HSLU Hochschule Luzern

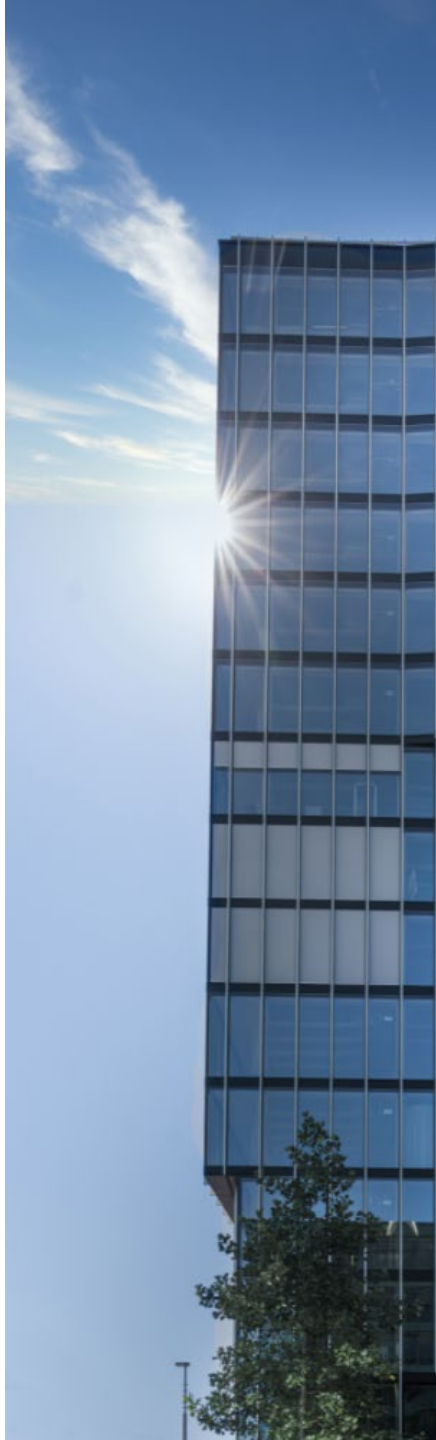
Objektorientierte Programmierung

Datenkapselung und Information Hiding

Objektorientiertes Design

Roland Gisler

FH Zentralschweiz



Inhalt

- Datenkapselung
- Zugriffsmodifizierer
- Setter- und Getter-Methoden
- Information Hiding

Lernziele

- Sie beherrschen das Konzept der Datenkapselung.
- Sie kennen die Wirkung der Zugriffsmodifizierer in Java und können diese adäquat einsetzen.
- Sie verstehen das Konzept der Setter- und Getter-Methoden.
- Sie verstehen die Designaspekte des Information Hiding.

Preambel: Datenkapselung vs. Information Hiding

- In vielen Quellen werden diese beiden Begriffe als Synonyme verwendet.
- Sie beschreiben bei differenzierter Betrachtung aber verwandte, aufeinander aufbauende, und nicht absolut identische Konzepte.
  - Ein Grund dafür ist, dass die Begriffe bis weit in die 1970er-Jahre zurückgehen, und man sich häufig auf die Original-quellen beruft.
- In diesem Input werden die Begriffe – angepasst an die heutige Situation und modernen Sprachkonzepte – etwas differenzierter dargestellt und erklärt.

Datenkapselung

Abstraktion und Modularisierung mit Klassen

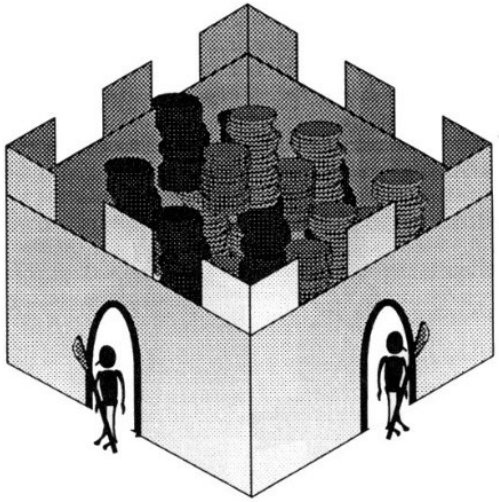
- In den meisten objektorientierten Sprachen verwendet man Klassen, um darin Daten (➔Zustand in Attributen) und Funktionen (➔Verhalten in Methoden) **zusammengefasst** abzubilden.
- Klassen bilden somit auf der tiefsten Ebene ein erstes Mittel zur so genannten ➔Modularisierung, welche wiederum eine wichtige Basis für eine mögliche, einfachere Wiederverwendung darstellt.
- Klassen (bzw. Module) interagieren dann gemeinsam miteinander und untereinander durch:
  - den Ausstauch von Daten.
  - den Aufruf von Methoden.

