# Informatik B

## Vorlesung 14

## Serialisierung, Autoboxing

# Serialisierung von Objekten

- Die Objekte innerhalb eines Java-Programmes sollen manchmal auch nach Beendigung der JVM verfügbar bleiben
- Objekte müssen ab und an im Netzwerk übertragen werden
- Dazu ist ein Mechanismus notwendig, der ein Objekt abspeichern, bzw. in einen Bytestrom umwandeln und später wieder rekonstruieren kann (Serialisierung / Deserialisierung)
- Dabei sollen alle mit dem Objekt verbundenen Variablen gespeichert / umgewandelt und später wieder rekonstruiert werden
- Hierbei sind alle Instanzvariablen von Interesse, also auch in Oberklassen definierte Variablen

# Serialisierung von Objekten

- Die Instanzvariablen eines Objektes können weitere Objekte referenzieren
- Auch diese referenzierten Objekte müssen mit gespeichert / umgewandelt werden können
- Die Speicherung, bzw. Umwandlung in einen Bytestrom eines Objektes nennt man Serialisierung
- Ein serialisiertes Objekt kann dann dauerhaft gespeichert werden
- Ist ein Objekt dauerhaft gespeichert nennt man dies persistente Speicherung

# Standard-Serialisierung

- Die Standard-Serialisierung ist die einfachste Möglichkeit, Objekte *persistent* zu machen
- Das Objekt wird in einen Bytestrom geschrieben
- Dazu dient die Klasse **`ObjectOutputStream`** und die Methode **`writeObject()`**
- Der **`ObjectOutputStream`** läuft die Zustände und Objektverweise rekursiv ab und schreibt diese in einen **`OutputStream`**
- Der **`OutputStream`** wird bei Erzeugung eines **`ObjectOutputStream`**-Objektes als Parameter übergeben
- Werden die Informationen in eine Datei geschrieben, sollte diese mit der Endung **`.ser`** versehen werden
- Beispiel: **`serialisierung1`**

# Laden serialisierter Objekte

- Liegen Objekte in persistenter Form vor, kann man diese auch wieder einlesen
- Dazu dient die Klasse **`ObjectInputStream`**, die mit einem **`InputStream`** (z.B. **`FileInputStream`**) die Daten einliest
- Die Klasse **`ObjectInputStream`** bietet mit der Methode **`readObject()`** die Möglichkeit, die Objekte auszulesen
- **`readObject()`** ermittelt den Typ des serialisierten Objekts und baut daraus das Zielobjekt auf
- Aus den Daten im Datenstrom werden dann die Zustände des Objekts wiederhergestellt
- Wenn nötig, rekonstruiert der **`ObjectInputStream`** auch Objekte, auf die verwiesen wurde
- Beispiel: **`serialisierung2`**

# `Serializable`

- Es können nur Objekte serialisiert werden, die das Interface **`Serializable`** implementieren
- In dem Interface sind keine Methoden definiert, es handelt sich also um ein Markerinterface
- Sehr viele Klassen der Java API implementieren das Interface, einige Klassen können jedoch nicht serialisiert werden
- Wie sollte z.B. ein **`BufferedWriter`** serialisiert werden, da er ja mit einer Datei (mit einem Stream) verbunden ist?

# `Serializable`

- Um eine eigene Klasse serialisierbar zu machen ist es also notwendig, das Markerinterface **`Serializable`** zu implementieren
- Dabei ist zu überlegen, ob Objekte mit sensiblen Daten serialisierbar sein sollen, denn bei der Serialisierung werden auch private Attribute serialisiert
- Durch die Serialisierung liegen diese Daten z.B. auf der Festplatte und es lassen sich die internen Belegungen ablesen und auch manipulieren

# `Serializable`

- Implementiert eine Klasse das Markerinterface, so ist sie automatisch in der Lage alle Instanzvariablen zu serialisieren
- Klassenvariablen werden nicht serialisiert, das wäre unsinnig
- Beispiel: **`serialisierung3`**
- Falls in der Klasse nicht serialisierbare Objekte referenziert werden würden, wird eine **`NotSerializableException`** geworfen
- Beispiel: **`serialisierung4`**

