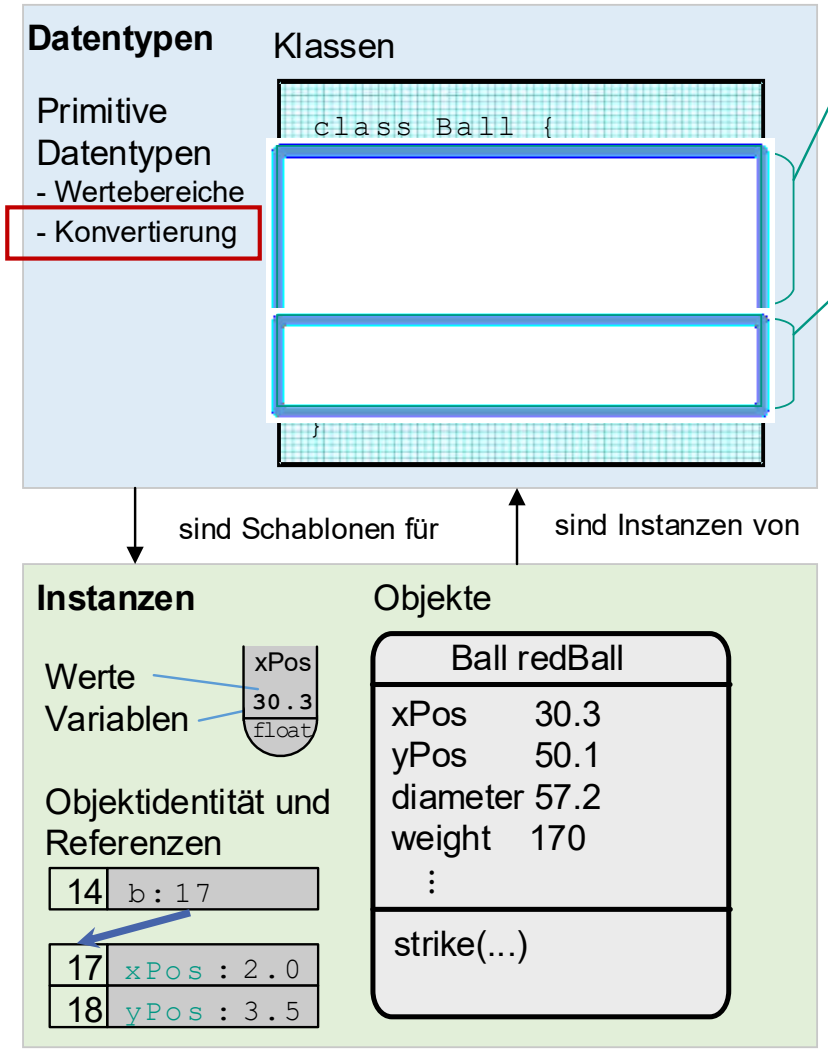
Vorlesung Programmieren

6. Konvertierung, Datenkapselung, Sichtbarkeit

PD Dr. rer. nat. Robert Heinrich

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Zustand
- Konstruktoren

Verhalten
- Kontrollstrukturen
-- Schleifen
-- Verzweigungen
- Rekursion
- Exceptions

