Übung 2: IO und Reflection (40 Punkte)

Abgabe: 23.04.2018, 8:30, als SVN Commit

Der Java-Serialisierungsmechansimus erlaubt beliebige Objektgraphen zu schreiben und wieder zu lesen. Ein Nachteil dieses Verfahrens ist aber, dass das Ausgabeformat relativ aufwendig ist und damit die entstehenden Datenmengen erheblich sein können. Des Weiteren ist das Speicherformat binär und kann vom Menschen nicht direkt gelesen werden. In dieser Übung wollen wir daher einen einfachen (und eingeschränkten) Objektserialisierungsmechanismus implementieren, der eine kompakte Serialisierung in einem Textformat erlaubt.

Der Serialisierungsmechansimus soll durch die Klassen `ObjectWriter` und `ObjectReader` bereitgestellt werden. Die Klassen definieren eine `write` bzw. eine `read` Methode:

```
public class ObjectWriter {
     public void write(Object obj, OutputStream out) throws IOException
}
public class ObjectReader {
     public Object read(InputStream in) throws IOException
}
```

Ein `ObjectWriter` erlaubt somit, ein Objekt auf einen `OutputStream` zu schreiben

```
ObjectWriter ow = new ObjectWriter();
try (OutputStream out = new FileOutputStream("SerizalizedObject.txt")) {
     Object a = …;
     ow.write(a, out);
} catch (Exception e) {
     e.printStackTrace();
}
```

und in analoger Weise eine `ObjectReader` das Objekt aus den geschriebenen Daten wieder zu lesen

```
ObjectReader or = new ObjectReader ();
try (InputStream in = new FileInputStream("SerizalizedObject.txt")) {
     Object a = or.read(in);
     …
} catch (Exception e) {
     e.printStackTrace();
}
```

Anforderungen

Folgende Serialisierungsmechanismen sollen unterstützt werden:

1) *Primitive Datentypen*: Lesen und Schreiben von primitiven Datentypen.

2) *String-Serialisierung*: Serialisierung von Objekttypen, die über einen Konstruktor mit einem String-Parameter verfügen und die eine `toString`-Methode haben, die eine entsprechende String-Repräsentation liefert. Das heißt, solche Objekte kann man einfach serialisieren, indem man den durch `toString` erzeugen String schreibt, und deserialisieren, indem man den String liest und mit diesem den Konstruktor aufruft. Zu den Typen, die derart geschrieben und gelesen werden können zählen z.B. `String`, alle Wrapper-Klassen für die Basisdatentypen und `Date` und `Time` vom Testbeispiel (siehe unten).

3) *Feld-Serialisierung*: Objekttypen, die einen Konstruktor ohne Parameter (Standardkonstruktor) haben, sollen durch Schreiben der Werte der Felder serialisiert werden. Das heißt, bei der Serialisierung wird zuerst der Name der Klasse und dann die Werte aller Felder geschrieben. Beim Deserialisieren wird zuerst der Klassenname gelesen, mit dem Standardkonstruktor der Klasse ein

leeres Objekt erzeugt und dann die Werte der Felder deserialisiert und mit diesen die Felder des Objekts gesetzt.

4) *Custom Serializer*: Als spezielles Feature bei der Feld-Serialisierung soll man bei Felddeklarationen durch eine `UseSerializer` Annotation einen speziellen `FieldSerializer` einstellen können. Das Vorgehen ist unten im Detail beschrieben.

5) *Objektgraphen*: Wurde ein Objekt bereits serialisiert und tritt dieses selbe Objekt als Wert eines weiteren Feldes auf, soll das Objekt kein zweites Mal serialisiert werden. Stattdessen soll eine eindeutige ID für das Objekt existieren und diese ID geschrieben werden. Bei der Deserialiserung soll mit der ID auf das dann bereits deseralisierte Objekt zugegriffen und die Referenz beim Feld gespeichert werden. Mit diesem Vorgehen soll die Serialiserung von Objektgraphen unterstützt werden, die beliebige Querverweise enthalten können.

6) *Exceptions*: Bei der Serialisierung von Objekten können Fälle auftreten, die Sie nicht richtig behandelt werden können. Zum Beispiel könnte eine Objekt eine Feld verwenden, das nicht serialisiert werden kann. Werfen Sie in diesem Fall eine entsprechende `IOException`. Überlegen Sie, welche `IOException` für den Fall am besten passen würde (z.B. gibt es eine Unterklasse `InvalidClassException` von `IOException`).