# `transient`

- Einzelne Instanzvariablen können von der Serialisierung ausgenommen werden
- Dies kann folgende Gründe haben:
  - Ein Objekt mit sensiblen Daten (z.B. Passwörter) sollte nicht serialisiert werden, da es dann manipulierbar wäre
  - Es lassen sich nicht alle Zustände wiederherstellen, z.B. bei den Streams
  - Eine Instanzvariable verweist auf ein nicht serialisierbares Objekt

# `transient`

- Um einzelne Instanzvariablen von der Serialisierung auszunehmen gibt es das Schlüsselwort **`transient`**
- Wird eine Variable mit diesem Schlüsselwort versehen, wird sie bei der Serialisierung nicht berücksichtigt
- Wird eine solche Variable wieder eingelesen bekommt sie den Defaultwert zugewiesen
- Beispiel: **`serialisierung5`**

# `serialPersistentFields`

- Eine andere Möglichkeit festzulegen, welche Instanzvariablen serialisiert werden sollen und welche nicht, bietet die Klassenvariable **`serialPersistentFields`**
- Die Variable muss folgendermaßen definiert werden: **`private static final ObjectStreamField[] serialPersistentFields`**
- Die einzelnen Einträge des Arrays sind die Variablen, die serialisiert werden sollen
- Beispiel: **`serialisierung6`**

# Serialisierung und Vererbung

- Sobald eine Klasse das Markerinterface **`Serializable`** implementiert, sind auch alle Unterklassen serialisierbar
- Umgekehrt gilt das natürlich nicht
- Wird eine Objekt serialisiert und die Oberklasse der Objektklasse ist nicht **`Serializable`** kann die Serialisierung nicht fortgeführt werden
- Um dennoch das Objekt bei der Deserialisierung mit sinnvollen Werten zu versehen, wird der *Default Constructor* aufgerufen
- Ist dieser nicht vorhanden, kann das Objekt zwar serialisiert werden, aber nicht wieder deserialisiert
- Beispiel: **`serialisierung7`**

# Versionierung

- Liegen die serialisierten Objekte in persistenter Form vor, sind sie vom eigentlichen Javacode abgekapselt
- Es kann vorkommen, dass sich der Javacode ändert und dann die persistent gespeicherten Objekte nicht mehr zur Implementation passen
- Java hat deshalb einen eingebauten Versionsschutz
- Zu einer Klasse kann eine Versionsnummer erzeugt werden
- Diese Nummer wird durch die Klassensignatur, die Instanzvariablen (beliebige Sichtbarkeit), Methoden (nur nicht **`private`**) und Konstruktoren (nur nicht **`private`**) definiert
- Ändert sich die serialisierbare Klasse in diesen Punkten, können die Objekte nicht mehr deserialisiert werden
- Beispiel: **`serialisierung8`**

# Versionierung

- Diese Art der Versionskontrolle ist ein sinnvoller Mechanismus, um die Kompatibilität zwischen persistenten Objekten und dem Quellcode zu testen
- Allerdings kann es sein, das sich eine Klasse (und somit die Versionsnummer) ändert, aber die persistenten Objekte weiterhin zum Code kompatibel sind
- Daher kann der Programmierer selber eine Versionsnummer angeben, die **`serialVersionUID`**

# `serialVersionUID`

- Um eine eigene Versionsnummer anzugeben, wird innerhalb der Klasse eine Klassenkonstante namens **`serialVersionUID`** angelegt:
  **`static final long serialVersionUID = …;`**
- Dadurch hat der Programmierer Kontrolle, ab wann Klasse und persistente Objekte nicht mehr zueinander passen
- Sun empfiehlt bei **`Serializable`** Klassen generell eine eigene **`serialVersionUID`** anzulegen, da sich die automatische Berechnung auch bei sich nicht ändernden Klassen je nach Compiler und JVM ändern könnte
- Beispiel: **`serialisierung9`**

# Serialisierung mit XML

- Die Serialisierung mit einem **`ObjectOutputStream`** hat zur Folge, dass die Objekte in einer für den Menschen nicht lesbarer Form vorliegen
- Will man eine textuelle Repräsentierung eines Objektes bietet sich die Serialisierung in ein XML-Format an
- Dazu muss lediglich anstelle eines **`ObjectOutputStream`**-Objektes ein Objekt der Klasse **`java.beans.XMLEncoder`** verwendet werden
- Zum Auslesen persistent gespeicherter Objekte dient die Klasse **`java.beans.XMLDecoder`**