Kopplung zwischen Klassen

- Wenn jede Klasse auf die Daten und Methoden jeder anderen Klasse Zugriff hätte, wäre eine extrem starke, gegenseitige Vernetzung und Abhängigkeit das Resultat!
  - Man bezeichnet das als **zu** starke ➔Kopplung.
- Es scheint logisch, dass eine starke Kopplung viele negative Aspekte hat, und man **wünscht** sich stattdessen eine möglichst **lose Kopplung**, weil diese:
  - Ermöglicht eine **einfachere** Wiederverwendung von Klassen.
  - Erlaubt den **leichteren** Austausch von einzelnen Klassen.
- Ein erstes einfaches Werkzeug, um die Kopplung zu kontrollieren ist der Einsatz von **Zugriffsmodifizierern**, welche eine feingranulare Kontrolle der gegenseitigen Zugriffe ermöglichen.

Datenkapselung

- Damit erreicht man eine **stärkere** Form der Datenkapselung: Attribute werden nicht nur einfach gemeinsam in einer Klasse zusammengefasst (➔Kohäsion), sondern durch explizite Zugriffsmodifizierer feingranular und differenziert gekapselt, d.h. vor der unkontrollierten Manipulation von aussen geschützt.



Zugriffsmodifizierer

Einsatzmöglichkeiten von Zugriffsmodifizieren

- `public`, `private`, `protected`…
- Die Zugriffsmodifizierer legen fest **ob** ein Element überhaupt, und wenn ja, für welche anderen Klassen sichtbar ist.
  - Nur auf sichtbare Elemente kann zugegriffen werden.
- Zugriffsmodifizierer können bei Java auf folgenden Elementen verwendet werden:
  - Attribute
  - Methoden
  - Klassen (auch `abstract`e ~)
  - Interfaces (sind implizit immer `public`)

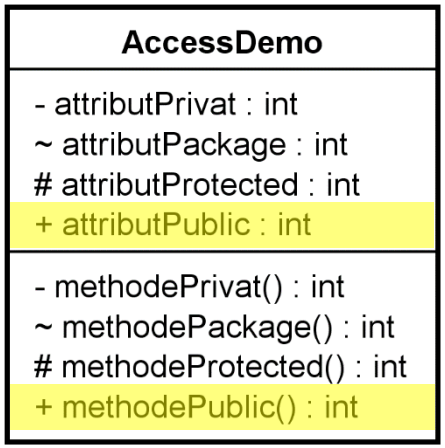
`public` - Zugriff

- Schlüsselwort: **`public`**
- Sichtbar in der Klasse selbst und auch in allen anderen Klassen, egal in welchem Package.

  ➔ **Maximale** Sichtbarkeit, vollständig öffentlich*.

  *Ausnahme: Bei Nutzung des **J**ava **P**latform **M**odul **S**ystems (JPMS) lässt sich die Sichtbarkeit wieder einschränken.
- Was einmal öffentlich ist, kann man nicht mehr (oder nur mit sehr grossem Aufwand) wieder zurücknehmen oder verändern!
- Sehr selten bei Attributen eingesetzt.
- Darstellung in UML: **+** («Plus»-Zeichen)
- Beispiel (anhand einer Methode):

