Arbeitsblatt: Cloning Varianten

In diesem Arbeitsblatt werden für die Implementierung der `clone()` Methode zusätzlich zu der im Kurs diskutierten Variante mit dem Aufruf von `Object.clone()` vier weitere Varianten vorgestellt.

In allen Beispielen betrachten wir folgende Klasse `Dictionary`. Diese soll mit verschiedenen Verfahren tief geklont werden (*deep copy*).

```
public class Dictionary implements Cloneable {
    private String language;
    private final int size;
    private String[] words;

    public Dictionary(String language, int size) {
        this.language = language;
        this.size = size;
        this.words = new String[size];
        for (int i = 0; i < size; i++)
             this.words[i] = "sample word " + i;
    }

    @Override
    public Dictionary clone() {
        try {
             Dictionary d = (Dictionary) super.clone();
             if (words != null) {
                 d.words = words.clone();
             }
             return d;
        } catch (CloneNotSupportedException e) {
             throw new InternalError();
        }
    }
}
```

(A) Cloning mit Copy-Konstruktor

Beim Cloning mit einem Copy-Konstruktor wird in jeder Klasse, die geklont werden soll, ein Konstruktor deklariert, der als Parameter eine Instanz der eigenen Klasse hat:

```
class C {
    public C(C c) {
        if(c == null) throw new IllegalArgumentException();
        ...
    }
}
```

Die neue Instanz wird dabei mit den Feldern der übergebenen Instanz initialisiert. Bei Referenzvariablen muss individuell entschieden werden, ob eine *deep* oder eine *shallow* Kopie das richtige ist. Falls die Klasse eine Erweiterung einer anderen Klasse ist, wird als erstes der Copy-Konstruktor der Basisklasse aufgerufen:

```
class D extends C {
  public D(D d){ super(d); … }
}
```

Wenn in einer Klasse ein Copy-Konstruktor vorhanden ist, dann kann die `clone` Methode (z.B. für die Klasse C) wie folgt implementiert werden:

```
  public Object clone() { return new C(this); }
```

Eine solche `clone`-Methode muss in jeder Klasse implementiert werden, da die von der Basisklasse geerbte Methode Objekte des falschen Typs zurückgibt.

Aufgaben:

1. Implementieren Sie für die Klasse `Dictionary` die Methode `clone`, die mit Copy-Konstruktoren arbeitet.

2. Messen Sie die Effizienz dieser Methode im Vergleich zum eingebauten Java Cloning.

3. Können mit dieser Methode auch finale Felder kopiert werden? (In der Klasse Dictionary ist z.B. das Feld `size` als final deklariert).

4. Wie ist ein Deep-Copy zu realisieren?

5. Wie verhält sich Clone mit Copy-Konstruktoren bei
   a) zyklischen Strukturen?
   b) alias Referenzen?

   Sie finden im Paket `patterns.clone.alias` zwei kleine Klassen, in welchen Datenstrukturen mit einem Zyklus bzw. mit einer Rauten-Struktur erzeugt werden. Kopieren Sie diese Datenstrukturen mit Copy-Konstruktor-Cloning und prüfen Sie, ob die Struktur erhalten bleibt.

6. Soll eine Klasse, welche Clone mit Copy-Konstruktoren impelmentiert, auch das `Cloneable` Interface implementieren?

(B) Reflective Clone

Bei der reflektiven Variante wird die Kopie mit Java Reflection erstellt. Die `clone`-Methode in der Klasse `C` sieht dann wie folgt aus:

```
public Object clone () {
   return ReflectiveClone.clone(this);
}
```

Die Implementierung der Klasse `ReflectiveClone` finden Sie im Projekt zu diesem Arbeitsblatt.

Die `clone`-Methode in dieser Klasse verwendet Funktionen aus dem Java Reflection Paket, um eine Kopie des gegebenen Objektes herzustellen. Das Java Reflection Paket (`java.lang.reflect.*`) enthält Methoden, um Objekte zur Laufzeit zu untersuchen. Schlüssel dazu ist die Klasse `Class`. Unten sind ein paar Beispiele angegeben.

- Zugriff auf alle Attribute eines Objektes x:

```
Class objClass = x.getClass();
Field[] fields = objClass.getDeclaredFields();
```

- Abfragen des Wertes eines Feldes einer Instanz x:

```
Field f = fields[0];
Object value = f.get(x)
```

Weitere Informationen zu Reflection finden Sie unter http://docs.oracle.com/javase/tutorial/reflect/

