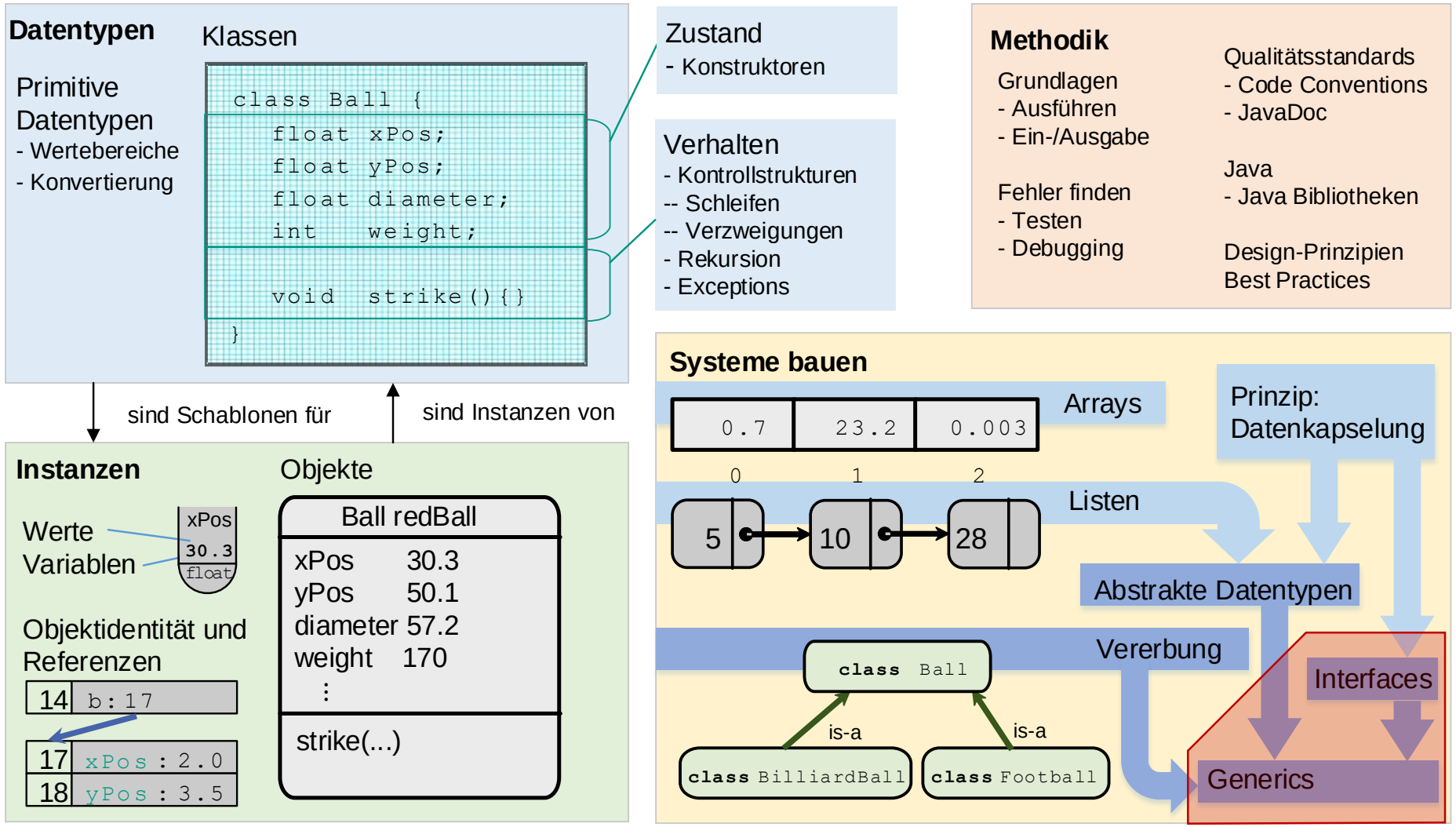
Karlsruher Institut für Technologie

Vorlesung Programmieren

10. Interfaces und Generics

PD Dr. rer. nat. Robert Heinrich

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Vorlesungsüberblick: Vorläufiger Semesterplan

23.10.2023	Erstsemesterbegrüßung: Einführung
25.10.2023	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
08.11.2023	Typen und Variablen
15.11.2023	Kontrollstrukturen (+Scanner)
22.11.2023	Konstruktoren und Methoden
29.11.2023	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
06.12.2023	Listen und Abstrakte Datentypen
13.12.2023	Vererbung
20.12.2023	Exceptions; Interfaces
10.01.2024	Generics; Rekursion
17.01.2024	Java-API; Objektorientierte Design-Prinzipien
24.01.2024	Best Practices; Finden und Beheben von Fehlern
31.01.2024	Testen und Assertions
07.02.2024	Junit; Parsen, Suchen, Sortieren
14.02.2024	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen

Lernziele

Interfaces

- Sie können mithilfe von Interfaces in Java Schnittstellen definieren und damit höhere Datenkapselung erreichen.
- Sie können auswählen, ob ein Interface oder eine abstrakte Klasse für ein gegebenes Problem passender ist.

