Grundlegende Modularisierung von einfachen Java Programmen

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de , Raum 149B

Version: 1.0.2.1

Kontrollfragen: kontrollfragen.de.md.html
Folien: https://delors.github.io/prog-java-modularisierung-101/folien.de.md.html
https://delors.github.io/prog-java-modularisierung-101/folien.de.md.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues
KI Verwendung: Bei der Erstellung der Unterlagen wurden KI Assistenten (insbesondere Claude aber ggf. auch ChatGPT, Ollama mit Gemma/LLama/Qwen oder OpenCode mit Kimi/Qwen/Deepseek...) unterstützend eingesetzt. Dies erfolgte insbesondere zur Unterstützung bei der Generierung von Grafiken (d. h. SVG Dateien), oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert und angepasst.

1. Grundlagen der Modularisierung

Modularisierung[1]

Definition

Modul

> Ein Modul ist im Software Engineering ein Baustein eines Softwaresystems, der bei der Modularisierung entsteht, eine funktional geschlossene Einheit darstellt und einen bestimmten Dienst bereitstellt.
>
> —wikipedia.org Modul (Software)

[1] Java unterstützt neben Klassen und Packages seit Java 9 auch Module als primäres Sprachkonstrukt. In dieser Vorlesung werden wir uns jedoch im Wesentlichen auf die grundlegende Modularisierung mit Hilfe von Klassen und Packages von Java Programmen beschränken.

Ziele bei der Modularisierung von Code

Wiederverwendbarkeit:
Durch Modularisierung kann Code einfacher wiederverwendet werden.

Wartbarkeit: Modularer Code, bei dem einzelne Module eine kohärente Funktionalität anbieten, sind einfacher zu verstehen und zu warten. Fehler lassen sich leichter finden und beheben.

Erweiterbarkeit: Die Modularisierung von Code ermöglicht es, neue Funktionalitäten hinzuzufügen, ohne bestehenden Code zu verändern.

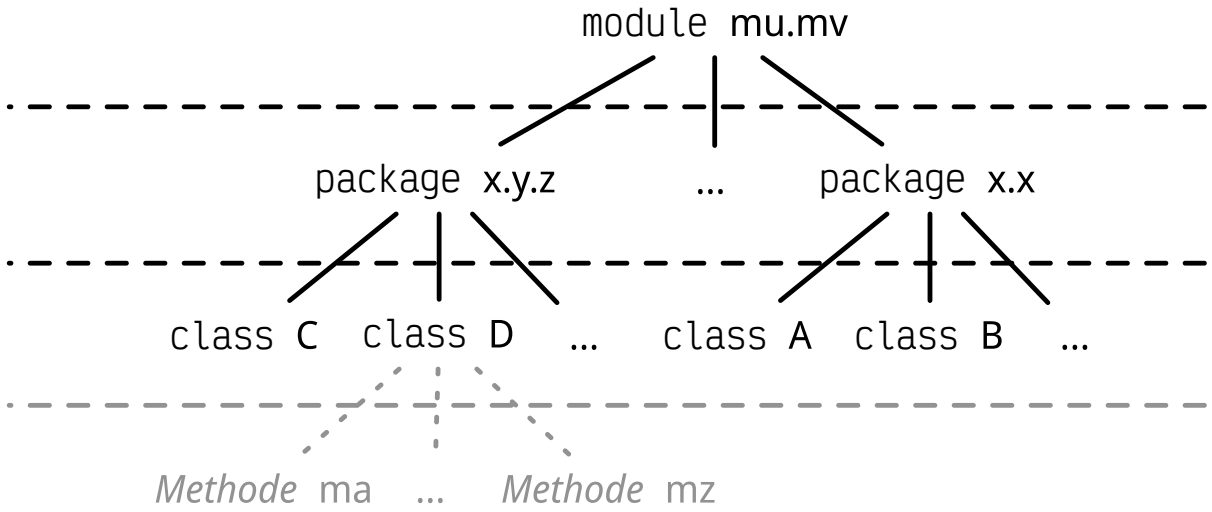
Kollaborative Entwicklung:
Durch eine Modularisierung ist es möglich effektiv mit mehreren Personen an einem Projekt zu arbeiten. Jede Person kann an einem Modul arbeiten, ohne die anderen Module zu beeinflussen.