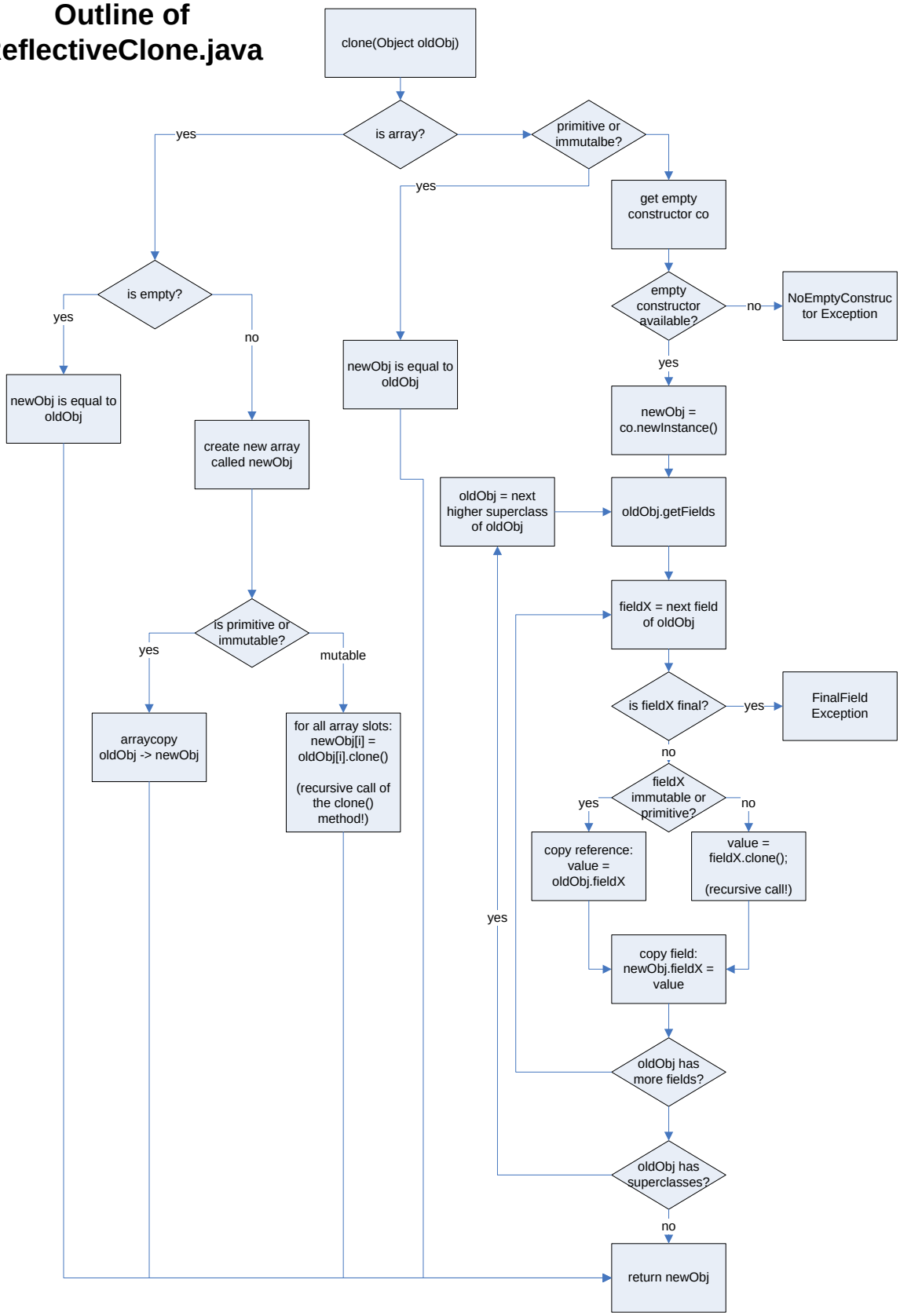
Reflection kann verwendet werden, um generisch Objekte zu kopieren. Reflection erlaubt es auf einen Konstruktor zuzugreifen und über diesen eine neue Instanz zu erzeugen. Dann können auf dieser neuen Instanz alle Felder gesetzt werden. Dies funktioniert auch für geschützte und private Felder. Felder mit einem primitiven Typ (`double`, `int`, …) oder Felder die auf ein unveränderbares Objekt zeigen (wie `String`) werden kopiert, bei den anderen Objekten wird rekursiv geklont. Diese Idee ist einfach, aber damit es richtig läuft, müssen ein paar Details beachtet werden. Eine Implementation finden Sie in der Klasse `RecursiveClone`. Das Ablaufdiagramm auf der Rückseite zeigt, wie diese Variante arbeitet.

Aufgaben:

1. Studieren Sie die Klasse `ReflectiveClone.java` und versuchen Sie zu verstehen wie die `clone`-Methode arbeitet. Sie müssen nicht jedes Detail verstehen. Es genügt, wenn sie den Ablauf grob skizzieren können. Das Ablaufdiagramm auf der Rückseite kann helfen den Code zu verstehen.
   - Wird per Default ein shallow oder ein deep copy gemacht?
   - Warum gibt es im Code der Klasse `ReflectiveClone` eine Unterscheidung zwischen veränderbaren und unveränderbaren Objekten sowie primitiven Datentypen?
2. Vergleichen Sie die Geschwindigkeit dieser Variante mit der Default `Object.clone` Variante
3. Können auch finale Felder so kopiert werden? (In der Klasse `Dictionary` ist z.B. das Feld `size` als final deklariert).
4. Was passiert wenn Klassen keinen Default-Konstruktor enthalten; z.B. wenn Sie in Ihrer Klasse ein Feld vom Typ `UUID` definiert haben (die Klasse `UUID` hat keinen Default-Konstruktor)?
5. Wie verhält sich `ReflectiveClone` bei
   a) zyklischen Strukturen?
   b) alias Referenzen?