Generics

- Sie können eine Klasse mit Typ-Parametern generisch machen
- Sie können insbesondere eine generische Listenimplementierung erstellen und verwenden

Quelle: http://phdcomics.com

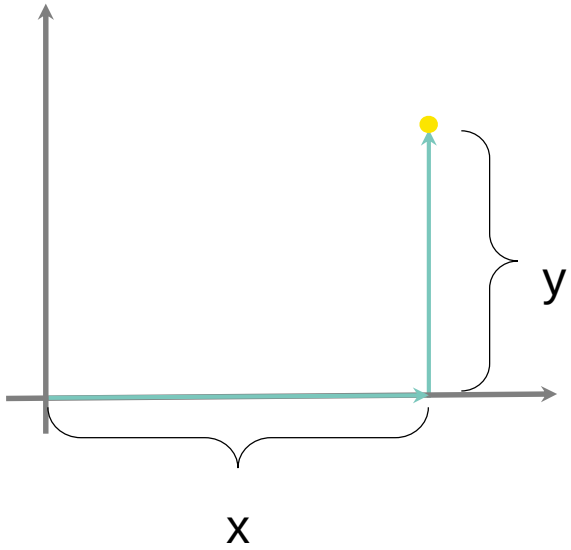
INTERFACES

Motivation – Interfaces

Annahme:

Wir haben eine Klasse `CartesianPoint` zur Beschreibung von Punkten in einem kartesischen Koordinatensystem.

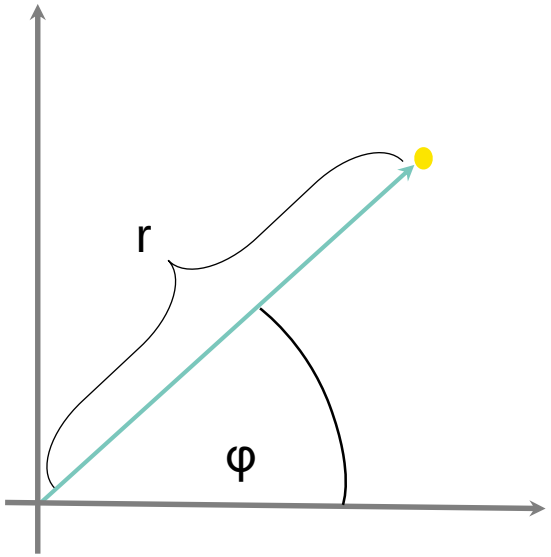
```
class CartesianPoint {
  private double x;
  private double y;
  public CartesianPoint(double x, double y) {…}
  public void shift(double x, double y) {…}
  public double distanceTo(CartesianPoint p) {…}
  public boolean equals(CartesianPoint p) {…}
  ...
}
```



Motivation – Interfaces

Wir möchten Punkte auch mit Polarkoordinaten implementieren.

```
class PolarPoint {
  private double phi;
  private double r;
  public PolarPoint(double phi, double r) {…}
  public void shift(double x, double y) {…}
  public void rotate(double phi) {…}
  public void rotate(PolarPoint center, double phi) {…}
  public double distanceTo(PolarPoint p) {…}
  public boolean equals(PolarPoint p) {…}
  ...
}
```



Interfaces als Schnittstellenbeschreibung (I)

Wir möchten nun mit beiden Punktklassen einheitlich arbeiten können.

➜ Definiere ein **Interface** `Point`. Ein Interface definiert alle nötigen `public` Methoden

```
interface Point {
  void shift(double x, double y);
  void rotate(double phi);  // rotate around origin
  void rotate(Point center, double phi);
  double distanceTo(Point p);
  boolean equals(Point p);
  double getX();
  double getY();
  double distance();
  double angle();
}
```

Dieses Interface kann dann in Variablendeklarationen, Methodensignaturen, Listen usw. anstelle von `CartesianPoint` und `PolarPoint` verwendet werden.