Im Prinzip haben wir bisher nur Methoden zur Strukturierung bzw. Modularisierung kennen gelernt.

2. Modularisierung in Java

Einführung in Java: Imports, Packages und Sichtbarkeiten

Modularisierungsebenen in Java



Einzelnen Methoden erlauben zwar bereits eine Modularisierung des Codes, da diese aber für sich nicht wiederverwendbar sind (es ist nicht möglich eine Methode alleine in einer Datei zu speichern und in einem anderen Kontext zu nutzen), ist es notwendig, diese in Klassen zu organisieren. Klassen, welche in einzelnen Dateien gespeichert werden, erlauben dann eine Wiederverwendung des Codes.

Grundlegende Konzepte und Mechanismen zur Modularisierung von Java Programmen

Klassen: Klassen sind die Bausteine von Java Programmen und alles - bis auf einfachste Programme - ist in Klassen organisiert.

Packages: Packages sind Sammlungen von verwandten Klassen und Schnittstellen.

`import`s: Imports ermöglichen den Zugriff auf Klassen aus anderen Packages, ohne deren vollständigen Namen zu schreiben.

Sichtbarkeiten: Sichtbarkeiten steuern den (erlaubten) Zugriff auf Klassen, Methoden und Variablen und helfen somit beim Verbergen von Implementierungsdetails.

Im Folgenden werden wir nur ein kohärentes Subset der Modularisierungsmöglichkeiten von Java Programmen betrachten. Insbesondere werden wir uns auf die wesentlichen Eigenschaften der genannten Konzepte und Mechanismen beschränken:

Klassen in Java

1. Klassen sind die grundlegenden Bausteine von Java Programmen.
2. Eine Klasse wird mit dem Schlüsselwort `class` deklariert.
3. Eine Klasse kann Felder (Variablen) und Methoden enthalten.
4. Eine Klasse wird in einer Datei mit dem Namen der Klasse (+*.java*) gespeichert.

> **⚠ Warnung**
>
> Die Hauptfunktion einer Klasse in Java ist es als Schablone für Objekte zu dienen, die eine gemeinsame Struktur und Verhalten haben. Dies werden wir aber erst später in der Vorlesung besprechen. Für den Moment nutzen wir Klassen „nur" zur Strukturierung bzw. Modularisierung des Codes.

Im einfachsten Fall sind die Klassen eines Java Programms alle im selben Verzeichnis gespeichert.

Dies erlaubt eine *direkte* Verwendung der Methoden der anderen Klassen durch Angabe des Klassennamens und des Methodennamens. (Vergleichbar mit der Verwendung von `Double.parseDouble` etc.)

Datei: *MyMath.java*

```
class MyMath {
    static final int ANSWER_TO_EVERYTHING = 42;
    static double fibonacci(int n) { ... }
    static boolean isPrim(int n) { ... }
}
```

Datei: *Main.java*

```
void main() {
    IO.println(MyMath.fibonacci(10));
}
```

Syntax:

```
class <KlassenName> {
    <Attribute (gel. auch Felder genannt)>*
    <Methoden>*
}
```

- Der `Klassenname` muss ein gültiger Bezeichner sein und mit dem Dateinamen (+ .java) übereinstimmen.
- Klassennamen beginnen in Java - per Konvention - immer mit einem Großbuchstaben (🇺🇸 *UpperCamelCase*).

Interfaces in Java

- Seit Java 8 (in Verbindung mit weiteren Ergänzungen in Java 9) können auch `interface`s zum Organisieren von Code verwendet werden.
- \

  Beispiel:

  ### Datei: *MyMath.java*

```
interface MyMath {
    static final int ANSWER_TO_EVERYTHING = 42;
    static double fibonacci(int n) { ... }
    static boolean isPrim(int n) { ... }
}
```

  ### Datei: *Main.java*

```
void main() {
    IO.println(MyMath.fibonacci(10));
}
```

Die Verwendung von Interfaces zu *reinen Strukturierungszwecken* ist jedoch unüblich.

Wir werden uns Interfaces in einer späteren Vorlesung genauer ansehen, wenn wir objekt-orientierte Programmierung in Java detaillierter besprechen.

Statische Methoden und statische Attribute von Klassen und Interfaces

- **Statische Methoden** gehören zur Klasse/Interface als solches.
- **Statische Attribute** gehören zur Klasse/Interface als solches.

