Tagesprogramm

dynamisches Binden versus static und private

final Variablen, Klassen und Methoden

abstrakte Klassen und Methoden

Dynamisches und statisches Binden

```
private static void test(IntContainer container) {
    container.add(...);              // dynamisches Binden
}

    IntContainer cont = new IntTree;
    test(cont);                      // statisches Binden
```

statisches Binden: Compiler weiß, welche Methode ausgeführt wird

→ Laufzeiteffizienz

dynamisches Binden: Laufzeitsystem bestimmt auszuführende Methode

→ Flexibilität und einfachere Wartbarkeit

static

```
public class X {

    public int objectVar = 0;       // public vermeiden
    public static int classVar = 0;

    public void objectMeth() { objectVar++; classVar++; }
    public static void classMeth() { classVar++; }
}
```

ohne `static`: gehört zu Objekt (`X obj = new X();`)

- Zugriff über Objekt (`obj.objectMeth()`, `obj.objectVar`)
- Methodenaufrufe im Allgemeinen dynamisch gebunden

mit `static`: gehört zu Klasse

- Zugriff über Klasse (`X.classMeth()`, `X.classVar`)
- Methodenaufrufe immer statisch gebunden

private

Sichtbarkeit:

`private` nur in eigener Klasse sichtbar

`public` überall sichtbar

Binden von Methoden:

`private` Methoden immer statisch gebunden (auch ohne `static`)

Verwendung von Variablen:

`private` Variablen von Programmier(in) einfacher zu verstehen

`private` Variablen von Compiler manchmal besser optimierbar

Aufgabe: Warum private Variablen?

Warum versteht man private Variablen meist einfacher als nicht-private?

A Weil niemand von draußen dazwischenpfuschen kann.

B Weil Klassenzusammenhalt und Objektkopplung gestärkt werden.

C Weil nur die Innensicht betrachtet werden muss.

D Weil ein geringerer Abstraktionsgrad immer einfacher ist.

Spezialfall: public Klassen

nicht geschachtelte Klassen:

in jeder Datei nur eine `public` Klasse (Dateiname $\approx$ Klassenname)

nur `public` Klassen von außen allgemein verwendbar

nicht-`public` Klassen nur im selben Paket verwendbar

`public` ist Normalfall

geschachtelte Klassen:

Sichtbarkeit so wie auch bei Variablen, Methoden, Konstruktoren

aber Einschränkungen von `private` können teilweise umgangen werden

trotzdem geschachtelte Klassen meist `private` (oder ohne Modifier)

Aufgabe: Sichtbarkeit von Klassen

Warum sind nicht geschachtelte Klassen meist `public`,
geschachtelte dagegen meist `private`?

A Weil nicht verwendbare Klassen nicht sinnvoll sind.

B Weil bei geschachtelten Klassen vorwiegend die Innenansicht zählt.

C Weil man nicht-`public` Klassen meist gut schachteln kann.

D Hat sich im Laufe der Zeit einfach so ergeben.

final auf Variable

Einschränkung der Änderbarbeit:

Wert einer `final` Variablen nach Initialisierung nicht änderbar

Besonderheiten von final Klassenvariablen:

`final static` Variablen vom Compiler sehr gut optimierbar

müssen bei der Deklaration initialisiert werden

können vorkommen, wo sonst nur Literale erlaubt sind

können in Interfaces deklariert werden (wo normale Variablen verboten sind)

sind häufig `public`

Sonderstellung meist durch Großschreibung hervorgehoben

```
public static final double PI = 3.14;
```

final auf Methode

```
public class X {
    public final void finalMethod() {...}
}
```

Methode kann in Unterklasse **nicht überschrieben** werden

darf nicht `static` sein

ermöglicht statisches Binden

kann in Obertyp (Interface oder Klasse) als nicht-`final` deklariert sein

einzelne `final` Methoden nur selten verwendet

final auf Klasse

```
public final class X {
    public void finalMethod() {...}
}
```

keine Unterklassen von `final` Klassen erlaubt

alle Methoden in `final` Klasse sind `final`

oft sinnvoll

je nach Programmierstil gar nicht bis häufig eingesetzt

Aufgabe: Bedeutung von final

Hat `final` auf Variablen dieselbe Bedeutung wie auf Klassen und Methoden?

A Ja, weil in allen Fällen Änderungen vermieden werden.

B Nein, weil es bei Variablen um dynamische Eigenschaften geht, bei Klassen und Methoden dagegen um statische.

C Nein, weil `static final` Variablen in der Praxis wichtig sind, final Klassen und (`static`) Methoden dagegen nicht.

D Ja, weil es um die finale Zielsetzung geht.

abstract

```
public abstract class X {
    private int v;

    public void concreteMethod() {...}
    public abstract void abstractMethod();
}

public abstract class Y extends X {
    @Override
    public void abstractMethod() {...}
}
```

abstrakte Klasse kann abstrakte (nicht implementierte) Methoden enthalten

kombiniert Eigenschaften von Interfaces mit denen von Klassen

`new X()` nicht erlaubt

abstrakte Klasse nur zusammen mit nicht-abstrakten Unterklassen sinnvoll

Empfehlungen

`private` überall wo möglich, `public` überall wo nötig

nicht-geschachtelte Klassen fast immer `public`

`static` eher vermeiden, manchmal aber nötig

`static final` auf Variablen (= Konstanten) nutzen

`final` auf einzelnen Methoden eher vermeiden

`final` auf Klassen sinnvoll,
nicht-abstrakte Klassen so verwenden, als ob sie `final` wären

abstrakte Klassen eher meiden, Interfaces vorziehen

keine Angst vor dynamischem Binden

Aufgabe: Begründung der Empfehlungen

Suchen Sie in Gruppen zu 2 bis 3 Personen Begründungen für die Empfehlungen.