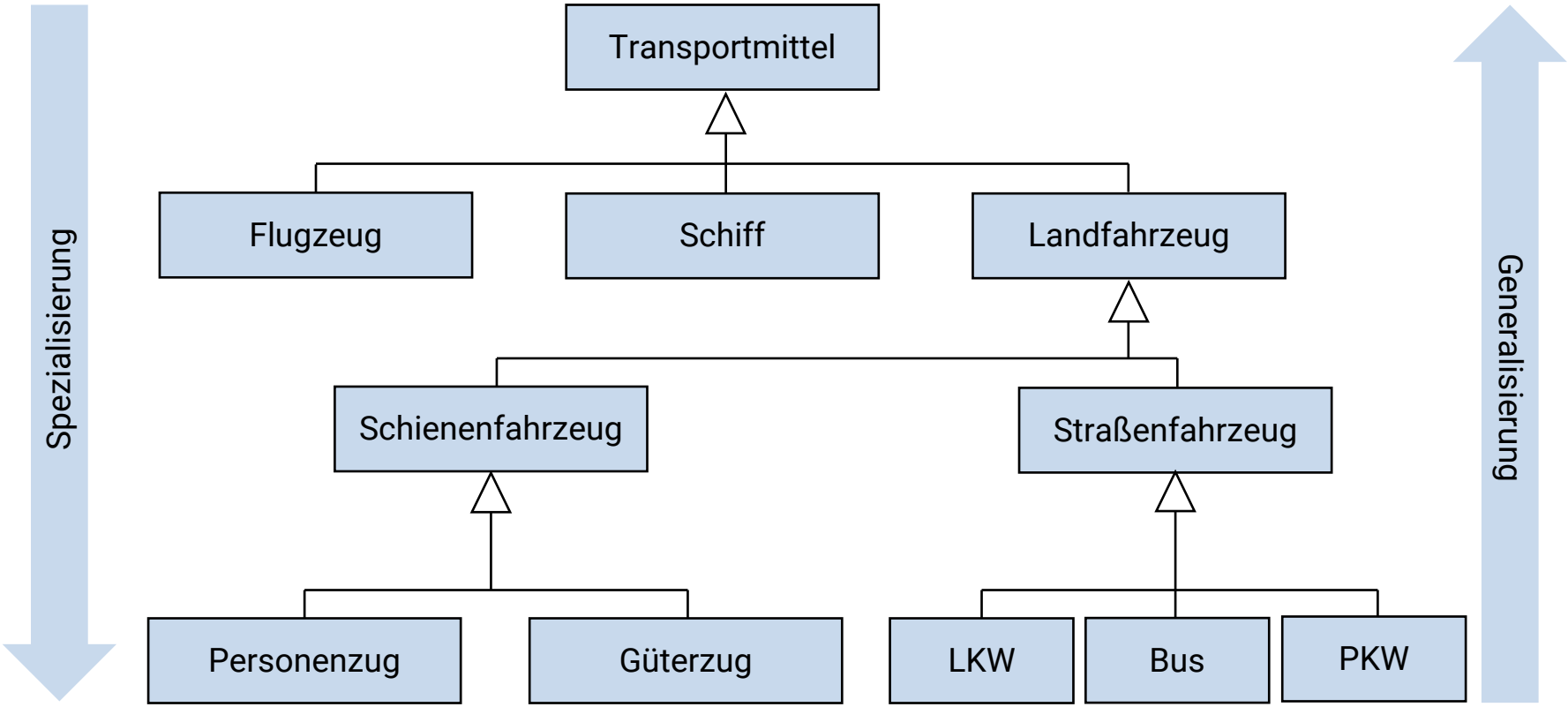


Abbildung angelehnt an „Methoden und Architekturen der Softwaretechnik“, Goll

Beispiel


Spezialisierung
Generalisierung
Transportmittel
Flugzeug
Schiff
Landfahrzeug
Schienenfahrzeug
Straßenfahrzeug
Personenzug
Güterzug
LKW
Bus
PKW

Prinzip der Ersetzbarkeit

- Wenn eine Klasse **B** eine Unterklasse der Klasse **A** ist, dann können in einem Programm alle Exemplare der Klasse **A** durch Exemplare der Klasse **B** ersetzt werden und es gelten weiterhin alle zugesicherten Eigenschaften der Klasse **A**.
  - Exemplare der Unterklasse sind gleichzeitig Exemplare der Oberklasse in Bezug auf die der Oberklasse zugrunde liegende Spezifikation.

