Department Mathematik/Informatik, Abteilung Informatik
Software & System Engineering
Weiterführende Konzepte der Programmierung, SS 2025

Niklas Bockholt, Chiara Hafner, Alicja Bezpalko,
Jiyan Alis, Dilara Günay,
Simon Wolf, Adrian Bajraktari, Dr. Mersedeh Sadeghi

Homework 3. Object-Oriented Programming

Registration Deadline: 27.04.2025, 23:59
Hand-in Deadline: 30.04.2025, 23:59

Exercise 1. Who Can See What?

Create a class `Account` with the following attributes:

- `public String owner`.
- `private double balance`.
- `protected int pin`.
- `String internalNote` (default / package-private access).

Write methods to:

- return the balance (`getBalance()`)
- change the pin (only with correct current pin)

Then create:

- class `AccountManager` in the **same** package.
- class `ExternalAudit` in a **different** package.
- class `CompanyAccount` in a different package that **inherits from** `Account`.

Try accessing all four attributes from all three classes.

- Explain which accesses fail and why.
- Suggest alternative designs (e.g., getters).
- What is the difference between using `protected` vs. using `private` and providing a getter for subclasses to access an instance variable?

Exercise 2. Who Are You Really?

Create:

- class `Message` with method `getType()` which returns `"Generic"`.
- class `Email extends Message` where `getType()` returns `"Email"`, and a method `send()` that prints `"E-Mail sent"`.

- class `SMS extends Message` where `getType()` returns `"SMS"`.

Write:

```
Message m1 = new Email();
Message m2 = new SMS();

System.out.println(m1.getType());
System.out.println(m2.getType());
// m1.send(); // Uncomment this line and explain what happens
```

Answer the following questions.

- What is the static type of `m1`? What is its dynamic type?
- Which method is called? Why?
- Why can't `send()` be called directly?
- How could you safely call `send()` anyway (e.g. `instanceof`)?

Antworten auf die Fragen:

1. **Typen von m1:**
   - **Statischer Typ:** `Message` (der deklarierte Typ)
   - **Dynamischer Typ:** `Email` (der tatsächliche Objekttyp zur Laufzeit)
2. **Methodenaufruf:**
   - Es wird die überschriebene Methode `getType()` aus der dynamischen Klasse (`Email` bzw. `SMS`) aufgerufen
   - Grund: Polymorphie - Java verwendet späte Bindung (late binding) für überschriebene Methoden
3. **Warum send() nicht direkt aufrufbar ist:**
   - Der statische Typ von m1 ist `Message`, und die `Message`-Klasse hat keine `send()`-Methode
   - Der Compiler kennt nur den deklarierten Typ, nicht den dynamischen Typ
4. **Sicherer Aufruf von send():**

java Copy Download

```java
if (m1 instanceof Email) {
    ((Email) m1).send(); // Typecast nach Überprüfung
}
```

Erweiterte Erklärung:

Polymorphie-Konzept:

- Die Ausgabe zeigt "Email" und "SMS", obwohl die Variablen als `Message` deklariert sind
- Dies ist Polymorphie in Aktion - die Laufzeitumgebung ruft die spezifischste Implementierung auf

Kompilierfehler:

- Wenn man `m1.send()` direkt aufruft, erhält man:

```
error: cannot find symbol
  m1.send();
    ^
symbol: method send()
```

- Der Compiler sucht die Methode im statischen Typ (`Message`)

Diese Aufgabe demonstriert wichtige OOP-Konzepte:

1. Vererbung und Methodenüberschreibung
2. Polymorphie und dynamische Methodenbindung
3. Unterschied zwischen statischem und dynamischem Typ
4. Sichere Verwendung von Typecasts mit `instanceof`

Exercise 1:

Attribut	AccountManager (selbes Paket)	ExternalAudit (anderes Paket)	CompanyAccount (Vererbung, anderes Paket)
public owner	✅ Zugriff	✅ Zugriff	✅ Zugriff
private balance	❌ (nur via getBalance())	❌ (nur via getBalance())	❌ (nur via getBalance())
protected pin	✅ Zugriff	❌ Kein Zugriff	✅ Zugriff
package internalNote	✅ Zugriff	❌ Kein Zugriff	❌ Kein Zugriff

- **Fehlgeschlagene Zugriffe:**
  - private balance: Nur innerhalb der Account-Klasse sichtbar
  - protected pin: Sichtbar für Unterklassen (auch in anderen Paketen) und Klassen im selben Paket
  - internalNote: Nur für Klassen im selben Paket sichtbar

2. **Alternative Designs:**
   - Für `balance`: Private halten + Getter/Setter (wie implementiert)
   - Für `pin`: Protected ist okay für Erweiterungen, aber besser wäre:

```java
private int pin;
protected int getPin() { return this.pin; }
protected void setPin(int pin) { this.pin = pin; }
```

   - Für `internalNote`: Entweder public/getter oder bei Paketsicht belassen
3. **protected vs. private mit Getter:**
   - **protected Variable:** Direkter Zugriff für Unterklassen, aber gefährlich (kann unkontrolliert verändert werden)
   - **private mit protected Getter:** Mehr Kontrolle (Validierung möglich), aber mehr Code
   - **Empfehlung:** Wichtige Variablen immer private + Getter/Setter, protected nur wenn wirklich nötig

Wichtige Java-Sichtbarkeitsregeln:

1. `public` - Überall sichtbar
2. `protected` - Paket + Unterklassen (auch andere Pakete)
3. *default (package-private)* - Nur im selben Paket
4. `private` - Nur in derselben Klasse