Praktikum Grundlagen der Programmierung I (Java)

WS 2009/2010

10. Dezember 2009

Übung 8.1 Binäres Lesen und Schreiben

Zur allgemeinen Struktur der Klasse `Person`: Diese besitzt wie vorgegeben Attribute für Name, Alter und Gewicht, mit entsprechenden Gettern und einem Konstruktor, dem beim Instanzieren eines Objektes die Feldwerte übergeben werden. Weiterhin ist `toString()` sinnvoll überladen.

```
public class Person {
    private String name;
    private int alter;
    private double gewicht;

    public Person(String name, int alter, double gewicht) {
        this.name = name;
        this.alter = alter;
        this.gewicht = gewicht;
    }

    public String getName() {
        return name;
    }

    public int getAlter() {
        return alter;
    }

    public double getGewicht() {
        return gewicht;
    }

    @Override
    public String toString() {
        return String.format("%s, %d Jahre, %f kg", name, alter, gewicht);
    }

}
```

Zum binären Schreiben des Strings mit dem Namen wird folgender Ansatz gewählt: Zunächst wird die Repräsentation des Strings als Byte-Array erzeugt. Diese ist bzgl. des vom System verwendeten Zeichensatzes, diese Binärdarstellung ist also nicht portabel! Da die Länge des Strings variabel ist, reicht es nicht aus, einfach diese Bytes in den binären Ausgabestrom zu schreiben. Entweder man benutzt eine Terminierung des Strings mit Nullbyte (wie in C), oder man schreibt vorher die Zahl der Bytes im String als Zahl in den OutputStream. Da letzteres beim Zurücklesen einfacher ist, wird hier diese Variante gewählt:

```
byte[] bytes = name.getBytes();
out.write(bytes.length);
out.write(bytes);
```

Bemerkung: Bei `out.write(bytes.length)` wird ein Byte geschrieben, obwohl als Parameter ein `int` erwartet wird. Von diesem wird aber nur das low byte geschrieben, die anderen werden ignoriert. (Dies bedeutet hier, dass der Name maximal 255 Zeichen haben darf)

Zum Schreiben des Alters wird der `int` zunächst in Bytes aufgesplittet und in einen Byte-Array der Länge 4 überführt (in Java hat int immer 4 Bytes). Hierzu wird die Methode `rotateRight` benutzt, um das gewünschte Byte an die Position des low byte zu rotieren und dieses dann mit der bitweisen Und-Operation `& 0xFF` „ausgeschnitten“. Dieser Byte-Array wird dann in den OutputStream geschrieben:

```
bytes = new byte[4];
for (int i=0; i<bytes.length; i++) {
    bytes[i] = (byte) (Integer.rotateRight(alter, 8*i) & 0xFF);
}
out.write(bytes);
```

In dieser Implementierung landet das low byte zuerst im Ausgabestrom, die entspicht der Bytereihenfolge Little Endian (wie sie z. B. auf x86 benutzt wird), würde man zuerst das high byte wegschreiben, wäre dies Big Endian (üblich z. B. auf PowerPC und Motorola 68000). Welche Reihenfolge man hier wählt, ist letztlich egal, solange man beim Zurücklesen dann dieselbe Reihenfolge nimmt.

Für das Gewicht wird zunächst die Binärdarstellung des double als long (double und long haben in Java immer 8 Byte) interpretiert (`doubleToLongBits`). Diese wird dann analog zu oben durch byteweises Rotieren in einen Byte-Array überführt (hier natürlich der Länge 8), dieser dann weggeschrieben:

```
bytes = new byte[8];
long temp = Double.doubleToLongBits(gewicht);
for (int i=0; i<bytes.length; i++) {
    bytes[i] = (byte) (Long.rotateRight(temp, 8*i) & 0xFF);
}
out.write(bytes);
```

Beispiel:

```
Person p1 = new Person("Jörg Müller",35,79.75);
OutputStream out = new FileOutputStream("personen.dat");
p1.toStream(out);
out.close();
```

ergibt als Binärdatei

- 1. Byte: 0x0B als Länge des String „Jörg Müller“ in Byte (11 bei System mit z.B. ISO-8859-1)
- 2.-12. Byte: Bytes mit dem Namen
- 13.-16. Byte: 0x23 0x00 0x00 0x00 mit dem Alter in 4 Bytes, Little Endian
- 17.-24. Byte: 0x00 0x00 0x00 0x00 0x00 0xF0 0x53 0x40 mit dem Gewicht als IEEE 64Bit-Float, Little Endian

Zum Zurücklesen muss exakt die gleiche Reihenfolge der Attribute und die selbe Bytereihenfolge eingehalten werden.