Interfaces als Schnittstellenbeschreibung (II)

Die Klassen `CartesianPoint` und `PolarPoint` müssen nun das Interface `Point` implementieren:

```
class CartesianPoint implements Point {
  private double x;
  private double y;
  ...
}

class PolarPoint implements Point {
  private double phi;
  private double r;
  ...
}
```

- Dabei **müssen alle** im Interface `Point` vorgegebenen Methoden implementiert werden und **`public`** verfügbar sein!
- `CartesianPoint` und `PolarPoint` sind **Subtypen** von `Point`

Interfaces als Schnittstellenbeschreibung (III)

Ein **Interface**

- ist eine Sammlung von Methodensignaturen ohne Implementierung
- legt die Methoden-Namen und Parameter-Typen fest
- macht aber keine Annahme über die Implementierung der Methoden
- kann selbst zur Typisierung verwendet werden:
  ➜ Eine Klasse `C`, die ein Interface `I` implementiert, bildet einen **Subtyp** von `I`.
- wird von einer Klasse implementiert, indem für jede Methode aus dem Interface eine Implementierung angegeben wird
- kann von einem anderen Interface erben:

```
interface I {
  public void foo();
}
```

```
interface J extends I {
  public void bar();
}
```

Interfaces – Syntax

Interface-Deklaration

```
interface InterfaceName {
  Konstanten
  Methodenköpfe
}
```

Implementierung

```
class KlassenName implements InterfaceName {…}
```

- Eine Klasse kann **mehrere Interfaces** gleichzeitig implementieren (Namen werden mit Kommata getrennt)
- Werden mehrere Interfaces implementiert und enthalten diese dieselbe Methodensignatur, so wird die entsprechende Methode in der Klasse **nur einmal implementiert**
- In der implementierenden Klasse muss **jede Methode** implementiert werden (oder die Klasse muss abstract bleiben)

Interfaces – Verwendung

Gegebene Methode (in einer weiteren Klasse):

```
public static void foo(Point p) {
  p.shift(3.0, 5.0);
}
```

Was passiert bei:

```
PolarPoint q = new PolarPoint(90.0, 1.0);
foo(q);
CartesianPoint r = new CartesianPoint(0.0, 1.0);
foo(r);
```

`p` hat den (statischen) Typ `Point`. Beim Aufruf von `p.shift(…)` wird von Java ermittelt, welches Objekt momentan in der Variable steckt und dann die entsprechende Methode dieses Objekts aufgerufen.

➜ Bei `foo(q)` wird `shift()` aus `PolarPoint` aufgerufen

➜ Bei `foo(r)` wird `shift()` aus `CartesianPoint` aufgerufen

Beispiel – Liste

Nächster Abstraktions-Schritt:

- Ein Algorithmus, der auf einer Liste von Punkten arbeitet, sollte unabhängig von der Implementierung der Liste funktionieren

  ➜ Erstelle ein Interface `PointList`, das von der Implementierung abstrahiert

```
interface PointList {
  void addFirst(Point p);
  void addLast(Point p);
  Point getFirst();
  Point getLast();
  ...
}

class DoublyLinkedPointList implements PointList {
  private ListCell first;
  private ListCell last;
  ...
}
```

Interfaces – Übersicht

- Reine Definition eines Typen (Menge von Methoden-Signaturen)
  ➜ Definition einer Schnittstelle
- Methoden-Implementierung erst in den Klassen
- Objekte können mehrere Typen haben
  (Typ einer Klasse und mehrere Interfaces)
- Interfaces können nicht instanziiert werden
  ➜ Keine Erzeugung mit `new`
- Man kann ein Interface von einem oder mehreren anderen mittels `extends` ableiten
- Eine Klasse kann ein oder mehrere Interfaces mittels `implements` implementieren

Abstrakte Klassen vs. Interfaces

Abstrakte Klassen:

- Nicht instanziierbare Oberklassen
- Können Attribute und Methoden-Implementierungen enthalten
- **Verwendung**: partielle Implementierung für gemeinsame Funktionalität zur Verfügung stellen

```
import java.awt.Color;
abstract class ColoredPoint implements Point {
  private Color color;
  public Color getColor() {return color;}
  ...
}
```

Interfaces:

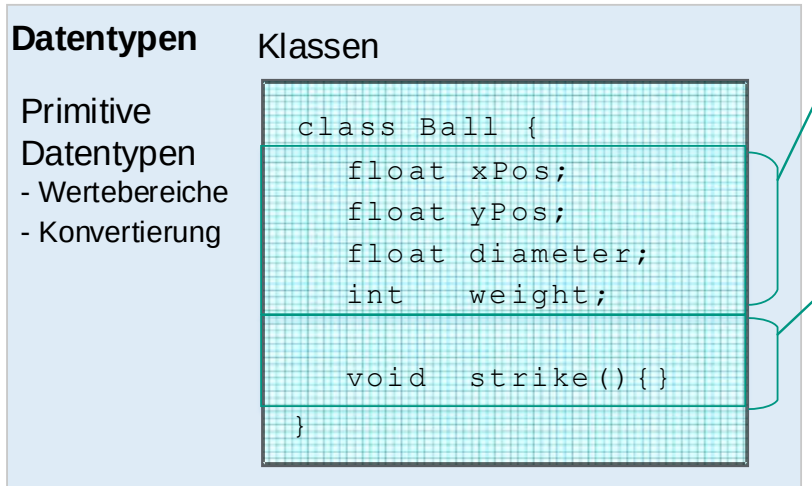
- Definieren Schnittstelle (Menge von Methoden, die eine Klasse zwingend zur Verfügung stellen muss)
- **Verwendung**: Abstraktion von konkreter Implementierung

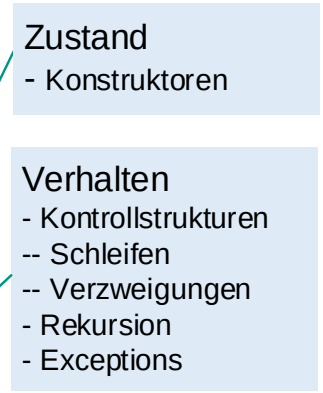
Aufgabe

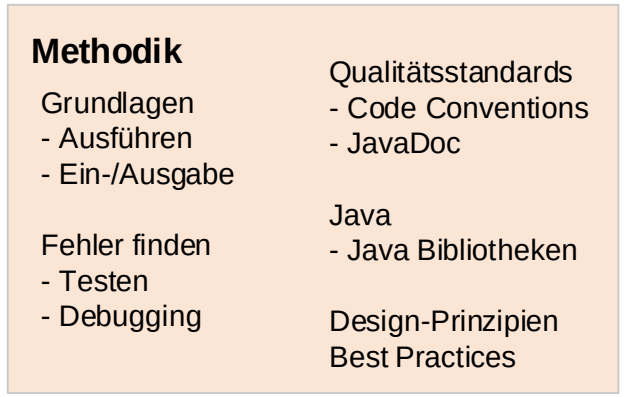
Welche Aussage zu Interfaces und abstrakten Klassen ist *falsch*?

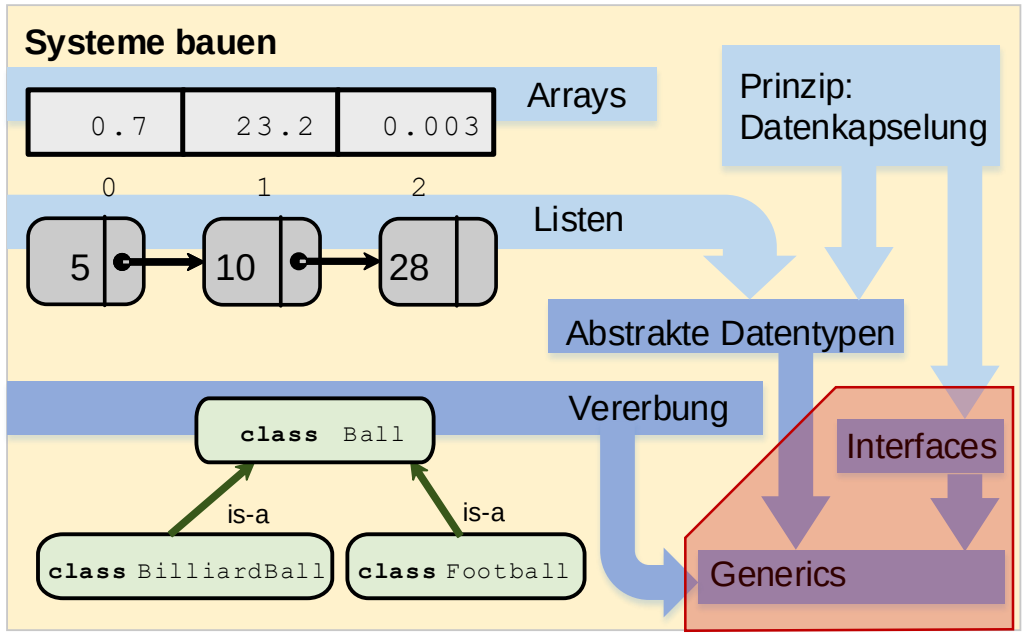
1. Man kann eine abstrakte Klasse von einer anderen abstrakten Klasse ableiten
2. Eine von einer abstrakten Klasse abgeleitete Klasse muss alle abstrakten Methoden implementieren
3. Eine Klasse kann mehrere Interfaces implementieren
4. Man kann ein Interface von einem anderen mittels `extends` ableiten
5. In einer abstrakten Klasse können sämtliche Methoden nicht-abstrakt sein

