Informatik B - Objektorientierte Programmierung in Java

Vorlesung 15:

Serialisierung, Persistenz, Versionierung

Serialisierung

- Inhalt
  - Begriffe
  - Serialisieren
  - Serialisierbare Klassen
  - Deserialisieren
  - Objektreferenzen und Serialisierung
  - Versionierung
  - Beispiele

Begriffe

- Serialisierung
  - Fähigkeit zu Serialisieren, d.h. ein Objekt, das im Hauptspeicher einer Anwendung existiert, in ein Format zu konvertieren, das es erlaubt, das Objekt in eine Datei zu schreiben oder über eine Netzwerkverbindung zu transportieren.
  - Fähigkeit zu Deserialisieren, d.h. ein in serialisierter Form vorliegendes Objekt zu rekonstruieren.

Begriffe

- Persistenz
  - Dauerhafte Speicherung von Daten auf einem Datenträger, so dass sie auch nach Beenden des Programms erhalten bleiben
- Serialisierung und Persistenz sind also nicht gleichzusetzen.
- Die persistente Speicherung von Objekten ist jedoch eine der Hauptanwendungen der Serialisierung.

Serialisieren

- Im Paket `java.io` gibt es die Klasse `ObjectOutputStream`, die es ermöglicht, Objekte zu serialisieren.
- Die Klasse `ObjectOutputStream` besitzt Operationen, um alle primitiven Datentypen zu serialisieren, z.B.

```
public void writeInt(int data)
throws IOException
...
```


Serialisieren

- Wichtigste Operation der Klasse `ObjectOutputStream` ist jedoch

```
public void writeObject(Object o)
throws IOException
```

mit der ein Objekt vollständig serialisiert werden kann.

Serialisieren

- Die Operation `writeObject` schreibt folgende Daten in den `OutputStream`
  - Die Signatur der Klasse des übergebenen Objekts
  - Alle nicht-statischen, nicht-transienten Attribute des übergebenen Objekts inklusive der Attribute, die aus allen Oberklassen geerbt wurden.
- Statische Attribute (Klassenvariablen) werden nicht serialisiert, denn sie gehören nicht zum Objekt sondern zur Klasse des Objekts.
- Wichtig ist weiterhin, dass ein Objekt nur dann mit `writeObject` serialisiert werden kann, wenn es das Interface `Serializable` implementiert.

Serialisieren

- Transiente Attribute
  - Attribute, die nicht serialisiert werden sollen, werden mit dem Modifikator `transient` versehen:

```
class MeineObjektKlasse {
   transient int temporär;
   ...
}
```

  - Typische Beispiele sind solche Attribute, deren Wert sich während des Programmablaufs dynamisch ergibt, oder solche, die nur der temporären Kommunikation zwischen zwei oder mehr Operationen dienen.

Serialisierbare Klassen

- Klassen, die serialisiert werden sollen, müssen das Interface `Serializable` implementieren:

```
public interface Serializable {
// hier stehen keine Methoden
}
```

- `Serializable` ist ein Marker-Interface:
  - Markiert die Klasse als serialisierbar
  - Zur Laufzeit untersucht das System, ob ein Objekt serialisiert werden muss.

Serialisierbare Klassen

- Das Interface definiert zwar keine Methoden, `writeObject` testet jedoch, ob das Objekt `Serializable` implementiert. Sonst: `NotSerializableException`
- Serialisierbarkeit wird mit vererbt.
- Die Standardklassen von Java sind alle serialisierbar, sofern dies ungefährlich ist:
  - Offene Dateien oder Netzwerk-Verbindungen werden nicht serialisiert!
  - Benutzt man selbst ähnliche Attribute, ist es sinnvoll sie als `transient` zu markieren!

Beispiel: `Time.java`

```
/* Time.java */
import java.io.*;
public class Time implements Serializable
{
  private int hour;
  private int minute;

  public Time(int hour, int minute)
  {
    this.hour = hour;
    this.minute = minute;
  }
  public String toString()
  {
    return hour + ":" + minute;
  }
}
```


Beispiel: `WriteObjects.java`

```
import java.io.*;

public class WriteObjects
{
  public static void main(String[] args)
  {
    try {
      FileOutputStream fs = new
                          FileOutputStream("test2.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeInt(123);
      os.writeObject("Hallo");
      os.writeObject(new Time(10, 30));
      os.writeObject(new Time(11, 25));
      os.close();
    } catch (IOException e) {
      System.err.println(e.toString());
    }
  }
}
```

Deserialisieren

- Zur Rekonstruktion von serialisierten Daten steht die Klasse `ObjectInputStream` aus dem Paket `java.io` zur Verfügung.
- Analog zu `ObjectOutputStream` gibt es Operationen zum Deserialisieren von primitiven Datentypen, z.B.

```
public int readInt()
throws IOException
...
```


Deserialisieren

- Wichtigste Operation ist jedoch

```
public Object readObject()
 throws
 ClassNotFoundException, IOException
```

mit der ein serialisiertes Objekt wieder hergestellt werden kann.