- Konsequenzen für Unterklassen
  - Aus der Oberklasse stammende Vorbedingungen für Operationen können nicht verschärft werden, sie dürfen lediglich eingehalten oder abgeschwächt werden.
  - Aus der Oberklasse stammende Nachbedingungen für Operationen dürfen nicht gelockert werden, sie dürfen lediglich eingehalten oder verschärft werden.
  - Aus der Oberklasse stammende Invarianten müssen immer eingehalten werden.

Definition übernommen aus „Objektorientierte Programmierung“, Lahres et al.

Kategorien von Klassen

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

Konkrete Klassen

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

Schnittstellenklassen

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

Abstrakte Klassen

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

Polymorphie

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

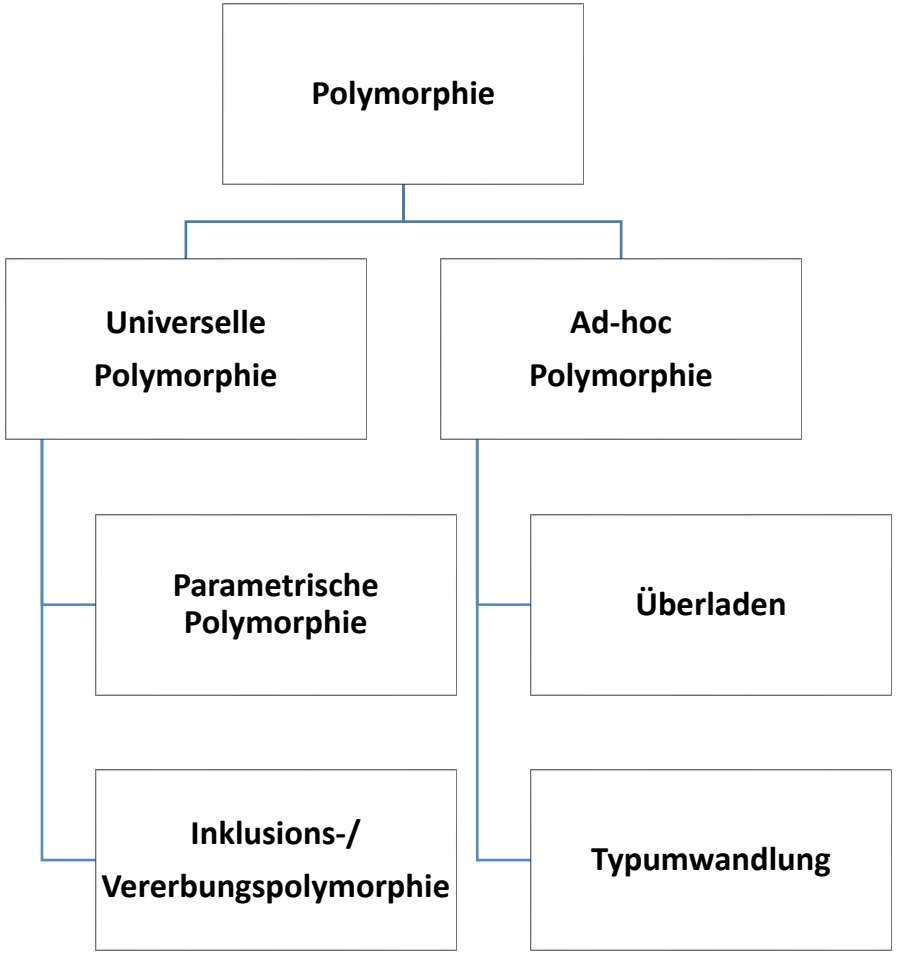


Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

Polymorphie (Arten)

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

Polymorphie in dieser Vorlesung

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

Vererbungspolymorphie

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

Vererbung der Spezifikation in Java

Interfaces (1)

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

Interfaces (2)

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

Beispiel Komplexe Zahlen (1)

- Einfaches Interface für komplexe Zahlen

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

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


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


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


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

Beispiel Komplexe Zahlen (2)

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

Polymorphie

- Interfaces definieren einen Datentyp.
  - Können in Variablendeklarationen, Parameterlisten, und als Ergebnistyp von Methoden verwendet werden.

```
Complex complexNumber = new Polar(3, 5);
```

- Warum funktioniert das Beispiel?
  - **Vererbungspolymorphie**
  - Die Klasse `Polar` implementiert das Interface `Complex`.
  - Objekte dieser Klasse können einer Variable vom Typ `Complex` zugewiesen werden (Prinzip der Ersetzbarkeit).
  - Die Variable `complexNumber` kann daher auf unterschiedliche Objekte zeigen, der Typ der zugewiesenen Klasse wird zur Laufzeit bestimmt (dynamischer Typ).

