PR2 – Formular für Lesenotizen

SS2021

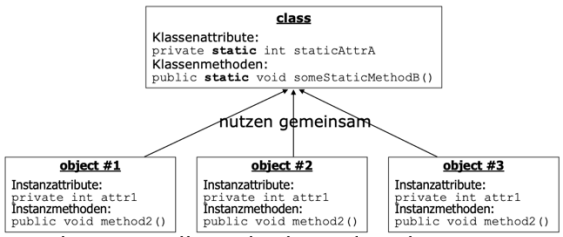
Nachname	Vorname	Matrikelnummer	Abgabedatum:
Lushaj	Detijon	1630149	15.04.21

L.4 static

Wiederverwendbare Methodensammlungen

- Bestimmte statische Methoden werden (unabhängig von Klassen) an vielen Stellen im Programm benötigt

– Wiederverwendung von Code durch Aufruf.



Modul: Ein wiederverwendbares Stück Software in Gestalt einer Klasse.

Syntax: `class.method(parameters);`

L.4.1 Statische Klassenelemente

static: Teil einer Klasse statt Teil eines Objekts.

- Statische Elemente einer Klasse werden nicht in jedes Objekt kopiert, sondern von allen Objekten der Klasse gemeinsam genutzt.

Syntax: `private static type name = value;`

Statisches Attribut / Klassenattribut: In der Klasse (statt im Objekt) gespeichertes Attribut.

- Ein einziges, gemeinsam genutztes, das von allen Objekten der Klasse genutzt und verändert werden kann.

Beispiel: Von einer anderen Klasse *(nur möglich, wenn das statische Attribut nicht private ist):*

```
ClassName.attrName // get the value      ClassName.attrName = value;// set the value
```

Statische Methoden / Klassenmethode:

In der Klasse (statt im Objekt) gespeicherte Methode.

Syntax: `public static type name(parameters) { statements; }`

- Wird gemeinsam von allen Objekten der Klasse genutzt.
- Statische Methoden werden nicht in Subklassen "kopiert".

static und Vererbung

- Statische Methoden einer Superklasse können von einer Subklasse nicht überschrieben werden (im Sinne des dynamischen Bindens).

Zusammenfassung des Begriffs „Java-Klasse"

- Ein Programm: Hat eine `main`-Methode und evtl. weitere statische Methoden.
  - Üblicherweise werden keine statischen Attribute deklariert (außer Klassenkonstanten mit `final`)
- Eine Objekt-Klasse: Definiert einen neuen Objekt-Typ.
  - Deklariert Instanzattribute, Konstruktoren und Instanzmethoden
  - Kann statische Attribute und/oder Methoden deklarieren.
  - Kapselt Daten (i. d. R. alle Instanzattribute und alle statischen Attribute `private`)

L.5 Verschiedenes

L.5.1 Datum und Zeit

Zeitpunkt `Instant` Zeitpunkt in ns `LocalDate` - Datum ohne Zeitzone `LocalTime` - Uhrzeit ohne Datum `LocalDate Time` - Uhrzeit und Datum ohne Zeitzone `ZonedDate Time` - Uhrzeit und Datum mit Zeitzone *2015-03- 03T21:23:45+01: 00 Europe/Berlin*	**Dauer** `Duration` Zeitspanne in s+ns. `Period` Berücksichtigt in Verbindung mit ZonedDateTime auch Sommerzeit *2 Jahre, 3 Monate und 4 Tage*

```
ZoneId zid= ZoneId.of("Europe/Berlin");
ZonedDateTime mauerfall= ZonedDateTime.of(1989, 11, 9, 21, 20, 0, 0, zid);
System.out.println(mauerfall);
```

Die Ausgabe: 1989-11-09T21:20+01:00[Europe/Berlin]

```
ZonedDateTime a= mauerfall.plus(Period.ofDays(140));
```

Die Ausgabe lautet; a: 1990-03-29T21:20+02:00[Europe/Berlin]

```
// Das Ausgangsdatum in Sekunden seit 1.1.1970 0:00 UTC: Instant i=
Instant.ofEpochSecond(mauerfall.getLong(ChronoField.INSTANT_SECONDS));
// Addiere 140 Tage als Duration:
i= i.plus(Duration.ofDays(140));
// Wandle das Ergebnis wieder in ein ZonedDateTime Objekt:
ZonedDateTime b= ZonedDateTime.ofInstant(i, zid);
```

Nun lautet die Ausgabe:b: 1990-03-29T22:20+02:00[Europe/Berlin]

Annotation: Einbindung von Metadaten in den Quelltext.