Deserialisieren

- Das Deserialisieren eines Objektes besteht stark vereinfacht aus den folgenden beiden Schritten:
  - Zunächst wird ein neues Objekt des zu deserialisierenden Typs angelegt, und die Attribute mit Standardwerten vorbelegt. Zudem wird der Default-Konstruktor der ersten nicht-serialisierbaren Oberklasse aufgerufen.
  - Anschließend werden die serialisierten Daten gelesen und den entsprechenden Attributen des angelegten Objekts zugewiesen.

Deserialisieren

- Nach dem Deserialisieren hat das erzeugte Objekt dieselbe Struktur und denselben Zustand, den das serialisierte Original hatte (mit Ausnahme der statischen und transienten Attribute).
- Beim Deserialisieren werden transiente Attribute lediglich mit einem typspezifischen Standardwert belegt.
- Da der Rückgabewert von `readObject` vom Typ `Object` ist, muss die Referenz auf das erzeugte Objekt in den tatsächlichen Typ (oder eine seiner Oberklassen) umgewandelt werden.

Deserialisieren

- Wurden mehrere Objekte in einen `ObjectOutputStream` serialisiert, müssen diese in der gleichen Reihenfolge wieder deserialisiert werden.
- Beim Deserialisieren von Objekten können einige Fehler passieren.
- Damit ein Aufruf von `readObject` erfolgreich ist, müssen mehrere Kriterien erfüllt sein:
  - Das nächste Element des Eingabestroms ist tatsächlich ein Objekt (kein primitiver Datentyp):
  - Das Objekt muss sich vollständig und fehlerfrei aus dem Eingabestrom lesen lassen.

Deserialisieren

- Es muss eine Konvertierung auf den gewünschten Typ erlauben, also entweder zu derselben oder einer daraus abgeleiteten Klasse gehören.
- Der Bytecode für die Klasse des zu deserialisierenden Objekts muss vorhanden sein. Er wird beim Serialisieren nicht mitgespeichert, sondern muss dem Empfängerprogramm wie üblich als kompilierter Bytecode zur Verfügung stehen.
- Die Klasseninformation des serialisierten Objekts und die im deserialierenden Programm als Bytecode vorhandene Klasse müssen zueinander kompatibel sein (siehe Versionierung).

Beispiel: `ReadObjects.java`

```
import java.io.*;
public class ReadObjects
{
  public static void main(String[] args)
  {
    try {
      FileInputStream fs = new FileInputStream("test2.ser");
      ObjectInputStream is = new ObjectInputStream(fs);
      System.out.println("" + is.readInt());
      System.out.println((String)is.readObject());
      Time time = (Time)is.readObject();
      System.out.println(time.toString());
      time = (Time)is.readObject();
      System.out.println(time.toString());
      is.close();
    } catch (ClassNotFoundException e) {
      System.err.println(e.toString());
    } catch (IOException e) {
      System.err.println(e.toString());
}  }  }
```


Deserialisierung

- Es ist wichtig zu verstehen, dass beim Deserialisieren *nicht* der Konstruktor des erzeugten Objekts aufgerufen wird. Bei einer serialisierbaren Klasse, die in ihrer Vererbungshierarchie Superklassen hat, die nicht das Interface **`Serializable`** implementieren, wird der parameterlose Konstruktor der nächsthöheren nicht-serialisierbaren Vaterklasse aufgerufen. Da die aus der nicht-serialisierbaren Vaterklasse geerbten Membervariablen nicht serialisiert werden, soll auf diese Weise sichergestellt werden, dass sie wenigstens sinnvoll initialisiert werden.

Deserialisierung

- Auch eventuell vorhandene Initialisierungen einzelner Membervariablen werden nicht ausgeführt. Wir könnten beispielsweise die **`Time`**-Klasse um eine Membervariable **`seconds`** erweitern:
  `private transient int seconds = 11;`
- Dann wäre zwar bei allen mit **`new`** konstruierten Objekten der Sekundenwert mit 11 vorbelegt. Bei Objekten, die durch Deserialisieren erzeugt wurden, bleibt er aber 0 (das ist der Standardwert eines **`int`**), denn der Initialisierungscode wird in diesem Fall nicht ausgeführt.