Statischer/Dynamischer Typ

Statischer Typ

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

Dynamischer Typ

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

Beispiel dynamischer Typ

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

Codebeispiele

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

Favor Abstract Over Concrete Types

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

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

Implementierung mehrerer Interfaces

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

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

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

Ableiten (bei Interfaces)

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

Konstanten

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

Statische Methoden

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

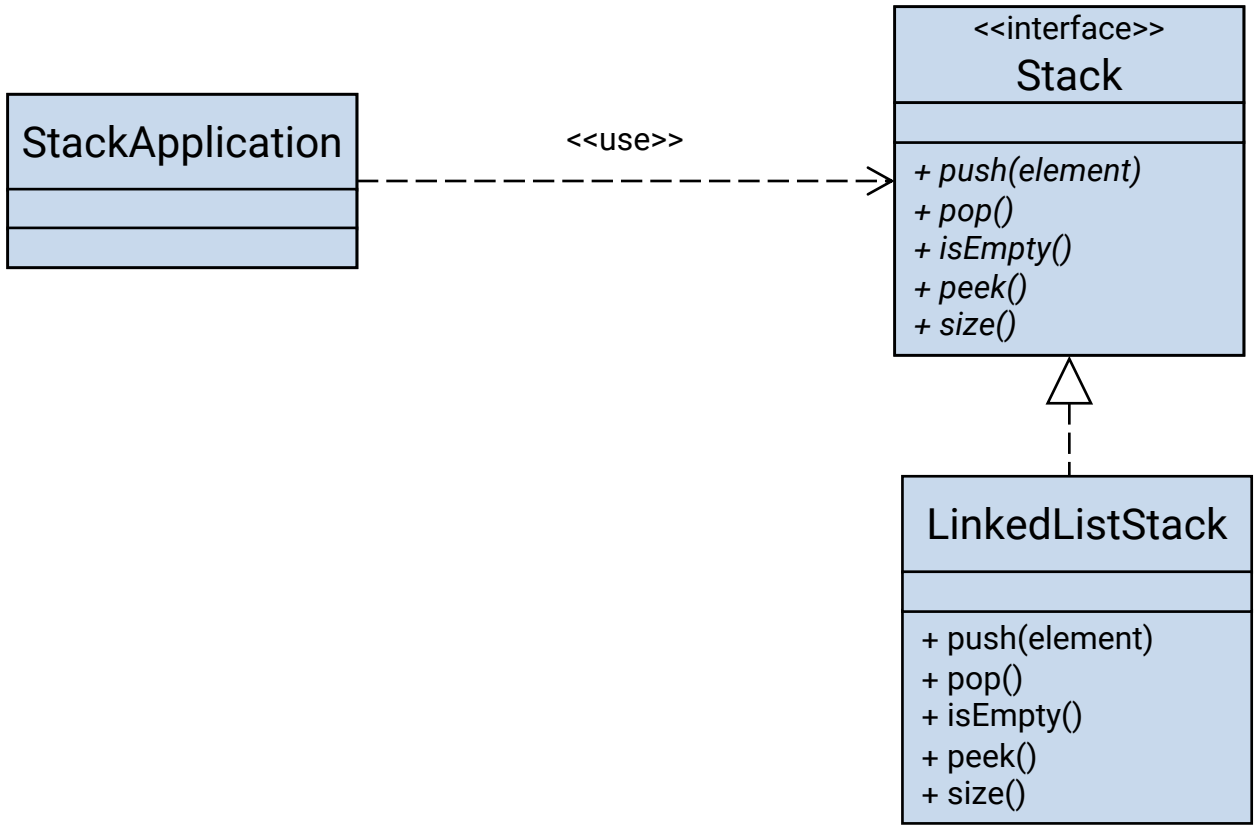
Default Methoden

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

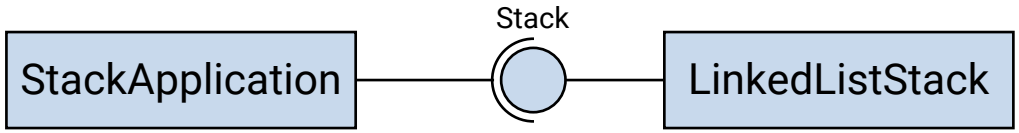
Einsatz von Interfaces

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

Einschub: UML-Notation für Interfaces



- „Lollipop“-Notation



Vererbung der Implementierung in Java

Vererbung und Interfaces