# Serialisierung mit XML

- Soll eine Klasse mit einem **`XMLEncoder`** serialisiert werden, muss sie die Struktur einer *JavaBean* aufweisen:
  - Es muss ein *Default Constructor* vorhanden sein
  - Für alle zu serialisierenden Attribute **`x`** müssen get- und set-Methoden vorhanden sein
  - Die Klasse muss **`public`** sein
- Beispiel: **`xmlserialisierung1`**

# *Autoboxing*

- Wollte man in älteren Javaversionen (vor Java5) einen primitiven Datentyp in ein entsprechendes Objekt umwandeln, musste man explizit eine Methode (oder einen Konstruktor) verwenden
- ```
  Integer i = Integer.valueOf(10);
              // new Integer(10);
  int j = i.intValue();
  ```
- Seit Java5 gibt es das so genannte *Autoboxing*, wodurch ein primitiver Datentyp direkt in das entsprechende Objekt umgewandelt werden kann und umgekehrt
- ```
  Integer i = 10;
  int j = i;
  ```

# *Autoboxing*

- Wird ein primitiver Datentyp in ein entsprechendes Objekt umgewandelt nennt man dies *Boxing*
- Wird aus einem Wrapper-Objekt der primitive Datentyp spricht man von *Unboxing*
- Dadurch wird die Arbeit mit primitiven Datentypen vereinfacht
- So können jetzt direkt primitive Datentypen in eine Collection eingefügt werden, ohne das der Programmierer jeweils Wrapper-Objekte erzeugen muss, dies übernimmt nun Java selber
- Das Autoboxing ist lediglich ein Feature der Semantik (der Programmiersprache)
- Wird der Quellcode mit einer *Autoboxing*-Anweisung compiliert, wird im Bytecode weiterhin z.B. die Methode **`intValue`** bzw. **`valueOf`** verwendet

# Probleme mit **`Integer`**-*Autoboxing*

- Die verkürzte Schreibweise kann zu großen Problemen bzgl. der Geschwindigkeit führen
- So ist jetzt folgender Code möglich:
  **`for(Integer i = 0; i<1000000; i++) { … }`**
- Dies bedeutet innerhalb der Schleife findet immer wieder ein Unboxing eines **`Integer`**-Objektes statt und ein Boxing einer **`int`**-Variablen:
  **`for(Integer i = Integer.valueOf(0); i.intValue()<1000000; i=Integer.valueOf(i.intValue()+1)) { … }`**
- Beispiel: **`autoboxing1`**

# Probleme mit **`Integer`**-*Autoboxing*

- Es gibt ein weiteres Problem bezüglich des *AutoBoxing*
- Die Entwickler von Java haben festgestellt, das insbesondere **`Integer`**-Objekte im Wertebereich **`[-128;127]`** besonders oft beim *AutoBoxing* verwendet werden
- Daher wurden die beim Boxing entstehenden **`Integer`**-Objekte dieses Wertebereichs in einem Pool vordefiniert und zur Verfügung gestellt
- Dadurch ist die Umwandlung innerhalb dieses Wertebereichs wesentlich schneller
- Allerdings existiert beim *AutoBoxing* innerhalb dieses Intervalls je Wert ein **`Integer`**-Objekt nur einmal

# Probleme mit **`Integer`**-*Autoboxing*

- Das bedeutet folgendes:
  - **`new Integer(10) == new Integer(10)`**
    - Liefert **`false`**, da neue Objekte erzeugt werden
  - **`Integer i = 4711; Integer j = 4711; i==j`**
    - Liefert **`false`** , da neue Objekte erzeugt werden
  - **`Integer i = 42; Integer j = 42; i==j`**
    - Liefert **`true`**, da die Objekte aus dem Pool geholt werden
  - **`Integer i = 42; i == new Integer(42)`**
    - Liefert **`false`**, da ein Objekt aus dem Pool stammt und eines neu erzeugt wurde
- Beispiel: **`autoboxing2`**

# Zusammenfassung

- Serialisierung von Objekten
  - Serialisieren
  - Persitenz
  - Deserialisieren
  - **`serialVersionUID`**
  - Speicherung als XML
- *AutoBoxing*

# Ausblick

- Reflection
  - Erzeugen von Objekten
  - Methoden aufrufen
  - Manipulation von Objekten