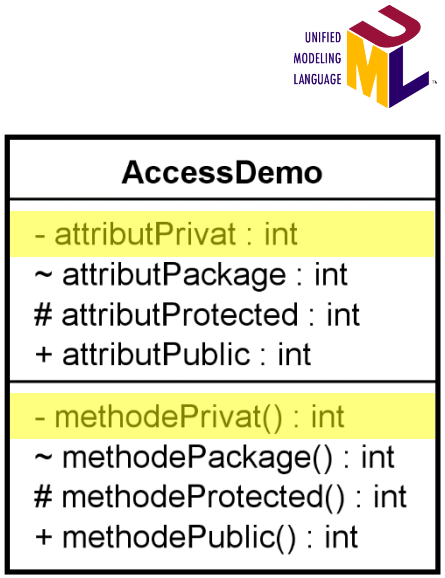
```
public int methodePublic() {
    ...
}
```



`private` - Zugriff

- Schlüsselwort: **`private`**
- Ausschliesslich sichtbar in der Klasse selbst, nirgends sonst!
  ➔ Maximale Kapselung, vollständig privat.
- Kann somit ohne Einfluss auf Dritte jederzeit verändert werden.
- Sehr häufig bei Attributen, aber auch internen (Hilfs-)Methoden.
- Darstellung in UML: **-** («Minus»-Zeichen)
- Beispiel (anhand eines Attributes):

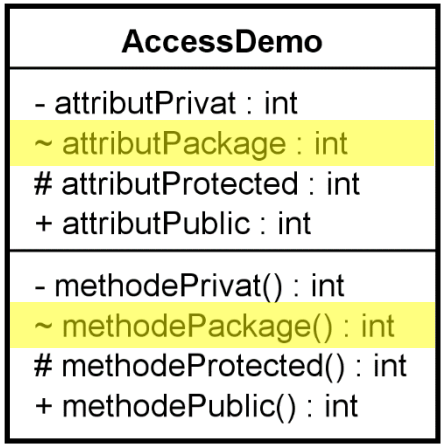
```
private int attributPrivat;
```



package – Zugriff (default)

- Schlüsselwort: **`<keines>`**
- Sichtbar in der Klasse selbst, und für jede Klasse des **selben** Packages zu dem diese Klasse gehört.
  - In manchen Sprachen bezeichnet man das auch als „friends".
- **Achtung**: Weil diese Sichtbarkeit **kein** explizites Schlüsselwort kennt, ist sie auch der Default wenn man es vergisst!
- Einsatz seltener, meist bei Methoden oder Klassen.
- Darstellung in UML: **~** («Tilde»-Zeichen)
- Beispiel (anhand einer Methode):

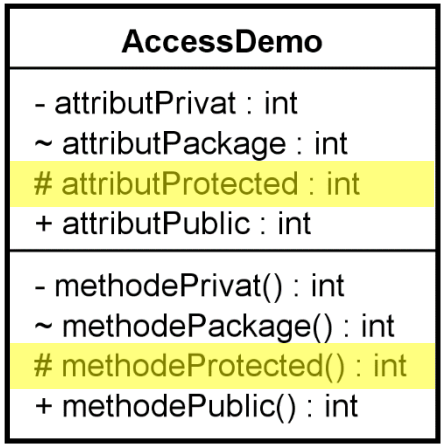
```
int methodePackage() {
    ...
}
```



`protected` - Zugriff

- Schlüsselwort: **`protected`**
- Sichtbarkeit wie bei «package», zusätzlich aber auch in allen davon spezialisierten Klassen (egal in welchem Package).
  - Konzept der ➔Vererbung (Input folgt später)
- Einsatz selten, und eher bei Methoden.
- Darstellung in UML: **#** («Raute/Doppelkreuz»-Zeichen)
- Beispiel (anhand ein einer Methode):

```
protected int methodeProtected() {
    ...
}
```



Übersicht: Zugriffsmodifizierer - Matrix

Zugriff / Modifizierer	Eigene Klasse	Klasse im gleichen Package	Unter-klasse* aus anderem Package	Nicht-Unter-klasse* aus anderem Package
`private`	ja	nein	nein	nein
*<kein**>*	ja	ja	nein	nein
`protected`	ja	ja	ja***	nein
`public`	ja	ja	ja***	ja***

