2. Data Hiding und co

Quelle: ep2-02_Data-Hiding_Objekterzeugung_Datensatz.pdf
Beinhaltet: Data-Hiding, Objekterzeugung, Datensatz

Data Hiding

1. Außen- und Innensicht

- **Außensicht**:
  - Definition des **abstrakten Datentyps (ADT)** aus Anwendersicht
  - Sichtbar ist nur, was für die Verwendung **notwendig** ist
  - Fokus auf **Benutzerfreundlichkeit** und **Schnittstelle**
- **Innensicht**:
  - Interne **Implementierung** des ADT
  - **Alle Details sichtbar** (Variablen, Methoden, Algorithmen etc.)
  - Fokus auf **Effizienz** und **Wartbarkeit**
- **Unterschiedliche Sichtbarkeiten → Data-Hiding**
  - Ziel: **Trennung** von Schnittstelle (Außensicht) und Implementierung (Innensicht)
  - Bessere **Modularität** und **Wartbarkeit**

2. Data-Hiding

- **Zugriffsmodifikatoren**:
  - `public`:
    - Gehört zur Außen- **und** Innensicht
    - Überall zugreifbar
  - `private`:
    - Gehört nur zur **Innensicht**
    - Nur innerhalb der **eigenen Klasse** zugreifbar
- **Änderung der Innensicht** bei **gleichbleibender Außensicht**:
  - Anwendungen bleiben **unverändert**
- **Änderung der Außensicht**:
  - Anwendungen müssen ggf. **angepasst** werden
- **Praxisempfehlung**:
  - Möglichst viele Methoden und Variablen als `private` deklarieren

- Dadurch **bessere Wartbarkeit**, auch wenn es anfangs als Nachteil empfunden werden kann

3. Sichtbarkeit auf Klassenebene

- **Zugriff zwischen Objekten derselben Klasse**:
  - Auch `private` Mitglieder eines anderen Objekts sind zugreifbar
  - Beispiel:
    java
    KopierenBearbeiten
    `public class A { private int x; public int add(A a) { return x + a.x; // Zugriff auf privates x von a erlaubt } }`
  - Erklärung: `a` ist vom Typ der Klasse `A`, daher ist Zugriff auf dessen private Felder innerhalb von `A` erlaubt
- **Fazit**:
  - **Außen-/Innensicht** → objektbezogen
  - `public` / `private` → klassenbezogen
  - Dies kann zu **scheinbar widersprüchlichem Verhalten** führen, ist aber durch das Klassenmodell gerechtfertigt

---

Klassen erstellen

Sichtbarkeit von Klassen: `public` Modifier

- `public class`:
  - Klasse ist allgemein verwendbar
  - Normalfall: genau eine `public` Klasse pro Datei
  - Klassenname = Dateiname (bis auf Dateiendung)
- **Ohne** `public` vor `class`:
  - Klasse ist nur im selben Ordner (Package) sichtbar
  - Dient als Hilfsklasse
- **Ausnahme**:
  - Bei Data-Hiding kann von der Standardregel abgewichen werden

Objekterzeugung mit `new`

- Ausführung von `new A()`:
  - Speicherbereich für Objektvariablen und Identität wird reserviert
  - Speicher wird mit Null-Werten vorinitialisiert
  - Ein Konstruktor der Klasse `A` wird zur Initialisierung ausgeführt
  - Eine Referenz auf den Speicherbereich (das Objekt) wird zurückgegeben
- **Identität von Objekten**:
  - Wenn $x == y$ wahr, dann referenzieren x und y dasselbe Objekt

Konstruktoren

- Konstruktor ist ähnlich wie Methode, hat:
  - **gleichen Namen wie die Klasse**
  - **keinen Ergebnistyp**
  - **Parameter** zur Initialisierung der Objektvariablen
- **Beispiel**:

```
public class Point {
    private int x, y;

    public Point(int initX, int initY) {
        x = initX;
        y = initY;
    }
}
```