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java









Hörsaaleinteilung für die Präsenzübung

- Die Präsenzübung findet am 17. Januar 2024 in zwei Sitzungen statt
  - 1. Sitzung von 17:30 bis 17:50 Uhr
  - 2. Sitzung von 18:10 bis 18:30 Uhr

- Informationen zur Hörsaal- und Sitzungseinteilung in der SDQ-NewsList
  - https://news.praktomat.cs.kit.edu

- Nachmeldungen sind nicht mehr möglich!

GENERICS

Motivation – Generische Klassen

Aufgabe 1: Erstelle eine Liste für Point-Objekte

- Haben wir bereits gemacht:
  Interface `PointList`, Implementierung `DoublyLinkedPointList`

Aufgabe 2: Erstelle eine Liste für Line-Objekte

- Entwirf Interface `LineList` und Klasse `DoublyLinkedLineList` analog zu Punkt-Listen

Aufgabe 3: Erstelle eine Liste für beliebige Objekte

???

```
interface LineList {
  void addFirst(Line p);
  void addLast(Line p);
  Line getFirst();
  Line getLast();
  ...
}

class DoublyLinkedLineList implements LineList {
  private ListCell first;
  private ListCell last;
  ...
}
```

- Beim Entwurf der Liste sollte es egal sein, welchen Typ die enthaltenen Objekte haben!

→ Mache den Typ des Listeninhalts **generisch**!

Generics – Konzept

Beobachtung

- Code von Container-Datentypen (und anderen Datenstrukturen) ist unabhängig vom Typ der Basisdaten

Beispiel

- Code zum Einfügen in eine Liste ist unabhängig vom Elementtyp

Idee Typ der Basisdaten ist **Parameter** der Datenstruktur

- Also: Auch Typen können Parameter sein, nicht nur Werte

Vorteile

- Der Compiler kann eine Typprüfung durchführen
  → Programmierfehler werden frühzeitig erkannt
- Keine redundanten Implementierungen

Syntax – Polymorphie in Java

Karlsruher Institut für Technologie

Generische/polymorphe Typen in Java

Syntax

```
class Name<Typ-Parameter> {…}
interface Name<Typ-Parameter> {…}
```

Verwendung

```
TypName<BasisTyp>
```

Beispiel:

- Klassendeklaration:
  ```
  class DoublyLinkedList<Data> {…}
  ```
- Verwendung:
  ```
  DoublyLinkedList<Point> ps = new DoublyLinkedList<Point>();
  ```

Generische Listen (I)

```
class DoublyLinkedPointList {
  private class ListCell {
    Point content;
    ListCell prev;          // Vorgänger
    ListCell next;          // Nachfolger
    ListCell(Point v, ListCell p, ListCell n) {
      this.content = v;
      this.prev = p;
      this.next = n;
    }
  }
  private ListCell first, last;
  public DoublyLinkedPointList() {
    first = last = null;  // leere Liste
  }
  public void addFirst(Point p) {…}
  public void addLast(Point p) {…}
  public void remove(Point p) {…}
  public boolean contains(Point p) {…}
  }
}
```

Generische Listen (II)

```
class DoublyLinkedList<T> {
  private class ListCell {
    T content;
    ListCell prev;       // Vorgänger
    ListCell next;       // Nachfolger
    ListCell(T v, ListCell p, ListCell n) {
      this.content = v;
      this.prev = p;
      this.next = n;
    }
  }
  private ListCell first, last;
  public DoublyLinkedList() {
    first = last = null;  // leere Liste
  }
  public void addFirst(T elem) {…}
  public void addLast(T elem) {…}
  public void remove(T elem) {…}
  public boolean contains(T elem) {…}
}
```

Generische Listen (II)

```
class DoublyLinkedList<T> implements List<T> {
  private class ListCell {
    T content;
    ListCell prev;       // Vorgänger
    ListCell next;       // Nachfolger
    ListCell(T v, ListCell p, ListCell n) {
      this.content = v;
      this.prev = p;
      this.next = n;
    }
  }
  private ListCell first, last;
  public DoublyLinkedList() {
    first = last = null;  // leere Liste
  }
  public void addFirst(T elem) {…}
  public void addLast(T elem) {…}
  public void remove(T elem) {…}
  public boolean contains(T elem) {…}
}
```