Custom Serializer:

Die Verwendung von Custom Serializer für bestimmte Felder von Klassen soll wie folgt funktionieren.

Mit der Annotation `UseSerializer`, die folgend definiert ist

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD })
public @interface UseSerializer {
     Class<? extends FieldSerializer> value();
}
```

kann bei einem Feld der zu verwendende Serializer bestimmt werden. Zum Beispiel ist in der folgenden Klasse `Person` bestimmt, dass das Feld `hobbies` durch einen `StringListSerializer` serialisiert werden soll:

```
class Person {
     @UseSerializer(StringListSerializer.class)
     private List<String> hobbies;
…
}
```

Implementierungen von Serializer wie z.B. der obige `StringListSerializer` sind vom Interface `FieldSerializer` abgeleitet, das Methoden `write` und `read` definiert:

```
public interface FieldSerializer {
     public abstract void write(Object v, PrintWriter writer) throws IOException;
     public abstract Object read(BufferedReader reader) throws IOException;
}
```

Die Serialisierung bzw. Deserialisierung erfolgt dann einfach durch Aufruf dieser Methoden. Die Klasse `StringListSerializer` könnte z.B. folgend definiert sein.

```
public class StringListSerializer implements FieldSerializer {

     @Override
     public void write(Object o, PrintWriter writer) throws IOException {
          @SuppressWarnings("unchecked")
          List<String> al = (List<String>) o;
```

```
            String elems = al.stream().collect(Collectors.joining(" "));
            writer.println(elems);
        }

        @Override
        public Object read(BufferedReader reader) throws IOException {
            String line = reader.readLine();
            String[] words = line.split(" ");
            ArrayList<String> al = new ArrayList<String>();
            for (String e : words) {
                al.add(e);
            }
            return al;
        }
    }
```

Hinweise:

Die Aufgabe ist allgemein unter Verwendung von Reflection zu lösen. Die Aufgabe ist aber auf die obigen Mechanismen eingeschränkt und unterstützt damit natürlich nicht alle Java-Datentypen. So werden z.B. Arrays und auch generischen Datentypen nicht allgemein unterstützt (nur eventuell über einen Custom Serializer).

Zum Schreiben und Lesen der Daten verwenden Sie am besten einen `PrintWriter` und einen `BufferedReader`. Schreiben Sie die Daten in einzelne Zeilen (Methode `println`), dann können Sie sie mit `readLine` von `BufferedReader` auch zeilenweise lesen.

Ad Anfoderung 3) Feld-Serialisierung:

- *Ermitteln aller Felder einer Klasse*: Um die Felder eines Objekts über Reflection setzen zu können, müssen sie alle Felder der Klasse ermitteln. Mit der Methode `getFields` erhalten Sie aber nur die public-Felder der Klasse, mit `getDeclaredFields` alle Felder, die in der Klasse selbst deklariert sind, aber nicht jene, die geerbt sind. Mit `getSuperclass` können Sie aber auf die Superklasse zugreifen. Sie müssen sich daher eine Methode schreiben, die alle Felder sammelt.
- *Lesen und Schreiben von privaten Feldern*: Private Felder können Sie normalerweise auch über Reflection nicht lesen und schreiben. Man kann private Felder aber über Reflection lesen und schreiben, wenn man vorher

  ```
  field.setAccessible(true);
  ```

  aufruft.

Ad Anfoderung 5) Objektgraphen:

- Um Serialisierung von Objektgraphen zu unterstützen, müssen Sie mit der ersten Serialisierung eines Objekts eine eindeutige ID mitspeichern. Tritt das selbe Objekt nochmals auf, speichern Sie nur diese ID (am besten zusammen mit einer eindeutigen Kennung). Verwenden Sie bei der Serialisierung eine Map, die jedem serialisierten Objekt eine ID zuordnet.
- Bei der Deserialisierung verwenden Sie diese ID, um das Objekt eindeutig zu identifizieren. Lesen Sie eine ID, müssen Sie mit der ID das bereits deserialisierte Objekt eindeutig identifizieren können. Verwenden Sie daher bei der Deserialisierung eine Map, mit der man von einer ID auf das Objekt zugreifen kann.

Beachten Sie des Weiteren, dass Sie null-Werte speziell behandeln müssen.

Download:

Im Download der Homepage finden Sie eine Klasse `Person` und eine Testklasse, in der eine Struktur von `Person`s aufgebaut wird. Sie sollten Ihre Implementierung mit dieser Struktur testen.

Des Weiteren werden die Definitionen der obigen Klassen wie oben dargestellt bereitgestellt.