- Wird durch `new Point(3, 5)` aufgerufen

- Initialisiert Objekt mit $x = 3, y = 5$

Überladene Konstruktoren und Default-Konstruktor

- Mehrere Konstruktoren mit unterschiedlicher Parameterliste möglich (Überladung)

```
public class Point {
    private int x, y;

    public Point(int initX, int initY) {
        x = initX;
        y = initY;
    }

    public Point() {}
}
```

- `new Point()` entspricht `new Point(0, 0)`
- **Default-Konstruktor**:
  - Wird automatisch erzeugt, wenn **kein anderer Konstruktor** vorhanden ist

Konstruktoraufruf mit `this(...)`

- Konstruktor kann andere Konstruktoren derselben Klasse aufrufen

```
public class Point {
    private int x, y;

    public Point(int initX, int initY) {
        x = initX;
        y = initY;
    }

    public Point() {
        this(1, 1);
    }

    public Point(Point p) {
        this(p.x, p.y);
    }
}
```

- Konstruktor-Aufruf mit `this(...)`:
  - **Nur als erste Anweisung im Konstruktor erlaubt**
- Beispiele:
  - `new Point(3, 5)`

- `new Point()`
- `new Point(new Point())`

Selbstreferenz mit `this`

- `this` referenziert das **aktuelle Objekt**, in dem sich der Code gerade befindet
- Wird oft zur Unterscheidung von Parameter- und Attributnamen genutzt

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point(Point p) {
        this(p.x, p.y);
    }

    public Point copy() {
        return new Point(this);
    }
}
```

- `this` ist eine **Pseudovariable**:
  - Nur **lesbar**, nicht überschreibbar
- In `this(...)` handelt es sich um **einen Konstruktoraufruf**, **nicht** um eine Selbstreferenz

---

Datenstruktur != abstrakter Datentyp

Datenstruktur

- Beschreibt, **wie Daten zusammenhängen**, **wie sie auffindbar sind** und **wie Operationen darauf zugreifen**
- Offene Aspekte:
  - verwendete Programmiersprache
  - konkrete Datentypen
  - mögliche Größenbeschränkungen

Abstrakter Datentyp (ADT)

- **Außensicht**: wie Objekte verwendet werden können
- Blendet **Implementierungsdetails** aus
- Lässt offen:
  - konkrete Algorithmen
  - Datenstrukturen
  - sonstige interne Details

Implementierung eines abstrakten Datentyps

- Umfasst:
  - konkrete Algorithmen
  - verwendete Datenstrukturen
- Klärt offene Punkte aus Sicht von ADT und Datenstruktur
- **Übergang zwischen ADT und Datenstruktur ist fließend**

Datensatz als Datenstruktur

- Sehr einfache Datenstruktur
- Besteht aus **zusammengehörenden Variablen**, die bei Bedarf gelesen oder geschrieben werden
- Beispiel:

```
Student:
    regNumber
    name
    mail
```

- In dieser Form **relativ uninteressant**

Datensatz als abstrakter Datentyp

- **Abstraktionsebene höher** als einfache Datenstruktur
- Fragestellungen zur Abstraktion:
  - Wie sind **Werte der Variablen eingeschränkt**?
  - Welche Variablen sind **wann lesbar**, **wann schreibbar**?
  - Bleiben Variablen **hinter der Abstraktion sichtbar**?
  - Welche Abstraktion ermöglicht eine **einfache Verwendbarkeit**?

Getter und Setter

Datensatz mit Gettern und Settern

- **Getter und Setter möglichst vermeiden**
  - Grund: lassen interne Variablenstruktur nach außen durchscheinen
  - Verstoßen gegen Prinzip der Datenkapselung
- **Beispiel**:

```
public class Student {
    private final int regNumber;
    private String name;

    public Student(int regNumber, String name) {
        this.regNumber = regNumber;
        setName(name);
    }

    public int regNumber() {
        return regNumber;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