Weitere Abstraktion:

```
interface List<T> {
  void addFirst(T elem);
  void addLast(T elem);
  void remove(T elem);
  boolean contains(T elem);
}
```

Verwendung:

```
public void foo() {
  List<Point> l = new DoublyLinkedList<Point>();
  Point p = new CartesianPoint(2.0, 4.0);
  l.addFirst(p);
  …
}
```

Generische Listen (III)

Aufgabe 1: Erstelle eine Liste für Point-Objekte

Lösung: `List<Point> pList = new DoublyLinkedList<Point>();`

Aufgabe 2: Erstelle eine Liste für Line-Objekte

Lösung: `List<Line> lList = new DoublyLinkedList<Line>();`

➜ Man kann jetzt Listen mit beliebigen Typen einfach erstellen!

Mehrere Typ-Parameter

Es dürfen auch mehrere Typ-Parameter verwendet werden:

```
public class Pair<T,S> {

  private T first;
  private S second;

  public Pair(T first, S second) {
    this.first = first;
    this.second = second;
  }

  public T getFirst() {
    return first;
  }

  public S getSecond() {
    return second;
  }
```

Verwendung:

```
Point point = new CartesianPoint(1.4, 5.3);
String pointName = "Punkt p";

Pair<Point, String> pair =
  new Pair<Point, String>(point, pointName);

Point myPoint = pair.getFirst();
```

Einschränkungen

Die Typ-Parameter dürfen nur mit Referenz-Typen instanziiert werden (also keine primitiven Typen).

Man kann auch die zulässigen Typen einschränken:

```
class PointList<T extends Point> {…}
```

`PointList` darf dann nur mit Typen instanziiert werden, die das Interface `Point` implementieren

```
PointList<PolarPoint> pps = new PointList<PolarPoint>(); // ist OK
PointList<String> // erzeugt einen Fehler
```

```
interface I<T extends C & I1 & I2> {…}
```

Problem: Unflexibel – Dazu Exkurs: Kovarianz / Invarianz

Karlsruher Institut für Technologie

Arrays in Java: kovariant

- `PolarPoint` ist Unterklasse von `Point`
  → `PolarPoint[] ist` Unterklasse von `Point[]`
- Beispiel:

```
Point[] a = new PolarPoint[1];
a[0] = new PolarPoint(45.0, 4.0); // ok
a[0] = new String("Hallo");       // Kompilier-Fehler: Type mismatch
```

- Aber: Nicht typsicher:

```
a[0] = new CartesianPoint(5.0, 3.0); // Fehler zur Laufzeit
```

Generics in Java: invariant

- keine Ableitungsbeziehung zwischen z.B. `DoublyLinkedList<Point>` und `DoublyLinkedList<PolarPoint>`
- Beispiel:

```
DoublyLinkedList<Point> a = new DoublyLinkedList<Polarpoint>();
// Kompilier-Fehler: Type mismatch
```

- Zwar typsicher, aber unflexibel
- → Lösung: Sog. Wildcards verwenden, wie im Folgenden erläutert

Wildcards (I)

Wenn eine Methode nur Funktionalität aus `Object` oder generische Listen-Operationen auf den Eingabeparametern anwendet, sollte man statt eines Typ-Parameters T die Wildcard `?` verwenden.

```
class Test {
  private void printList(List<?> l) {
    for (Object o : l) {
      System.out.println(o);
    }
  }

  public void twoPoints() {
    List<Point> ls = new DoublyLinkedList<Point>();
    ls.add(new PolarPoint(45.0, 4.0));
    ls.add(new PolarPoint(90.0, 1.0));
    printList(ls);
  }
}
```

Definition einer Methode, die mit beliebigen Listen arbeiten kann

Verwendung der Methode mit einer Liste von Punkten

Angepasstes Beispiel Example 4.5.1-1. Unbounded Wildcards aus der Java Language Specification (Java SE 8 Edition)
docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.5.1

Wildcards (II)

Warum kann man dann nicht einfach `java.lang.Object` verwenden?

→ Zu unflexibel

Beispiel mit Object: Kompiliert nicht, weil Generics invariant sind.

```
private void printList(List<Object> l) {…}
public void testPrintList() {
    List<Point> lp = new DoublyLinkedList<Point>();
    printList(lp); // Fehler: Method is not applicable
}
```

```
private void printList(List<?> l) {…}
public void testPrintList() {
    List<Point> lp = new DoublyLinkedList<Point>();
    printList(lp); // ok
}
```

Vgl. Java Language Specification (Java SE 8 Edition): docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.5.1

Bounded Wildcards

Über den Unbounded Wildcard `?` dürfen keine Annahmen gemacht werden.