Methodik

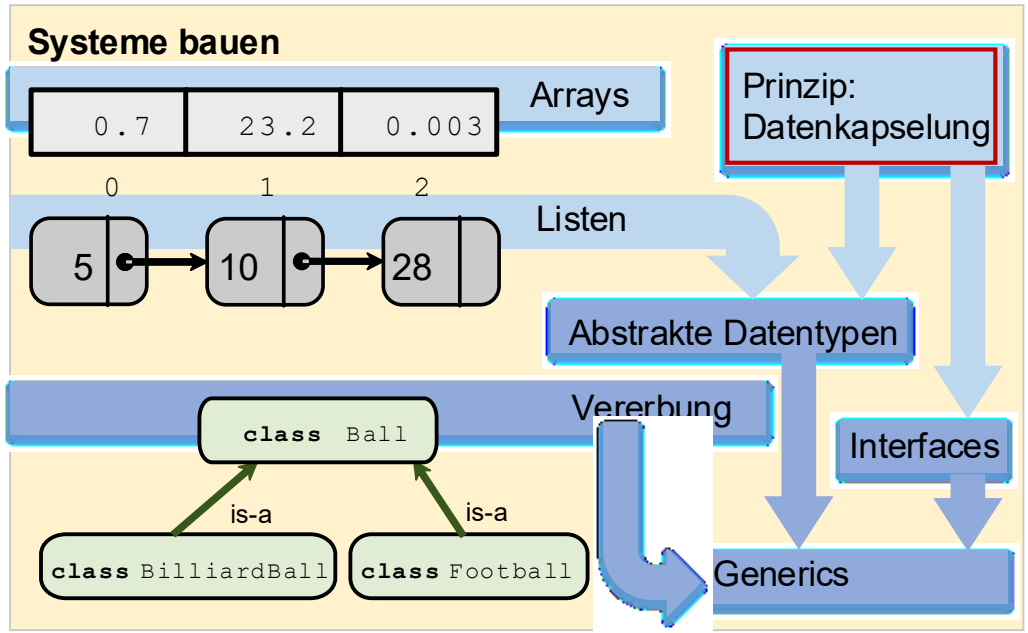
Grundlagen
- Ausführen
- Ein-/Ausgabe

Fehler finden
- Testen
- Debugging

Qualitätsstandards
- Code Conventions
- JavaDoc

Java
- Java Bibliotheken

Design-Prinzipien
Best Practices



Lernziele

Quelle: http://phdcomics.com

Konvertierung

- Sie wissen, welche Konvertierungen Java automatisch vornimmt
- Sie können Konvertierungen explizit über den Cast-Operator vornehmen
- Sie können eine Methode zur Konvertierung von Objekten Ihrer eigenen Klassen in Strings angeben

Datenkapselung

- Sie können Attribute und Methoden Ihrer Klassen nach außen verbergen
- Sie kennen die Bedeutung der einzelnen Zugriffsrechte in Java

Sichtbarkeit

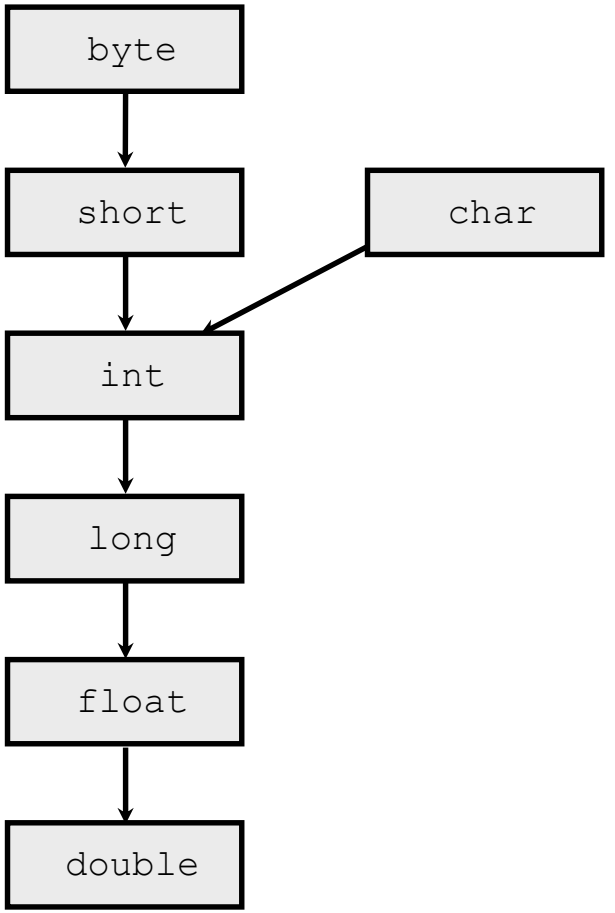
- Sie kennen den Gültigkeitsbereich und den Sichtbarkeitsbereich von Variablen sowie ihre Lebensdauer