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

Vererbung der Implementierung (allgemein)

- Unterklassen erben die in der Oberklasse bereits implementierte Funktionalität.

- Unterklassen erben
  - Verpflichtungen
  - Alle Methoden
  - Alle Daten

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

- Funktionalität kann komplett übernommen werden **oder** von der Unterklasse verändert/erweitert werden (sogenanntes **Überschreiben**).

Überschreiben

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

Zugriffsschutz

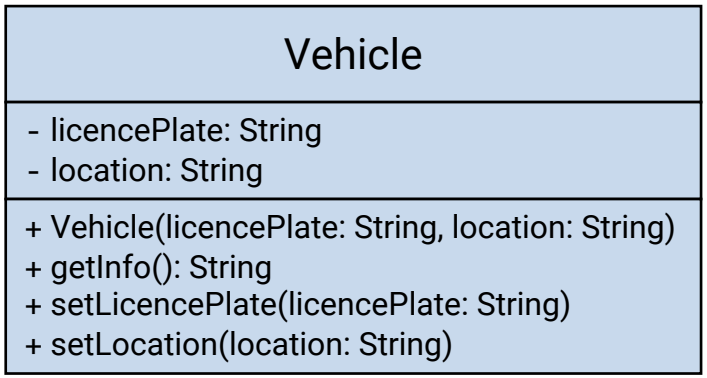
	Klasse	Paket	Subklasse(n)	Alle Klassen
`private`	✓			
default	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

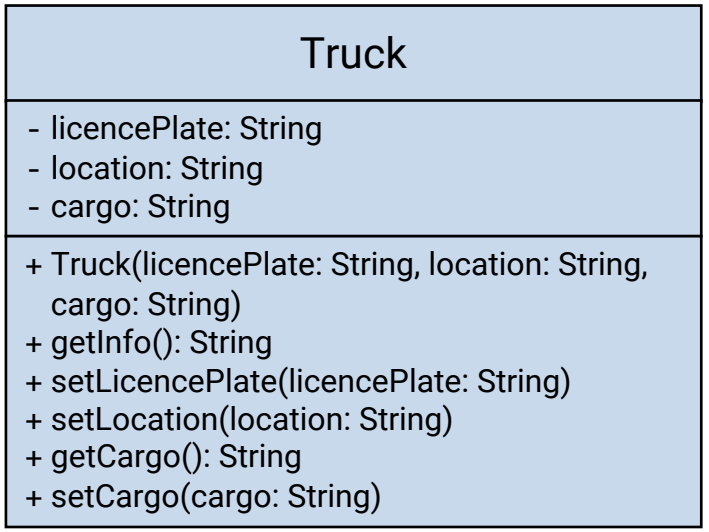
Beispiel Fahrzeugverwaltung

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

Fahrzeugverwaltung: Lösung ohne Vererbung

- Vehicle
  - src/at/ac/uibk/pm/inheritance/vehicles/noinheritance/Vehicle.java
- Truck
  - src/at/ac/uibk/pm/inheritance/vehicles/noinheritance/Truck.java
- VehicleApplication (Main)
  - src/at/ac/uibk/pm/inheritance/vehicles/noinheritance/VehicleApplication.java





Vererbung in Java

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

Konstruktoren bei der Vererbung (1)