`@Override` - Die annotierte Methode überschreibt eine Methode aus der Superklasse oder implementiert eine Methode einer Schnittstelle (Marker-Annotation).

`@Deprecated` - Das markierte Element ist veraltet und sollte nicht mehr verwendet werden (Marker-Annotation).

`@SuppressWarnings` - Unterdrückt bestimmte Compiler-Warnungen.

```
@SuppressWarnings("rawtypes") public static void g() { ArrayList list= new ArrayList(); list.add(1); }
```

Javadoc- Tags	Bedeutung
`@author <name>`	- Name des Autors
`@version <id>`	- Versionsbezeichnung
`@param <name> <bedeutung>`	- Dokumentation eines Parameters mit seiner Bedeutung
`@return <bedeutung>`	- Erläuterung des Rückgabewertes
`@see <querverweis>`	- Schafft einen Querverweis auf einen anderen dokumentierten Namen.
`@exception <klasse> <erläuterung>`	- Beschreibung, wann eine Exception der angegebenen Klasse geworfen wird.

JAR – Java Archive

JAR: Dateiformat für die Bündelung mehrerer Java- Klassendateien in einer Datei. *Als Byte Code – zip Datei*

Manifest-Datei: Eine spezielle Datei in einer JAR-Datei, die Meta-Informationen zur JAR-Datei enthält.

L.5.2 Streams (Ein-/Ausgabe)

`throws IOException`

`InputStream`	`Reader`	für die Eingabe von
`OutputStream`	`Writer`	für die Ausgabe von
Bytes	Zeichen	

L.5.2.1 InputStream

```
int i= System.in.read();
System.out.println((char)i);
```

Für genau diesen Zweck, das Einlesen von Zeichen statt Bytes, ist `InputStream` die falsche Klasse.

L.5.2.2 InputStreamReader

```
InputStreamReader r= new InputStreamReader(System.in);
System.out.println(r.getEncoding());
int i= r.read();
System.out.println((char)i);
```

Die Bedeutung der Variable i ist nun kein einzelnes Byte mehr, sondern ein Zeichencode. - Unicode

- `BufferedWriter` - ist ein Writer, der Zeichen puffert, bevor er sie wegschreibt.

L.5.3 Serialisierung von Objektgeflechten

Ziel: "Aktives" Java-Objekt zu "passiver" Folge von Bytes machen und umgekehrt

Serialisierung: Ein Objekt zu serialisieren bedeutet, seine Speicherrepräsentation in eine Folge von Bytes zu verwandeln. Die Bytefolge kann dann z. B. in eine Datei geschrieben werden.

Deserialisierung: Einen Bytestrom zu deserialisieren bedeutet, aus der Bytefolge die Speicherrepräsentation eines Objektes zu rekonstruieren. Das Ergebnis der Deserialisierung ist ein Objekt im Speicher.

Vorteile – Einfach. Man schreibt so gut wie keinen eigenen Code.

Nachteile – Wenige Eingriffsmöglichkeiten. • Alternative: Externalizable

– Nicht wirklich menschenlesbares Format. • Alternative: XML, JSON, ...

Probleme bei jeder Abbildung Bytefolge ↔Objekt: – Versionierung von Bytefolge und Klasse

Wenn sich der Quelltext und damit der Bytecode einer Klasse zwischen Serialisierung und späterer Deserialisierung ändert, kann es zu Problemen kommen

```
public static void speichern(Object o, String filename) throws IOException {
    // OutputStream für Datei erzeugen
    FileOutputStream file = new FileOutputStream(filename); //("personen.dat");
    // ObjectOutputstream, der Objekte serialisiert, erzeugen
    ObjectOutputStream out = new ObjectOutputStream(file);
    // Serialisieren der Objekte
    out.writeObject(o);              //oos.writeObject(gerd);  oos.writeObject(maria);
    out.close();
}
public static Object laden(String filename) throws IOException, ClassNotFoundException {
    // InputStream für Serialisierungs-Datei erzeugen
    FileInputStream f = new FileInputStream(filename);
    // ObjectInputstream, der Objekte deserialisiert, erzeugen
    ObjectInputStream input = new ObjectInputStream(f);
    // Objekte müssen in selber Reihenfolge deserialisiert werden
    Object o= input.readObject();    //gerd = (Person) ois.readObject(); maria = (Person) ois.readObject();
    input.close();
    return o;
}
```

```
public class Person implements Serializable {
    private String name;        //muss imp. Warden!!!
    private int alter;
    private transient int gehalt; // nicht speichern
    private static final long serialVersionUID = 1;
    […] // fuer die Versionenunterscheidung!!!
}
```