* Unterklasse: Spezialisierung, Konzept der ➔ Vererbung (folgt später)
** Die sogenannte «Package»-Sichtbarkeit ist **ohne** Schlüsselwort als Default gegeben.
*** Die Klasse muss dann natürlich importiert (sichtbar gemacht) werden

Setter und Getter

Zugriffsmethoden auf Attribute

- In der Objektorientierung wird in der Regel **nicht direkt** auf Attribute einer Klasse zugegriffen, sondern immer über Zugriffsmethoden.
  - Das ist ein grundlegendes Konzept der ➔**Datenkapselung.**
- Dadurch kann man die Zugriffe auch differenzieren in
  - **lesenden** Zugriff (Getter, Query, Accessor).
  - **schreibenden** Zugriff (Setter, Mutator).
- Je nach implementierten Methoden kann auf ein Attribut nur lesend, nur schreibend oder auf beide Arten zugegriffen werden.
  - Diese Differenzierung wäre bei einem Direktzugriff auf das Attribut **nicht** möglich!
- Durch die Kapselung mit den Methoden bleibt die interne Implementation **verborgen** und kann jederzeit verändert werden.

Konvention für Getter-Methoden

- Sichtbarkeit des Attributes ist in der Regel **`private`**.
  - Zugriff nur aus Objekten der gleichen Klasse.
- Sichtbarkeit der **`get`**-Methode ist häufig **`public`**.
  - Alternativ: «default» (Package) oder **`protected`**.
- Der Name der Methode ergibt sich aus dem Namen des Attributes und dem Prefix **`get`**. Beispiel:
  - Ein Attribut mit dem Bezeichner **`name`** erhält eine Getter-Methode mit dem Namen **`getName()`**.
- Der Rückgabetyp der Methoden ist meist identisch mit dem Typ des Attributes, muss aber nicht!

Konvention für Setter-Methoden

- Sichtbarkeit der **`set`**-Methode ist häufig identisch mit **`get`**-Methode.
  - Muss aber nicht! Kann z.B. auch eingeschränkter sein!
- Der Name der Methode ergibt sich ebenfalls aus dem Namen des Attributes und dem Prefix **`set`**. Beispiel:
  - Attribut mit Bezeichner **`name`** erhält eine Setter-Methode mit dem Namen **`setName(…)`**.
- Der Rückgabetyp ist immer **`void`**.
- Die **`set`**-Methode hat einen formalen Parameter.
  - Dessen Typ ist meistens identisch zum Typ des Attributes.
  - Durch die Verwendung von **`this`** können der formale Parameter und das Attribut den gleichen (aussagekräftigen) Namen tragen.