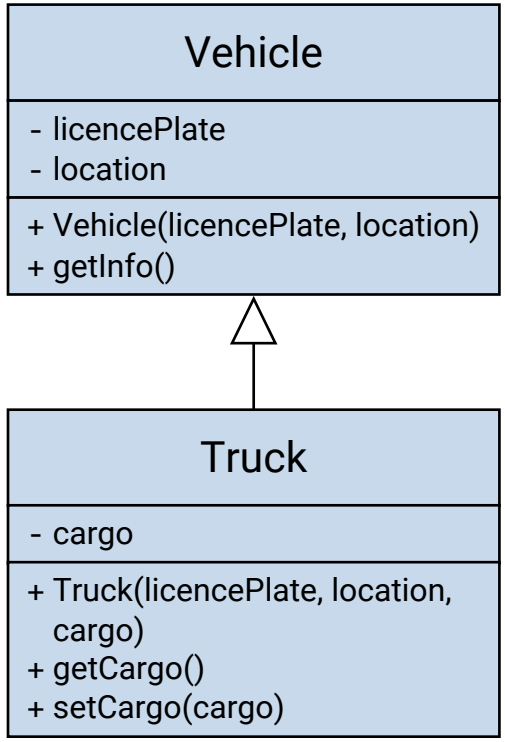
- Konstruktoren werden nicht vererbt.
- In jedem Konstruktor einer Subklasse muss direkt oder indirekt (über Konstruktorenverkettung) ein Konstruktor der Superklasse aufgerufen werden.
- Wird nichts angegeben, dann ruft ein Konstruktor implizit den parameterlosen Konstruktor der Superklasse auf.
- Wenn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

Konstruktoren bei der Vererbung (2)

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

Fahrzeugverwaltung: Lösung mit Vererbung

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



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

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

Zugriffsschutz mit protected

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

Optimierung 1 Fahrzeugverwaltung

- Das Beispiel kann noch weiter optimiert werden.
- Problem: Die `getInfo()`-Methode berücksichtigt noch nicht die Besonderheiten (`cargo`) des Trucks.

  → Überschreiben der `getInfo()`-Methode
- Für Zugriff auf `licensePlate` und `location` der Superklasse müssen diese `protected` sein.

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

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

src/at/ac/uibk/pm/inheritance/vehicles/optimization1 (Package)

Optimierung 2 Fahrzeugverwaltung

- Problem: Das `protected`-Setzen der Felder der Superklasse widerspricht dem Kapselungsprinzip.

  → Verwenden der Getter der Klasse `Vehicle`

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

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

src/at/ac/uibk/pm/inheritance/vehicles/optimization2 (Package)

Optimierung 3 Fahrzeugverwaltung

- Problem: duplizierter Code durch die zwei getrennten `getInfo()`-Implementierungen.

  → Verwenden der getInfo()-Methode der Klasse `Vehicle`

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

src/at/ac/uibk/pm/inheritance/vehicles/optimization3 (Package)

Zugriff auf die Superklasse

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

Typkonvertierung bei Vererbung

- Es ist erlaubt, dass einer Variable o_A vom Typ `A` ein Objekt o_B einer beliebigen Subklasse `B` zugewiesen werden darf.
  - $o_A = o_B$ ist immer erlaubt (Ersetzbarkeitsprinzip).
  - Wird als Up-Cast bezeichnet.

- $o_B = (B)\ o_A$ ist nur zulässig, wenn o_A auf ein `B`-Objekt zeigt (wenn der Typ von o_A also `B` ist).
  - Expliziter Cast notwendig.
  - Wird als Down-Cast bezeichnet.
  - **Down-Cast ist problematisch!**
    - Kann meist mit Hilfe der dynamischen Polymorphie und dem Prinzip der Ersetzbarkeit umgangen werden.

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

instanceof - Operator

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

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

Dynamische Bindung

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

Beispiel 1: Dynamische Bindung

```java
public class A {
  private String name = "A";

  public String getName() {
    return name;
  }
}
```

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

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

Ausgabe:

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

(für bessere Lesbarkeit wurden abstrahierte Klassennamen gewählt)

Beispiel 2: Dynamische Bindung

```java
public class A {
  private String name = "A";

  public String getName() {
    return name;
  }
}
```

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

  @Override
  public String getName() {
    return name;
  }
}
```

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

Ausgabe:

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

(für bessere Lesbarkeit wurden abstrahierte Klassennamen gewählt)

Statische Bindung

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

Beispiel: Statische Bindung Datenelemente

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

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

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

Ausgabe:

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

(für bessere Lesbarkeit wurden abstrahierte Klassennamen gewählt)

final bei Methoden und Klassen

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

Kovarianz, Kontravarianz, Invarianz

- Was darf beim Überschreiben verändert werden?

- Drei Varianten möglich:
  - Kovarianz (entlang der Vererbungsrichtung)
    - Redefinition von Parameter- und Rückgabetypen mit *Subtypen*
  - Kontravarianz (entgegen der Vererbungsrichtung)
    - Redefinition von Parameter- und Rückgabetypen mit *Supertypen*
  - Invarianz
    - Typen bleiben gleich

- Beim Überschreiben in Java
  - Kovarianz beim Rückgabetyp
  - Sonst Invarianz

Kovarianter Rückgabetyp

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

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

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