- **regNumber**:
  - `final`, d.h. **nach Initialisierung nicht mehr veränderbar**
  - Aber: auch **nicht-finale** Variablen können nach außen **nur lesbar** gemacht werden
- Einschränkungen durch Typen auch in der **Außensicht** sichtbar
- Getter/Setter **übertragen wesentliche Funktionalität nach außen**
  - führen zu Verlust von Kontrolle und Abstraktion

Umgang mit zusammenhängenden Daten

- Zugriff erfolgt **indirekt** über Methoden, nicht direkt über Variablen
- Beispiel: Suche in einem Array von `Student`-Objekten

```
private static Student find(Student[] studs, int reg) {
    for (Student stud : studs) {
        if (stud.regNumber() == reg) {
            return stud;
        }
    }
    return new Student(reg, "Max Mustermann");
}
```

- Vergleich über **indirekten Zugriff** (hier `stud.regNumber()`)
- **Rückgabe eines vollständigen Datensatzes** (mit allen Variablen)

Funktionalität angereicherter Datensatz

`Student`-Klasse ohne Getter und Setter

- **Wesentliche Funktionalität in die Klasse verschoben**:
  - Statt Getter und Setter gibt es nun Methoden, die innerhalb der Klasse verwendet werden.
  - Vermeidet das Offenlegen der internen Datenstruktur nach außen und wahrt die Kapselung.
- **Beispiel**:

```
public class Student {
    private final int regNumber;
    private String name;
    private String mail;

    public Student(int regNumber, String name) {
        this.regNumber = regNumber;
        this.name = name;
        mail = "e" + regNumber + "@student.tuwien.ac.at";
    }

    public void showPersonalData() {
        // Anzeige der persönlichen Daten
    }

    public void editPersonalData() {
        // Bearbeiten der persönlichen Daten
    }

    public void mail(String head, String text) {
        // Funktion zum Senden einer E-Mail
    }
}
```

- **Vorteile**:
  - **Keine Getter und Setter notwendig**, da alle Zugriffe und Operationen innerhalb der Klasse bleiben.
  - **Verborgene Datenstruktur**: Die Variablen `regNumber`, `name` und `mail` sind nur innerhalb der Klasse zugänglich.
- **Funktionalitäten**:
  - `showPersonalData()`: zeigt die persönlichen Daten an.
  - `editPersonalData()`: ermöglicht das Bearbeiten der persönlichen Daten.
  - `mail()`: sendet eine E-Mail mit dem angegebenen Betreff und Text.

Prinzip der Datenkapselung

- **Getter und Setter vermeiden**: Durch das Verschieben der wesentlichen Funktionalität innerhalb der Klasse werden externe Zugriffe auf die Variablen vermieden, was die **Datenkapselung** fördert.
- Wenn **alle zugreifenden Methoden innerhalb der Klasse** sind, ist der Zugriff auf die Variablen kontrolliert und sicher.

Idee hinter objektorientierter Programmierung

- **Schwerpunkt auf Funktionalität**, nicht auf dem Datensatz:
  - Der **Datensatz** bleibt **hinter der Funktionalität** gänzlich abstrakt.
  - Fokus liegt darauf, wie **Operationen und Funktionen** auf den Daten ausgeführt werden, nicht auf der reinen Speicherung der Daten.
- **Software-Objekt simuliert ein „reales Objekt“**:
  - Ein Software-Objekt muss **nicht nur konkrete, materielle** Objekte abbilden, sondern auch **immaterielle** Objekte oder Konzepte.
  - Es geht darum, **nur die in der Software relevanten Eigenschaften** eines „realen Objekts“ zu simulieren.
- **Modellierte Objekte** sind:
  - Häufig **mit Funktionalität angereicherte Datensätze**:
    - Das bedeutet, die Daten sind nicht isoliert, sondern sie haben eine **funktionale Bedeutung**, die es ermöglicht, Operationen oder Methoden darauf anzuwenden.