Annahmen möglich bei sog. Bounded Wildcards

- Lower bounded Wildcard `<? super Klassenname>`
- Upper bounded Wildcard `<? extends Klassenname>`

```
private void printPointList(List<? extends Point> l) {
  for (Point p : l) {
    System.out.println(p.distance());
  }
}
public void testPrintPoints() {
  List<CartesianPoint> lp = new DoublyLinkedList<CartesianPoint>();
  printPointList(lp); // OK
}
```

Richtlinien zur Verwendung:
https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html

Wrapper-Objekte

Problem: Typ-Parameter können nicht mit primitiven Typen instanziiert werden

Lösung: Wrapper-Klassen (definiert in `java.lang`)

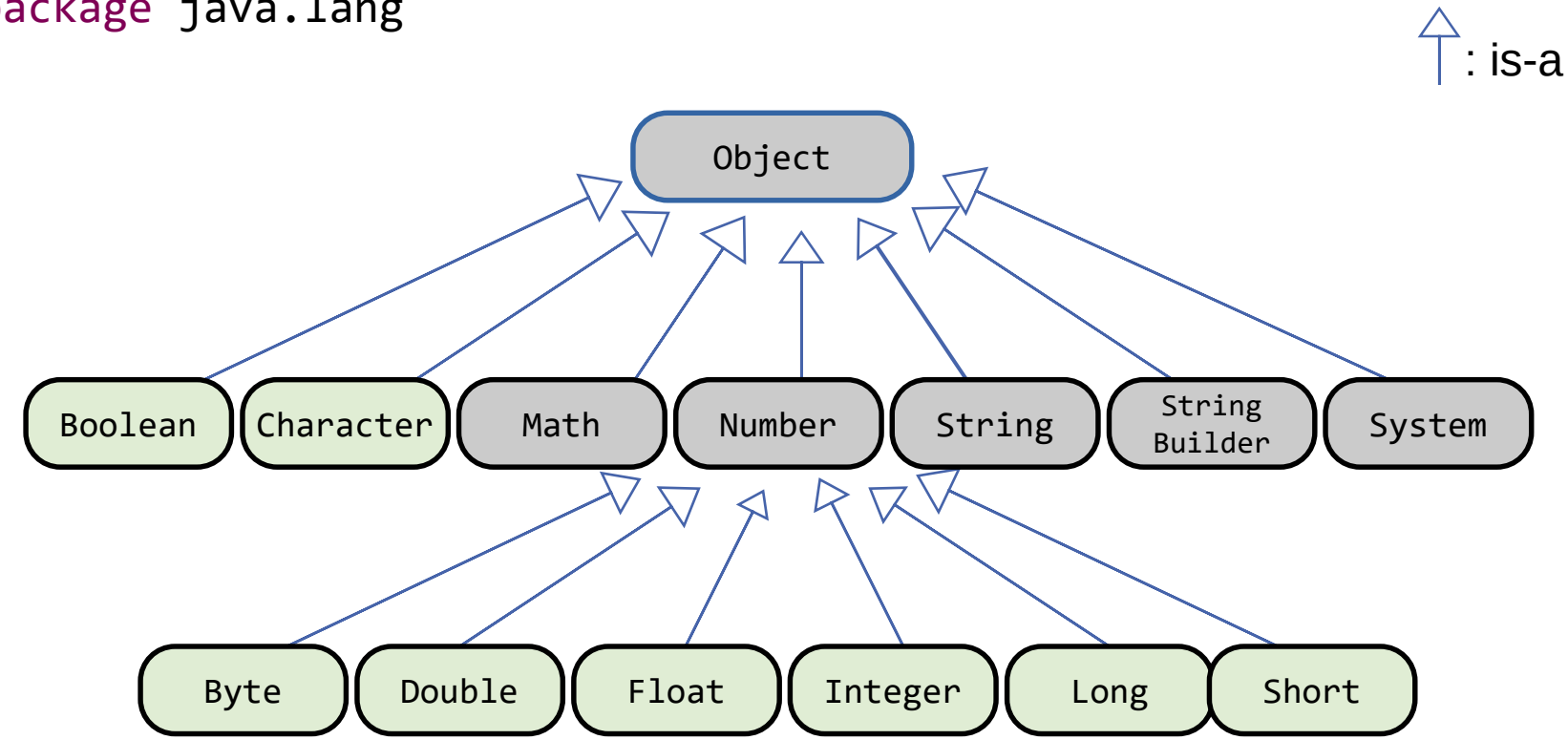
- Wrapper-Klassen nehmen einen primitiven Datentyp in einem Objekt auf
- `Byte`, `Short`, `Integer`, `Long`, `Double`, `Float`, `Boolean`, `Character` (und `Void`)