Objektreferenzen und Serialisierung

- Die Serialisierung von Objekten ist nicht trivial, da Objekte Attribute enthalten können, die selbst wieder auf Objekte verweisen, z.B. zum Aufbau von Assoziationen.
- Ein Verfahren zur Serialisierung muss also die folgenden Probleme behandeln:
  - Verfolgung von Objektreferenzen
  - Korrekter Umgang mit zyklischen Referenzen
  - Wiederaufbau der Strukturen aus der serialisierten Form

Objektreferenzen und Serialisierung

- Eine wichtige Eigenschaft der Serialisierung von Java ist die automatische Sicherung und Rekonstruktion von Objektreferenzen:
  - Besitzt ein Objekt selbst Zeichenketten, Felder oder andere Objekte als Attribute, so werden diese ebenso wie Attribute primitiver Datentypen serialisiert und deserialisiert.
  - Da eine Objektvariable lediglich einen Verweis auf das im Hauptspeicher allozierte Objekt darstellt, ist es wichtig, dass diese Verweise auch nach dem Serialisieren und Deserialisieren erhalten bleiben.

Objektreferenzen und Serialisierung

  - Insbesondere darf ein Objekt auch dann nur einmal angelegt werden, wenn darauf von mehr als einer Variablen verwiesen wird.
  - Auch nach dem Deserialisieren darf das Objekt nur einmal vorhanden sein und die verschiedenen Objektvariablen müssen auf dieses Objekt zeigen.
- Der `ObjectOutputStream` hält zu diesem Zweck eine Hash-Tabelle, in der alle bereits serialisierten Objekte verzeichnet werden.

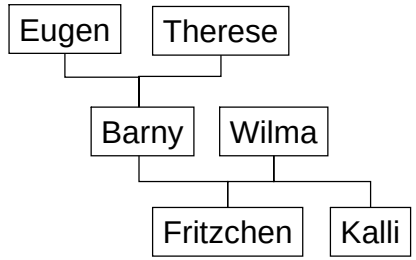
Objektreferenzen und Serialisierung

- Anmerkungen
  - Bei komplexen Objektbeziehungen kann es sein, dass an dem zu serialisierenden Objekt indirekt viele weitere Objekte hängen und beim Serialisieren wesentliche mehr Objekte gespeichert werden, als erwartet wurde.
  - Wenn ein bereits serialisiertes Objekt verändert und anschließend erneut serialisiert wird, bleibt die Veränderung beim Deserialisieren unsichtbar.

Objektreferenzen und Serialisierung

- Durch das Zwischenspeichern der bereits serialisierten Objekte in **`ObjectOutputStream`** werden viele Verweise auf Objekte gehalten, die sonst möglicherweise für das Programm unerreichbar wären. Da der Garbage Collector diese Objekte nicht freigibt, kann es beim Serialisieren einer großen Anzahl von Objekten zu Speicherproblemen kommen. Mit Hilfe der Methode **`reset`** kann der **`ObjectOutputStream`** in den Anfangszustand versetzt werden; alle bereits bekannten Objektreferenzen werden »vergessen«. Wird ein bereits serialisiertes Objekt danach noch einmal gespeichert, wird kein Verweis, sondern das Objekt selbst noch einmal geschrieben.

Beispiel



- Familie.java

```
//Erzeugen der Familie
 Person opa = new Person("Eugen");
 Person oma = new Person("Therese");
 Person vater = new Person("Barny");
 Person mutter = new Person("Wilma");
 Person kind1 = new Person("Fritzchen");
 Person kind2 = new Person("Kalli");
 vater.father = opa;
 vater.mother = oma;
 kind1.father = kind2.father = vater;
 kind1.mother = kind2.mother = mutter;
```


(De-)Serialisieren der Familie

```
//Serialisieren der Familie
 try {
  FileOutputStream fs = new FileOutputStream("test3.ser");
  ObjectOutputStream os = new ObjectOutputStream(fs);
  os.writeObject(kind1);
  os.writeObject(kind2);
  os.close();
  } catch (IOException e) {
   System.err.println(e.toString());
}

//Rekonstruieren der Familie
 kind1 = kind2 = null;
 try {
  FileInputStream fs = new FileInputStream("test3.ser");
  ObjectInputStream is = new ObjectInputStream(fs);
  kind1 = (Person)is.readObject();
  kind2 = (Person)is.readObject();
...
```

Versionierung

- Problem
  - Bei der Serialisierung von Objekten können Konsistenzprobleme auftreten, da eine Datei mit serialisierten Objekten nur die Daten enthält; der zugehörige Code befindet sich dagegen in einer `.class`-Datei.
- Lösung
  - Das Serialisierungs-API versucht, diesem Problem mit einem Versionierungsmechanismus zu begegnen:
    - Beim Serialisieren eines Objektes wird eine Versionsnummer für die zugehörige Klasse erzeugt und mit in die Ausgabedatei gespeichert.
    - In die Versionsnummer gehen Name und Signatur der Klasse, implementierte Interfaces sowie Operationen und Konstruktoren ein.
    - Soll das Objekt später deserialisiert werden, so wird die in der Datei gespeicherte Versionsnummer mit der aktuellen Versionsnummer der Klasse verglichen. Stimmen beide nicht überein, so gibt es eine Ausnahme vom Typ `InvalidClassException` und der Deserialisierungsvorgang bricht ab.