Typ-Konvertierung

- Typen von Variablen, Konstanten und Methoden können von Java automatisch angepasst werden
  - z.B. innerhalb des Ausdrucks `a+b`, wobei `int a;` und `byte b;`
- **Die von Java vorgenommenen Anpassungen fallen in 11 Gruppen:**
  - Erweiternde Primitivkonvertierung (*widening primitive conversion*)
  - Einschränkende Primitivkonvertierung (*narrowing primitive conversion*)
  - String-Konvertierung (*string conversion*)
  - ...und weitere, die wir hier nicht betrachten
- **Java kann Anpassungen in folgenden Fällen vornehmen:**
  - bei Zuweisungen (*assignment conversion*)
  - bei Methodenaufrufen (*method invocation conversion*)
  - in Ausdrücken bei numerischen Typen (*numeric promotion*)
  - erzwungene Konvertierung (*casting conversion*)

Widening Primitive Conversions

- **Widening Primitive Conversions**
  - **kein Informationsverlust** bzgl. der **Größe** eines numerischen Werts
  - **Informationsverlust** hinsichtlich **Präzision** möglich
  - *Widening primitive conversions* erfolgen automatisch (kein expliziter Cast erforderlich)




```
class Test {
  public static void main(String[] args) {
    int big = 1234567890;
    float approx = big;
    System.out.println(big - (int)approx);
  }
}
```

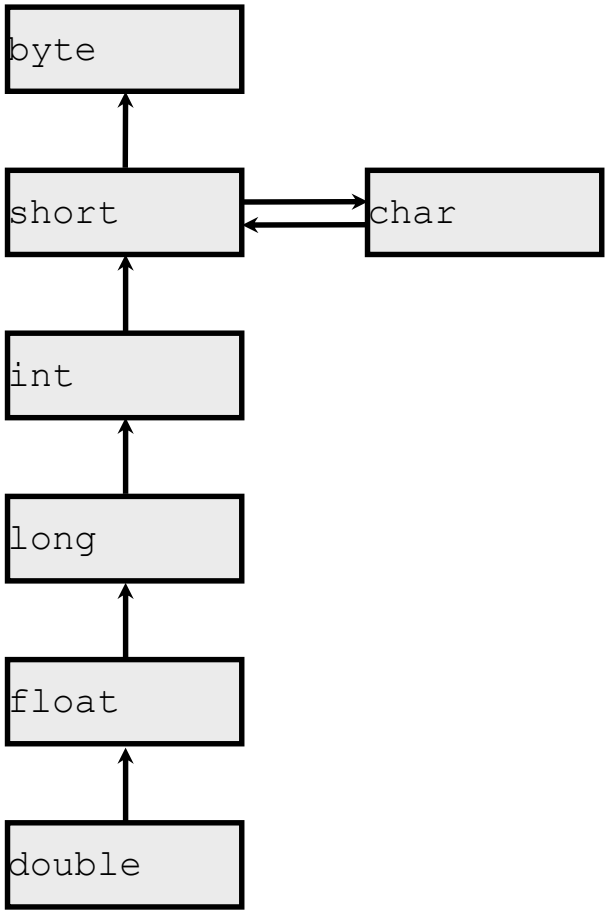
Ausgabe: -46

Beispiel von https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.2

Narrowing Primitive Conversions

- **Narrowing Primitive Conversions**
  - **Informationsverlust** bzgl. der **Größe** eines numerischen Werts möglich
  - **Informationsverlust** hinsichtlich **Präzision** möglich
- Bei ganzzahligen Typen:
  - höherwertige Bits werden abgeschnitten
- *Narrowing primitive conversions* müssen über den Cast-Operator erzwungen werden
- Bei *narrowing primitive conversions* sollte immer sichergestellt sein, dass kein Informationsverlust bzgl. der Größe auftritt!
  - Bsp.:

```
int x = 2000;
byte b = (byte)x; // b hat nun den Wert -48
```



String Conversions

- Jeder Datentyp kann nach String konvertiert werden
- Für primitive Typen geschieht dies automatisch
  - Bsp.: `int x = 10; System.out.println("x = " + x);`
- Für Referenztypen (Klassen) kann eine eigene Methode (`toString`) zur Konvertierung angegeben werden:

Beispiel:

```
class Vector2D {
  double x, y;

  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

- Ist keine `toString`-Methode vorhanden, so wird eine Default-Methode verwendet, die den *Klassennamen* und den *Hash-Wert* des Objekts ausgibt

Casting Conversions

- In Java können Typkonvertierungen mittels des Cast-Operators erzwungen werden
  - Bsp.: `double d = 2.5; int x = (int) d;`

- Falls eine Konvertierung nicht möglich ist, kann ein Fehler zur Compilezeit oder zur Laufzeit resultieren

Datenkapselung

Datenkapselung:

- Verbergen von Daten/Informationen vor dem Zugriff von außen
- Zugriff auf Daten nur über genau festgelegte Schnittstellen möglich

Vorteile:

- Ermöglicht "lose gekoppelte" Programmkomponenten
- Weniger Abhängigkeiten: keine gemeinsam genutzten "globalen" Variablen
- Komponenten müssen nichts über die interne Implementierung anderer Klassen wissen

Beispiel ohne gute Datenkapselung

```
class Patient {
  String name;
  int age;
  ...
}
```

```
class PatientFile {
  Patient patient;
  boolean isChild() {
    return (patient.age < 15);
  }
  ...
}
```

Datenkapselung

Datenkapselung:

- Verbergen von Daten/Informationen vor dem Zugriff von außen
- Zugriff auf Daten nur über genau festgelegte Schnittstellen möglich

Vorteile:

- Ermöglicht "lose gekoppelte" Programmkomponenten
- Weniger Abhängigkeiten: keine gemeinsam genutzten "globalen" Variablen
- Komponenten müssen nichts über die interne Implementierung anderer Klassen wissen

Beispiel ohne gute Datenkapselung nach Änderung:

Änderung

```
class Patient {
  String name;
  int birthYear;
  ...
}
```

```
class PatientFile {
  Patient patient;
  boolean isChild() {
    return (patient.age < 15);
  }
  ...
}
```

Datenkapselung: Zugriffsrechte

- Java erlaubt eine Steuerung der Zugriffsrechte über die Attribute **`private`** und **`public`**.
  - Diese Attribute können bei der Deklaration von Methoden, Attributen (und auch Klassen) verwendet werden.
  - **`private`**: Zugriff nur innerhalb der eigenen Klasse (d.h. in Methoden der eigenen Klasse) möglich
  - **`public`**: Zugriff vom gesamten Programmtext (d.h. aus allen Klassen) möglich.

- **Richtlinie:**
  - Sämtliche Attribute einer Klasse sollten **`private`** sein!
    - Bei Konstanten (**`static`**) kann evtl. die Verwendung von **`public`** sinnvoll sein.
  - Zugriff auf Attribute dann über **`set-`** und **`get-`**Methoden.
  - Methoden sind meist **`public`**. Wenn sie nur als lokale Hilfsmethoden in der Klasse gedacht sind, sollten sie **`private`** sein.

Zugriffsrechte: Beispiel

```java
class Patient {
  private String name;
  private int age;

  public String getName() {
    return name;
  }
  public void setName(String newName)  {
    name = newName;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int newAge) {
    age = newAge;
  }
}
```

```java
class PatientFile {
  private Patient patient;

  public boolean isChild() {
    return (patient.getAge() < 15);
  }
  ...
}
```

Zugriffsrechte: Beispiel