Beispiel für einfache Setter- und Getter-Methoden

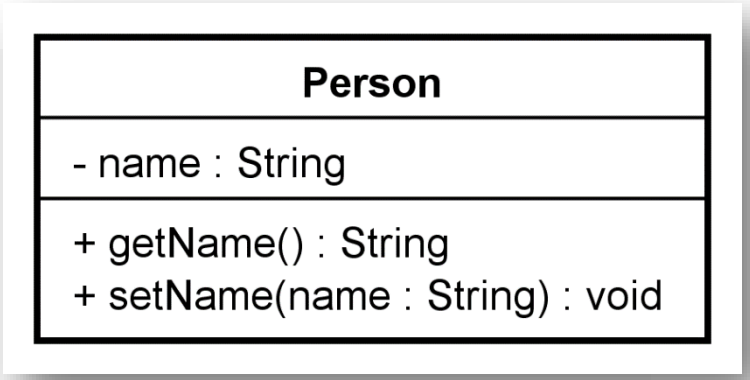
- Klasse mit einem einzigen privaten Attribut (**name**) und je einer öffentlichen Getter- und Setter-Methode:

```
public final class Person {

    private String name;

    public String getName() {
        return this.name;
    }

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



Generierte Methoden

- In der Regel können (Standard-) Setter- und Getter-Methoden von der Entwicklungsumgebung **generiert** werden.
  - Implementieren Sie diese somit **nicht** von Hand!
- Beispiel 1 - NetBeans:
  - Zu Codestelle navigieren, wo die Methoden hin sollen, dann Kontext-Menü: **`Insert Code…`** (Attribute auswählen)
  - Alternativ: **`Encapsulate Fields…`** aus dem **`Refactor`**-Menu.
- Beispiel 2 - Eclipse:
  - Attribut im Editor markieren, dann über Kontext-Menü: **`Source`** → **`Generate Setter/Getter…`** (Methoden wählen).
- Welche Getter- und Setter man generiert, sollte aber immer **wohlüberlegt** entschieden werden!
  ➔ So wenig wie möglich, so viel wie nötig!

Empfehlungen

- Sofern nicht schon die ganze Klasse als `final` markiert ist, lohnt es sich, die Setter- und Getter-Methoden `final` zu implementieren
  - Sie können dann bei Spezialisierungen (➔Vererbung) nicht mehr überschrieben werden ➔ besseres, sicheres Design.
- Die formalen Parameter von Setter-Methoden können konsequent `final` definiert werden.
  - Es macht keinen Sinn, dass diese Zuweisungen erhalten!
- Die Namensgebung der formalen Parameter von Setter-Methoden ist nicht verbindlich geregelt.
  - Identisch mit Attribut: Namenkonflikt mit `this` auflösen.
  - Mit einem Prefix: **`name`** ➔ **`newName`**, **`aName`**, **`pName`** etc.
  - Empfehlung: Egal wie, aber **einheitlich** arbeiten!
- JavaDoc: Bei Standard-Getter/Setter selten sinnvoll, aber wenn…

Wichtige Hinweise

- Allein die Tatsache, dass Attribute strikte privat deklariert, und Zugriffe ausschliesslich über Getter und Setter erfolgen, erfüllt zwar die minimalsten Anforderungen der Datenkapselung, heisst aber noch lange nicht, dass es sich dabei um gutes Design handelt!
  - **Keine Objektorientierung mit der «Brechstange»!**
- Es ist eine bewusste Differenzierung nötig zwischen

  ➔ Datenkapselung: Wohlüberlegte Zugriffe auf Daten ausschliesslich über Methoden.

  ➔ Information Hiding: Bewusstes abstrahieren oder vollständiges «verstecken» der internen Repräsentation eines Objektes.

Zugriffsmodifizierer - Empfehlungen

- Definieren Sie die Sichtbarkeit der Elemente so verschlossen wie möglich, und so offen wie nötig - aber auf jeden Fall immer bewusst!
- Die „Standardmodellierung" mit privaten Attributen und öffentlichen Setter- und Getter-Methoden ist zwar häufig, aber keineswegs der Weisheit letzter Schluss!
  - Formale Objektorientierung / Datenkapselung erfüllt, aber nicht automatisch gutes Design.
- Datenkapselung kann auch bedeuten, ganz bewusst eine Methode wegzulassen, oder den Zugriff differenzierter zu steuern.
  - z.B. um Schreibzugriffe komplett zu verhindern.
- Einfache Setter- und Getter-Methoden nicht selber schreiben, sondern durch IDE generieren lassen!

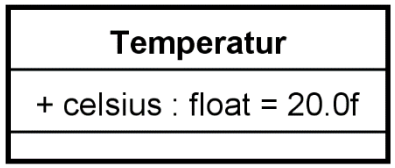
Information Hiding

Information Hiding

- Information Hiding kann man als ein auf der Datenkapselung basierendes, erweitertes Konzept bzw. Design verstehen.