Syntax:

```
static <returnType> <methodName>(<parameters>) { <body> }
static final <type> <name> = <value>;
```

Das Java Development Kit (JDK) enthält viele Klassen – z. B. `java.lang.Math`, `java.lang.System`, `java.io.File`, `java.io.IO`, `java.util.Arrays` etc. – mit statischen Methoden - z. B. `parseDouble(...)` – und Attributen.

Hinweis

Wenn Sie das `static` Schlüsselwort vergessen, dann haben Sie Instanzmethoden und Instanzattribute. Diese können Sie nur nutzen, nachdem Sie basierend auf der Klasse ein Objekt instantiiert haben.

✎ Übung - Erste Refaktorisierung des Codes

2.1. Aufteilung in Klassen

Nehmen Sie Ihren Code (Berechnung der Fibonacci-Zahlen, Fakultät, Kubikwurzel und Primzahltest) und ordnen Sie diesen einer Klasse zu. Überlegen Sie sich einen geeigneten Namen für die Klasse und speichern Sie die Klasse in einer entsprechenden Datei. In einer zweiten Datei (`Main.java`) schreiben Sie eine `main`-Methode, die - basierenden auf Kommandozeilenparametern - die passenden Methoden der Klasse aufruft und die Ergebnisse auf der Konsole ausgibt. Die `main` Methode soll dabei die grundlegende Fehlerbehandlung übernehmen, falls die Kommandozeilenargumente nicht passen.

Denken Sie daran, dass die Methoden und Attribute der Klasse statisch (`static`) sein müssen, damit Sie diese ohne Instanziierung eines Objekts der Klasse aufrufen können.

Beispiel

```
$ java Main.java cbrt 1000 isPrim 97 fibonacci 30 ack 1
1000.0^⅓ = 10.0
isPrim(97) = true
fibonnci(30) = 832040
[error] Ungültige Funktion: ack
```

Java Packages

- **Packages** sind Sammlungen von **verwandten Klassen** und **Schnittstellen**.
- Sie helfen, Code in **logische Gruppen** zu organisieren und **Namenskonflikte** zu vermeiden.
- Vergleichbar mit **Ordnern** für Dateien in einem Dateisystem.

Syntax & Semantik: `package <packageName>;`

- Die Packagedeklaration steht am Anfang einer Java-Datei.
- Per Konvention erfolgt die Benennung in umgekehrter Domain-Reihenfolge.
- Der Packagename muss die Verzeichnisstruktur widerspiegeln.

Beispiel

```
1 package de.dhbw.mannheim.vl.programmierung;
2 class Klasse { ... }
```

Imports in Java

`import`s: ermöglichen den Zugriff auf Klassen aus anderen Packages, ohne deren vollständigen Namen zu schreiben.

Syntax:
```
import <packageName>.<className>;
```

- Erleichtert das Lesen und Schreiben des Codes, da der vollständige Klassenname nicht jedes Mal geschrieben werden muss.
- Das Package `java.lang` wird immer automatisch importiert (enthält u. a. die Klassen `String`, `Math`, etc.)

Java unterstützt auch ein Wildcard-Import, z. B. `import java.util.*;`. Dies sollte jedoch in nicht-trivialem Code vermieden werden, da es zu Konflikten führen kann.

`import static`: ermöglicht den Import von statischen Methoden und Attributen. Danach kann ohne Angabe des Klassennamens auf die Methode bzw. das Attribut zugegriffen werden.

Syntax:
```
import static <packageName>.<className>.<methodName>;
import static <packageName>.<className>.<attribute>;
```

`import module`: (Seit Java 23) ermöglicht den Import aller Klassen eine Moduls.

Syntax:
```
import modul <moduleName>;
```

Beispiele für Imports

Spezifischer Import einer Klasse:

```
import java.math.BigDecimal;

... {
    var one = BigDecimal.ONE;
}
```

Import aller Klassen (und Interfaces) des Packages:

```
import java.math.*;

... {
    var one = BigDecimal.ONE;
}
```

Import einer Klassenmethode (statisch):

```
import static java.lang.Math.sqrt;

... {
    var x = sqrt(2);
}
```

Import eines Klassenattributs (statisch):

```
import static java.lang.System.out;

... {
    out.println("Hello World!");
}
```

Import eines Java Modules (ab Java 23):

```
import module java.base;

... {
    IO.println(BigDecimal.ONE);
}
```

Hinweis

Ein Java-Skript/die JShell importiert immer implizit:

```
import module java.base;
```

Wenn Sie in der JShell also auch `println` und `readln` direkt verwenden wollen, dann müssen Sie lediglich `import static java.io.IO.*;` hinzufügen.