```java
class Patient {
  private String name;
  private int birthYear;

  public String getName() {
    return name;
  }
  public void setName(String newName) {
    name = newName;
  }

  public int getAge() {
    return (currentYear() - birthYear);
  }

  public void setAge(int age) {
    birthYear = currentYear() - age;
  }

  ...
  private int currentYear() { ... }
}
```

```java
class PatientFile {

  private Patient patient;

  public boolean isChild() {
    return (patient.getAge() < 15);
  }
  ...
}
```

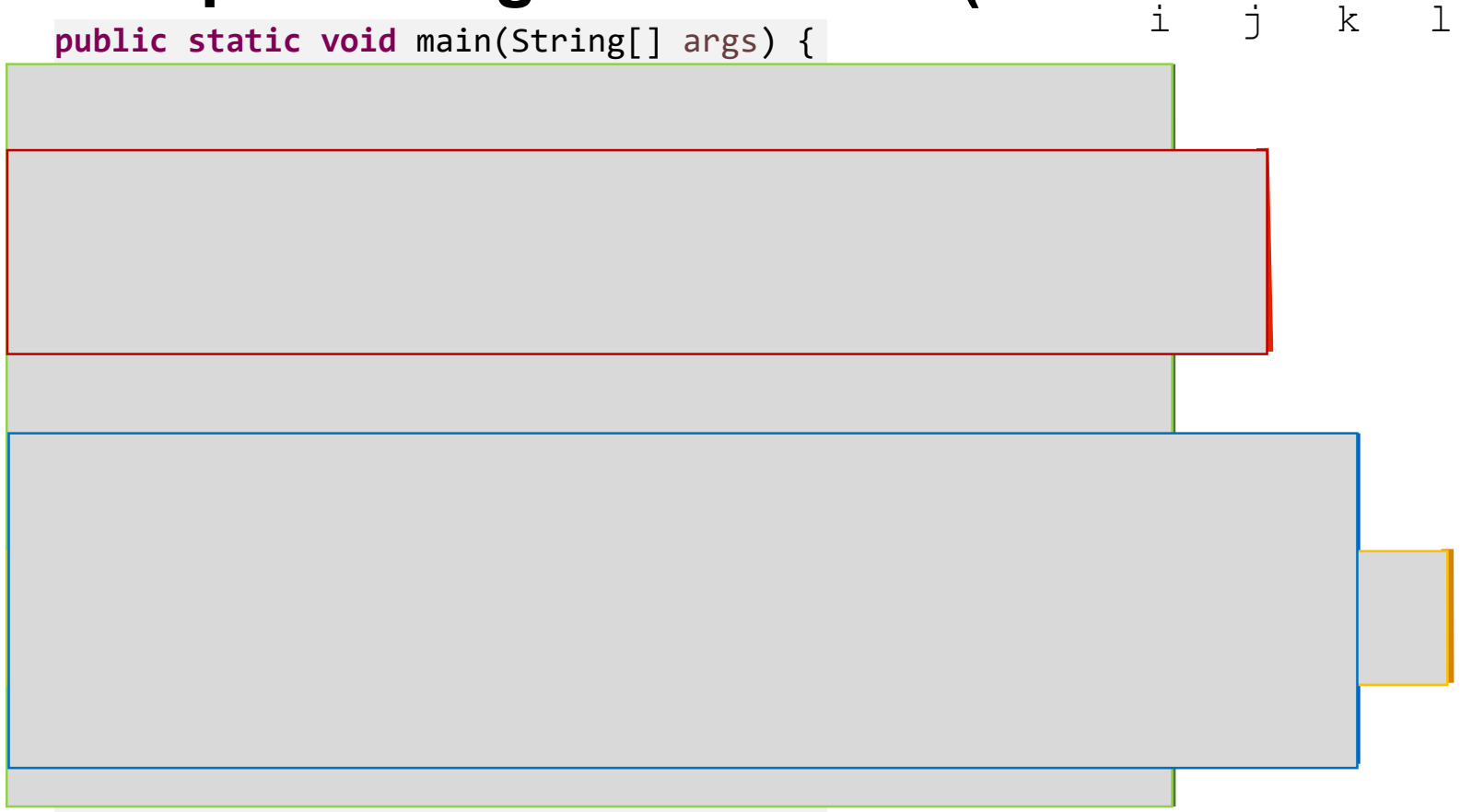
Änderung des Attributes `age` in `birthYear` erfordert nur ***lokale*** Änderungen in der Klasse `Patient`, aber keine Änderungen in anderen Klassen (z.B. `PatientFile`)!

➜ lose Kopplung, bessere Wartbarkeit

Gültigkeit, Sichtbarkeit, Lebensdauer von Variablen

- Der **Gültigkeitsbereich (*scope*)** einer Variablen ist derjenige Teil des Programmtextes, auf den sich die Deklaration bezieht.
- Der **Sichtbarkeitsbereich** einer Variablen ist derjenige Teil des Programmtextes, in dem auf die Variable über ihren Namen zugegriffen werden kann.
- Die **Lebensdauer** einer Variablen umfasst diejenigen Zeitabschnitte der Programmausführung, in denen für die Variable Speicherplatz angelegt ist.

Beispiel Gültigkeitsbereich (lokale Variablen)



In Java ist eine lokale Variable ab deren Deklaration bis zum Ende des Blocks, in dem sie deklariert wurde, gültig.

Gültigkeit

- **Lokale Variablen** sind ab deren Deklaration bis zum Ende des Blocks, in dem sie deklariert wurden, gültig.
- **Parameter** sind innerhalb der Methode gültig.
- **Attribute** sind innerhalb der Klasse (also den Methoden der Klasse), in der sie deklariert wurden, gültig.

Überschattung

- **Überschattung:**
  - Falls Variablen mit gleichem Namen in einer Klasse verwendet werden (z.B. einmal als Attribut, einmal als Parameter), kann es Programmbereiche geben, in denen zwei Variablen des gleichen Namens gültig sind. Eine Variable überschattet dann die andere.

- **Lokale Variablen / Parameter:**
  - In Java ist es verboten, mehrere lokale Variablen oder Parameter desselben Namens innerhalb einer Methode zu deklarieren
  - **Konsequenz** für lokale Variablen / Parameter: **keine Überschattung möglich**

- **Attribute:**
  - Attribute dürfen gleich heißen wie lokale Variablen / Parameter!
  - Bei Namenskonflikten haben lokale Variablen / Parameter Vorrang
    - Lokale Variablen können daher Attribute überschatten
    - Zum Auflösen von solchen Namenskonflikten **`this`**-Referenz verwenden

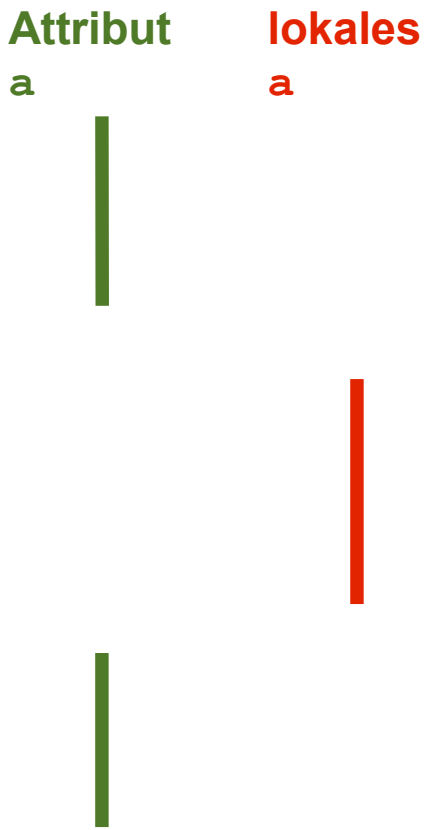
Überschattung: Beispiel