Versionierung

- Beim Aufruf von **`writeObject`** wird automatisch eine Versionsnummer der Klasse – die **`serialVersionUID`** - berechnet und in den Ausgabestrom geschrieben, um beim Deserialisieren mit der ID der Klasse verglichen werden zu können.

Signatur durch `serialVersionUID`

- Signaturen aller Methoden, Konstruktoren, Felder und Interfaces werden überprüft
- Signatur besteht aus Bezeichnern, Modifier, Rückgabetyp, Parametern
- Hieraus wird durch Hash-Verfahren eine (eindeutige) Seriennummer erzeugt
- Änderungen an Kommentaren, Methodenrümpfen ändern die Seriennummer nicht.

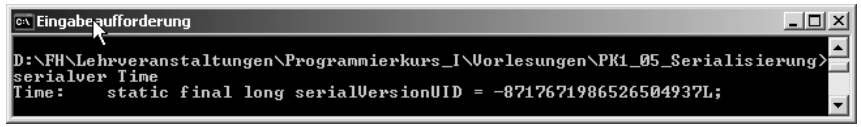
`serialver`

- Die **`serialVersionUID`** einer Klasse kann mit Hilfe des Hilfsprogramms **`serialver`** ermittelt werden. Dieses einfache Programm wird zusammen mit dem Namen der Klasse in der Kommandozeile aufgerufen und liefert die Versionsnummer als Ausgabe. Alternativ kann es auch mit dem Argument **`-show`** aufgerufen werden. Es hat dann eine einfache Oberfläche, in der der Name der Klasse interaktiv eingegeben werden kann.

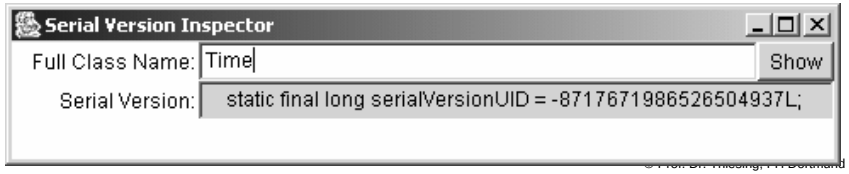
Aufruf von `serialver`

`serialver` wird in der Eingabeaufforderung aufgerufen

- **`serialver Time`**
  `Time:  static final long serialVersionUID = -8717671986526504937L;`



- **`serialver -show`**



Versionierung

- Diese Art der Versionierung ist zwar recht sicher, aber auch sehr rigoros. Schon eine kleine Änderung an der Klasse macht die serialisierten Objekte unbrauchbar, weil sie sich nicht mehr deserialisieren lassen.
- Wird beispielsweise eine neue Methode

```
public void test()
```

hinzugefügt (die für das Deserialisieren eigentlich völlig bedeutungslos ist), ändert sich die **`serialVersionUID`**, und die Datei **`test2.ser`** lässt sich zukünftig nicht deserialisieren.

Versionierung

- Anstatt die Versionsnummer automatisch berechnen zu lassen, kann sie von der zu serialisierenden Klasse auch fest vorgegeben werden.
- Die Verwaltung der Versionsnummer liegt dann in der Verantwortung des Programmierers.
- Werden Änderungen an der Klasse vorgenommen, die zu einer Inkompatibilität führen, muss die Versionsnummer entsprechend geändert werden, und die serialisierten Daten dürfen nicht mehr verwendet werden.

`serialVersionUID` setzen

```
import java.io.*;
public class Time implements Serializable
{
  static final long serialVersionUID = -8717671986526504937L;
  private int hour;
  private int minute;

  public Time(int hour, int minute)
  {
    this.hour = hour;
    this.minute = minute;
  }

  public String toString()
  {
    return hour + ":" + minute;
  }
}
```

`serialVersionUID` setzen

- Jetzt muss die Anwendung natürlich selbst darauf achten, dass die durchgeführten Änderungen kompatibel sind, dass also durch das Laden der Daten aus dem älteren Objekt keine Inkonsistenzen verursacht werden. Dabei mögen folgende Regeln als Anhaltspunkte dienen:
  - Das Hinzufügen oder Entfernen von Methoden ist meist unkritisch.
  - Das Entfernen von Membervariablen ist meist unkritisch.
  - Beim Hinzufügen neuer Membervariablen muss beachtet werden, dass diese nach dem Deserialisieren uninitialisiert sind.
  - Problematisch ist es meist, Membervariablen umzubenennen, sie auf **`transient`** oder **`static`** (oder umgekehrt) zu ändern, die Klasse auf **`Serializable`** (oder umgekehrt) zu ändern oder den Klassennamen zu ändern.