```
A a = new B();
B b = new B();

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

(für bessere Lesbarkeit wurden abstrahierte Klassennamen gewählt)

Vererbung der Implementierung

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

Probleme

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

Verletzung des Prinzips der Ersetzbarkeit (1)

```
public class Rectangle {
  private int width;
  private int length;

  public void setWidth(int width) { ... }
  public void setLength(int length) { ... }
  …
}
```

```
public class Square extends Rectangle {

}
```

Probleme:

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

Verletzung des Prinzips der Ersetzbarkeit (2)

```
public class Rectangle {
  private int width;
  private int length;

  public void setWidth(int width) { ... }
  public void setLength(int length) { ... }
  …
}
```

```
public class Square extends Rectangle {

  @Override
  public void setWidth(int width) {
    super.setWidth(width);
    super.setLenght(width);
  }

  @Override
  public void setLenght(int length) {
    super.setWidth(length);
    super.setLenght(length);
  }

}
```

Lösungsversuch:

- `setWidth` und `setLength` überschreiben.

Probleme:

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

Problem der instabilen Basisklasse

- Gilt das Prinzip der Ersetzbarkeit? Gilt dies auch in Zukunft?
- Problem der instabilen Basisklasse (Fragile Base Class Problem).
  - Anpassungen an der Basisklasse können zu unerwartetem Verhalten von abgeleiteten Klassen führen.

  →Wartung von Systemen, die nur Vererbung der Implementierung benutzen, ist schwierig.

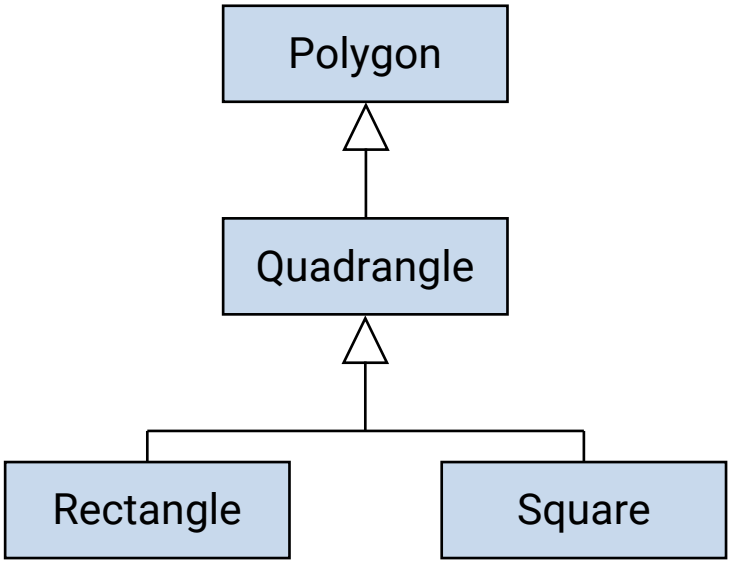
- Sind spätere Änderungen an der Basisklasse sehr wahrscheinlich:
  - Vererbung der *Spezifikation*
  - Vererbung der Implementierung vermeiden
  - Redundanten Code vermeiden durch
    - Aggregation
    - Komposition
    - Delegation

Delegation

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

Abstraktionsebenen

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



„is-a“ vs. „has-a“

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

    → Dreieck wird von Polygon abgeleitet.

  - 2 Klassen Dreieck und Punkt (has-a)
    - Dreieck ist ein Punkt – Falsch
    - Dreieck enthält einen Punkt – Richtig

    → Klasse Dreieck verwendet daher die Klasse Punkt (Objektvariable vom Typ Punkt).

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

Abstrakte Klassen in Java

Abstrakte Klassen

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

Vererbung bei abstrakten Klassen

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

Abstrakte Klasse vs. Interface (1)

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

Abstrakte Klasse vs. Interface (2)

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

Quellen

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