```
class HoleInScope {
  private int a;

  void foo() {
    ...
    a = 5;
    ...
    for (int a = 1; a < 10; a++) {
      ...
      s += a;
      ...
    }
    ...
    a++;
    ...
  }
}
```

Schlechter Programmierstil!
Lokale Variablen sollten nicht so heißen wie Attribute!

Sichtbarkeit



Sichtbarkeit (*Visibility*)

- Wie wir bereits gesehen hatten, kann über **`private`**/**`public`**-Attribute der Zugriff auf Methoden und Attribute gesteuert werden.

- **`private`** und **`public`** beeinflussen die Sichtbarkeit:
  - **`private`-Attribute** sind nur in der eigenen (deklarierenden) Klasse sichtbar
  - **`public`-Attribute** sind überall sichtbar (Zugriff über *`Objektname.Attribut`*)

Pakete (*packages*)

- **Definition:**
  Ein **Paket** (*package*) ist eine Gruppierung von zusammengehörigen Klassen.
- Die Zugehörigkeit von Klassen kann über das Schlüsselwort **`package`** gesteuert werden
  - **Syntax:** `package` *`Name`*`;`
  - Eine **`package`**-Deklaration muss immer am Anfang der Datei stehen und zeigt an, dass die nachfolgende Klasse zum Paket *`Name`* gehört.
- Beispiel:

`Vector2D.java`