Sichtbarkeiten (Access Modifiers)

Um festzulegen, wer auf Klassen, Methoden und Variablen zugreifen kann, verwendet Java **Sichtbarkeiten** (Access Modifiers). Dies ist ein Konzept aus dem Bereich *Programming-in-the-Large*. Für kleinere Projekte, bei denen alle Klassen im selben Package sind, ist dies nicht relevant.

Die vier Sichtbarkeiten in Java sind:

1. `public`: Zugriff von überall
2. `protected`: Zugriff innerhalb des gleichen Packages und von Subklassen
3. *(package-private)*: Zugriff nur innerhalb des gleichen Packages
4. `private`: Zugriff nur innerhalb der gleichen Klasse

Sichtbarkeiten und deren Verwendung

Abstraktes Beispiel:

```java
public class PublicClass {
    public int publicVar;        // Zugriff von überall
    protected int protectedVar;  // Zugriff innerhalb des Packages und Subklassen
    int defaultVar;              // Zugriff nur im selben Package
    private int privateVar;      // Zugriff nur innerhalb dieser Klasse
}
```

Konkretes Beispiel:

```java
public class MyMath {
    public static int THE_ANSWER = 42;
    private static double cbrt(double x,double guess, int steps) { ... }
    public static double cbrt(double x) { cbrt(x,1.0,1); }
}
```

```java
public interface MyMath {
    static int THE_ANSWER = 42;
    private static double cbrt(double x,double guess, int steps) { ... }
    static double cbrt(double x) { cbrt(x,1.0,1); }
}
```

Java interfaces kennen nur die Sichtbarkeiten `public` und `private`. Wenn keine Sichtbarkeit angegeben wird, ist die Methode bzw. das Attribut implizit `public`.

Anwendung in der Praxis

- `public`: Offene API, z. B. für Libraries.
- `private`: dient der Kapselung z. B. interne Hilfsmethoden und interner Zustand.
- `protected`: Ermöglicht Vererbung und Zugriff für verwandte Klassen.
- `<default bzw. keine explizite Angabe>`: Für interne Logik innerhalb eines Packages.

Beispiel für die konkrete Anwendung

Verzeichnis mit der fachlichen Logik für mathematische Funktionen:

```
package de.dhbw.mannheim.calculator.math;

public class Functions {
    public static double cbrt(double x) { ... }
}
```

Code mit der Logik für die Interaktion mit dem Benutzer:

```
package de.dhbw.mannheim.calculator;

import de.dhbw.mannheim.calculator.math.Functions;

public class Main {
    public static void main(String[] args) {
        Functions.cbrt(Double.parseDouble(args[0]));
    }
}
```

Best Practices für Packages und Sichtbarkeiten

- Organisiere Klassen logisch in Packages.
- Die beiden bei weitem häufigsten Sichtbarkeiten sind `public` und `private`.
- Nutze `public` nur bei notwendigen Klassen und Methoden.
- Halte Klassenvariablen **privat**, um Daten zu kapseln.
- Methoden, die nur innerhalb einer Klasse verwendet werden, sollten **private** sein.
- Vermeide übermäßige Imports (`import java.util.\*;` kann zu Konflikten führen).

Zusammenfassung

- **Packages** gruppieren verwandte Klassen und vermeiden Namenskonflikte.
- **Imports** erlauben das Verwenden von Klassen aus anderen Packages.
- **Sichtbarkeiten** steuern den Zugriff und helfen beim Schutz der Daten.

✎ Übung

2.2. Modularisierung der Codebasis

Verschieben Sie Ihre Klasse mit den mathematischen Funktionen in das package `math`. Die Datei mit der `main` Methode bleibt an ihrem Platz. Fügen Sie ggf. ein import Statement hinzu.

Wie müssen Sie Ihren Code ändern, wenn Sie innerhalb der Datei `Main.java` direkt auf die Methoden zugreifen wollen ohne jedes mal den Klassennamen voranstellen zu müssen?