Anwendungsbeispiel:

```
Integer intObj = new Integer(5);
intObj = Integer.valueOf(17);
Double doubleObj = new Double(4.5);
```

Auszug Java-Klassenhierarchie

```
package java.lang
```



Autoboxing

»Rechnen« mit Wrapper-Objekten vor Java 5 nicht direkt möglich:

```
i++;
intObj = new Integer(intObj.intValue() + 1);
```

Seit Java 5: **Autoboxing**

```
int i = 42;
Integer j = i; // expandiert zu j = Integer.valueOf(i); »Boxing«
int k = j;     // expandiert zu k = j.intValue();       »Unboxing«
```

Generische Methoden

Syntax

```
<Typ-Parameter> Typ Name(Parameter-Liste) {…}
```

Beispiel: Zufällige Auswahl zweier Werte

```
public static <T> T randomSelect(T e1, T e2) {
  return Math.random() > 0.5 ? e1 : e2;
}
```

Beispiel: Minimum zweier vergleichbarer Objekte

```
<T extends Comparable<T>> T min(T e1, T e2) {
  return e1.compareTo(e2) <= 0 ? e1 : e2;
}
```

Hinweis: T ist hier ein neuer Typ-Parameter und nicht derselbe wie der einer möglichweise umgebenden generischen Klasse.

Vergleichen von Objekten

Sollen Objekte z.B. sortiert werden, muss man sie vergleichen können.
Dies wird am besten über eine einheitliche Schnittstelle gemacht:

```
interface Comparable<T> {
  /** Compares this object with the specified object for order.
  */
  int compareTo(T obj);
}
```

Diese Methode vergleicht zwei Objekte `A` und `B` des gleich Typs.

`A.compareTo(B)` liefert:

- einen positiven Wert (z.B. +1), wenn `A` größer als `B` ist
- einen negativen Wert (z.B. −1), wenn `A` kleiner als `B` ist
- die Zahl 0, wenn die beiden Objekte gleich sind

Viele Java-Datenklassen implementieren das Interface `Comparable`, z.B.:

- `java.util.Date`
- `java.lang.String`

Vergleichen von Objekten

- Ein Feld `numbers` von `int`-Werten kann durch den Befehl sortiert werden: `java.util.Arrays.sort(numbers)`
- Andere einfache Datentypen können analog sortiert werden.

```
import java.util.Arrays;

public class SortExample {
  public static void main(String[] args) {
    int[] arr = {42, 8, 4, 23, 16, 15};
    Arrays.sort(arr);
  }
}
```

Ergebnis: `[4, 8, 15, 16, 23, 42]`

Ziel: Beliebige Felder von selbst definierten Objekten mit der `sort()`-Methode sortieren.

Vergleichen von Objekten

Beispiel-Implementierung des Interfaces:

```
class PolarPoint implements Comparable<PolarPoint> {
  private double phi;
  private double r;
  public PolarPoint(double phi, double r) {…}
  public int compareTo(PolarPoint p) {
    return (int)(this.r – p.r);
  }
  …
}
```

Nach dem Implementieren des Interfaces `Comparable` können `PolarPoints` mit der Methode `java.util.Arrays.sort(…)` sortiert werden. Kollektionen können auch mit `java.util.Collections.sort(…)` sortiert werden.

Nach welchem Kriterium vergleicht die `sort()`-Methode die Punkte?

Radius des Punktes

Zusammenfassung

Abstrakte Klassen:

- Nicht instanziierbare Oberklassen
- Können Attribute und Methoden-Implementierungen enthalten
- **Verwendung:** partielle Implementierung für gemeinsame Funktionalität zur Verfügung stellen

Interfaces:

- Definieren Schnittstelle (Menge von Methoden, die eine Klasse zwingend zur Verfügung stellen muss)
- **Verwendung:** Abstraktion von konkreter Implementierung

Generics:

- Ersetzen eines konkreten Typs durch eine "Typ-Variable" (Parameter)
- Typ-Parameter erlaubt Wiederverwendung bzw. Generalisierung von Programmcode
- **Verwendung:** Container-Klassen, Abstraktion von Typen