```
package geometry;

class Vector2D {
  ...
}
```

`Line2D.java`

```
package geometry;

class Line2D {
  ...
}
```

Mehr zu Paketen

- **Namenskonvention:**
  - Paketnamen werden klein geschrieben
  - Um Namenskonflikte zu vermeiden, wird häufig der umgekehrte Domainnamen dem Paketnamen vorangestellt
    - **Beispiel:** Domain: `kit.edu`; Paketname: `edu.kit.geometry`
- Um auf Klassen anderer Pakete zuzugreifen wird deren **qualifizierter Name** verwendet:
  - **Beispiel**: `edu.kit.geometry.Vector2D v = new edu.kit.geometry.Vector2D();`
- Um Schreibarbeit zu sparen, können Pakete / Klassen **importiert** werden:

Beispiele:

```
import edu.kit.geometry.Vector2D;

class Vector2DTest {
  Vector2D p;
  ...
}
```

```
import edu.kit.geometry.*;

class Line2DTest {
  Vector2D p;
  Line2D l;
  ...
}
```

Pakete und Sichtbarkeit

- Pakete dienen auch einer feineren Abstufung der Sichtbarkeit:

Modifikator	Sichtbarkeit		
	in Klasse	in Paket	„Welt"
`public`	✓	✓	✓
–	✓	✓	–
`private`	✓	–	–

Gleichheit von Objekten

In Java gibt es zwei Möglichkeiten, auf Gleichheit zu prüfen:

- **"=="-Operator**: prüft auf Wert-Gleichheit
  - Bei Objekten bedeutet dies **Gleichheit der Referenz** (also ob auf dasselbe Objekt verwiesen wird)
- **`equals`-Methode**: prüft auf inhaltliche Objekt-Gleichheit
  - `equals` sollte für jede eigene Klasse überschrieben werden

Beispiel:

```
public class Vector2D {
  double x, y;
  ...
  public boolean equals(Vector2D other) {
    if (other == null)
      return false;
    return (this.x == other.x) && (this.y == other.y);
  }
}
```

Verwendungsbeispiel: Gleichheit von Objekten

- z.B. Ball soll aufblinken wenn sich Geschwindigkeit ändert

```
class Ball {
  // Attribute
  Vector2D position; // in cm
  Vector2D velocity; // in m/s
  ...
  public void setVelocity(Vector2D newVelocity) {
    if (!this.velocity.equals(newVelocity)) {
      // velocity changed: Inform listeners
    }
    this.velocity = newVelocity;
  }
}
```

Randbemerkung:

- Schlechter Stil, die Details zur Benutzeroberfläche direkt in fachlicher Klasse zu implementieren
- Deswegen nur: Informiere die interessierten Objekte.
- Hier: Objekte die Benutzeroberfläche steuern.

Zusammenfassung

Typ-Konvertierungen:

- Erweiternde und einschränkende Konvertierungen (und einige weitere)
- Mögliche Informationsverluste müssen stets bedacht werden

Datenkapselung:

- Erlaubt Daten / Methoden vor dem Zugriff von außen zu verbergen
- Zugriff nur über definierte Schnittstelle

Sichtbarkeit:

- Sichtbarkeits-Modifikatoren (**`public / private`**) und Pakete unterstützen die Datenkapselung

Literaturhinweis - Weiterlesen

- Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger "Grundkurs Programmieren in Java", 7. Auflage, 2014 (mit Java 8), Hansa-Verlag
  - "Sichtbarkeit und Verdecken von Variablen"
  - "Kapselung"