Zunächst muss also ein Byte gelesen werden, das angibt, wie groß der Byte-Array für den Namen dimensioniert werden muss. Anschließend werden entsprechend die Bytes in diesen Array gelesen und das ganze als String interpretiert (wiederum bzgl. des Systemzeichensatzes, dieser muss also mit dem Systemzeichensatz des Rechners, der zum Schreiben benutzt wurde, identisch sein):

```
int length = in.read();
byte[] bytes = new byte[length];
in.read(bytes);
String name = new String(bytes);
```

Für das Alter werden 4 Bytes gelesen und diese dann benutzt, um mit Rotieren und bitweisem Oder dann einen „leeren“ int zu füllen:

```
int alter = 0;
bytes = new byte[4];
in.read(bytes);
for (int i=0; i<bytes.length; i++) {
    alter |= Integer.rotateLeft(bytes[i] & 0xFF, 8*i);
}
```

Der zusammengesetzte Operator `a |= b` entspricht dabei `a = a | b`.

Beim Gewicht geht es ganz Analog, nur dass nun 8 Bytes gelesen, als long zusammengesetzt und dann binär als double interpretiert werden:

```
long temp = 0;
bytes = new byte[8];
in.read(bytes);
for (int i=0; i<bytes.length; i++) {
    temp |= Long.rotateLeft(bytes[i] & 0xFF, 8*i);
}
double gewicht = Double.longBitsToDouble(temp);
```

Nachdem diese Werte gelesen wurden, kann nun mit `new Person(name,alter,gewicht)` eine neue Instanz von Person angelegt werden.

Übung 8.2 Verwendung von DataOutputStream bzw. DataInputStream

Das Konvertieren von Datentypen in Bytes „von Hand“ ist mühsam, Java bietet hierfür High-Level-Klassen, die direkt Methoden zum Schreiben bzw. Lesen der Datentypen zur Verfügung stellen.

Hier wird das Schreiben ganz einfach:

```
DataOutputStream dataout = new DataOutputStream(out);
dataout.writeUTF(name);
dataout.writeInt(alter);
dataout.writeDouble(gewicht);
```

Bemerkenswert ist, dass Java als Byte-Reihenfolge Big Endian wählt, unabhängig von der Bytereihenfolge des benutzten Systems. Mit `writeUTF` wird ein String im Format UTF8 geschrieben, unabhängig vom Systemzeichensatz. Eine so erzeugte Binärdatei ist also portabel!

Mit denselben Daten (Jörg Müller, 35 Jahre, 79.75kg) ergibt sich folgende Binärdatei:

- 1.–2. Byte: 0x00 0x0D mit der Länge des Strings, dargestellt in 2 Byte big endian, hier also 13
- 3.–15. Byte: 13 Bytes mit dem Namen in UTF8 (da Umlaute in UTF8 zwei Bytes benötigen, ist die Byte-Darstellung des Namens also länger als vorher!)
- 16.–19. Byte: 0x00 0x00 0x00 0x23 mit dem Alter in 4 Bytes, big endian
- 20.–27. Byte: 0x40 0x53 0xF0 0x00 0x00 0x00 0x00 0x00 mit dem Gewicht als IEEE 64bit-Float, big endian

Man beachte, dass die Bytewerte für Alter und Gewicht zu Aufgabe 1 identisch sind, nur in genau umgekehrter Reihenfolge.

Das Lesen ist genauso einfach:

```
DataInputStream datain = new DataInputStream(in);
String name = datain.readUTF();
int alter = datain.readInt();
double gewicht = datain.readDouble();
```

Man kann noch bemerken, dass es statt `writeUTF(String s)` und `readUTF()` noch die Methoden `writeChars(String s)` und `readChars()` gibt. Bei diesen werden für jedes Zeichen zwei Bytes geschrieben bzw. gelesen. Hier wird also mehr Platz verbraucht (bei „Jörg Müller“ 22 Bytes statt 15 in UTF8), es gibt aber eine feste Beziehung zwischen Zahl der Zeichen und Zahl der Bytes (nämlich einfach Faktor 2), während die Länge der eines UTF8-kodierten Strings in Bytes vom Inhalt abhängt, da die Zahl der Bytes, die in UTF8 für die Darstellung eines Zeichens benötigt werden, vom Zeichen abhängen.