- Eigentlich muss man nur wissen, welche **Methoden** von einer Klasse Angeboten werden: Die **➔Schnittstelle**, das Interface!
- Diese Schnittstelle soll möglichst **einfach** und **schmal** sein.
  - Beinhaltet auch die Datentypen die als Parameter oder Rückgabewerte verwendet werden!
- Details der internen Realisation interessieren hingegen nicht!
- Daraus resultiert eine **losere Kopplung** zwischen den Klassen und damit eine grössere Freiheit in der (internen) Implementation einer Klasse.

Beispiel ohne Datenkapselung – Schlecht!

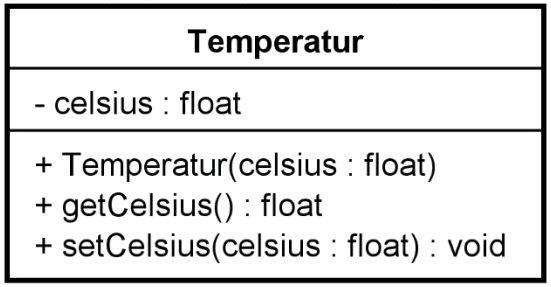
- Die folgende Klasse verfügt weder über Datenkapselung noch über Information Hiding: **Schlechtes** Beispiel!



- Das Attribut **`celsius`** (public) kann von anderen Klassen beliebig gelesen und geschrieben werden!
- Somit ist weder eine Kontrolle des Wertes noch jemals eine Änderung des Namens oder des Typs möglich.
  - Zumindest nicht ohne umfangreiches ➔Refactoring aller Klassen die **`Temperatur`** verwenden.

Beispiel mit Datenkapselung - Gut

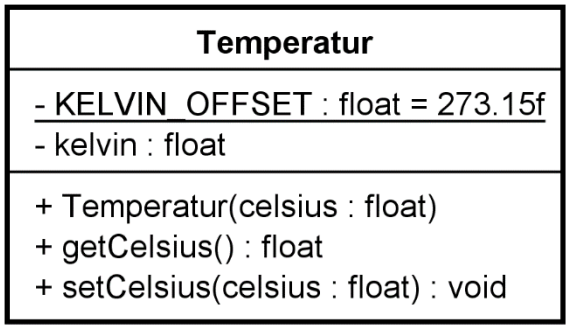
- Dasselbe Beispiel mit **fundamentaler** Datenkapselung:



- Das Attribut **`celsius`** ist **privat** und somit geschützt.
- Es existieren Setter- und Getter-Methoden, die eine Kontrolle des Zugriffes auf das Attribut erlauben.
- Lese- und Schreibzugriff sind getrennt kontrollierbar.
- Die Schnittstellen lassen den Schluss zu, dass die Temperatur intern tatsächlich auch als **`float`**, und in der Einheit Grad Celsius gespeichert wird.

Beispiel mit Information Hiding – Noch besser!

- Weiter **verbessert** mit Information Hiding:



- Die Schnittstelle ist zwar **unverändert**, tatsächlich wird die Temperatur jetzt aber intern in Kelvin gespeichert!
- In diesem einfachen Beispiel eher sinnlos, könnte man trotzdem den internen Datentyp z.B. auch auf `double` verändern.
- Spätestens wenn man z.B. noch `get-` und `setFahrenheit(…)` ergänzen würde, ist die interne Repräsentation völlig «versteckt».

Datenkapselung / Information Hiding - Empfehlungen

- Fundamentale Grundlage stellen möglichst private Attribute und ein Minimum (sehr vereinfacht formuliert) an öffentlichen Methoden (➔schmale Schnittstelle) dar.
- Bei den Parametern und Rückgabewerten der Methoden achtet man darauf, dass diese nicht zu viele Information über die interne Repräsentation (oder gar direkte Referenzen!) preis geben.
- Gegebenenfalls **kopiert** man vorgängig die Daten, welche exportiert werden, so dass sich Änderungen daran **nicht** auf die interne Repräsentation auswirken können.
  - Passiert bei elementaren Datentypen automatisch.
  - Bei Objekten (nur die Referenzen werden kopiert!) ggf. explizit umzusetzen, alternativ ➔ Immutable Objects.

Zusammenfassung

- Datenkapselung beschreibt die Zusammenfassung von Daten (Attributen) und Methoden (Verhalten) in einer Klasse.
- Die Zugriffsmodifizierer **`private`**, «*package default*», **`protected`** und **`public`** werden für den Zugriffsschutz auf Attribute, Methoden und Klassen verwendet.
- Information Hiding ist ein Designaspekt, der sich nicht nur mittels Zugriffsmodifizierern erreichen lässt, sondern auch die interne Darstellung möglichst von der Schnittstelle (öffentliche Methoden) entkoppelt.
- Gutes Information Hiding erlaubt eine möglichst lose Kopplung zwischen den Klassen, gleichzeitig kann man die Implementation in einem gewissen Spielraum jederzeit verändern und anpassen.

Fragen?