   Sie finden im Paket `patterns.clone.alias` zwei kleine Klassen, in welchen Datenstrukturen mit einem Zyklus bzw. mit einer Rauten-Struktur erzeugt werden. Kopieren Sie diese Datenstrukturen mit Reflective-Cloning und prüfen Sie, ob die Struktur erhalten bleibt.
6. Kann `ReflectiveClone` mit inneren (nicht statischen) Elementklassen umgehen?
7. Soll eine Klasse, welche Clone mit `ReflectiveClone` implementiert, auch das `Cloneable` Interface implementieren?

Outline of ReflectiveClone.java



(C) Implementation von Clone via Java Serialization

Java Serialization erlaubt es, ein Objekt (samt referenzierten Objekten) in einen Byte-Strom umzuwandeln, und umgekehrt, einen serialisierten Objektgraphen wieder in Java-Objekte zu konvertieren.

Diese Technik kann verwendet werden, um eine Kopie eines Objektes zu erzeugen. Dazu wird der Objektgraph über einen Stream in einen Byte-Array im Hauptspeicher geschrieben und danach wieder daraus gelesen. Die `clone`-Methode in einer Klasse sieht dann wie folgt aus:

```
public Object clone () {
   try {
      ByteArrayOutputStream out = new ByteArrayOutputStream();
      ObjectOutputStream oout = new ObjectOutputStream(out);
      oout.writeObject(this);

      ObjectInputStream oin = new ObjectinputStream(
         new ByteArrayInputStream(out.toByteArray()));

      return oin.readObject();
   }
   catch (Exception e)  {
      throw new RuntimeException ("cannot clone class");
   }
}
```

Voraussetzung ist, dass die Klasse die geklont wird (und alle referenzierten Klassen), als serialisierbar deklariert ist (`implements java.io.Serializable`).

Bemerkung:
Java Serialisierung ist bequem. Klassen können serialisierbar gemacht werden, indem das Marker-Interface `Serializable` implementiert wird. Falls eine Klasse von einer nicht als serialisierbar deklarierten Klasse abgeleitet und als serialisierbar deklariert wird, so muss die Basisklasse einen Default-Konstruktor aufweisen.

Aufgaben:

1. Vergleichen Sie die Geschwindigkeit dieser Variante mit der Default `Object.clone` Variante

2. Wird beim Deserialisieren eines Objektes der Default-Konstruktor aufgerufen? Und wie verhält sich diese Variante wenn kein Defaultkonstruktor vorhanden ist oder wenn dieser privat deklariert ist?

3. Können auch finale Felder so kopiert werden? (In der Klasse `Dictionary` ist z.B. das Feld `size` als final deklariert).

4. Werden referenzierte Strings kopiert?

5. Wie verhält sich Clone via Serialisierung bei
   a) zyklischen Strukturen?
   b) alias Referenzen?

   Sie finden im Paket `patterns.clone.alias` zwei kleine Klassen, in welchen Datenstrukturen mit einem Zyklus bzw. mit einer Rauten-Struktur erzeugt werden. Kopieren Sie diese Datenstrukturen mit Serialization-Cloning und prüfen Sie, ob die Struktur erhalten bleibt.

6. Soll eine Klasse, welche Clone mit Serialisierung implementiert, auch das `Cloneable` Interface implementieren?

(D) Cloner Clone

Unter https://github.com/kostaskougios/cloning wurde eine kleine Java-Bibliothek publiziert mit der Objektstrukturen tief kopiert werden können. Diese Bibliothek ist im Projekt zu diesem Arbeitsblatt bereits enthalten (und die Sourcen sind ebenfalls bereits verlinkt).

Die Kopie einer Datenstruktur kann mit der Methode `deepClone` der Klasse `Cloner` erstellt werden. Eine Instanz der Klasse `Cloner` kann über die Methode `Cloner.standard()` erzeugt werden.

Die cloning-Bibliothek verwendet Java Reflection um Kopien zu erzeugen. Das Java Reflection Paket (`java.lang.reflect.*`) enthält Methoden um Objekte zur Laufzeit zu untersuchen. Schlüssel dazu ist die Klasse `Class`. Im Folgenden sind ein paar Beispiele angegeben.

- Zugriff auf alle Attribute eines Objektes x:

```
Class objClass = x.getClass();
Field[] fields = objClass.getDeclaredFields();
```

- Abfragen des Wertes eines Feldes einer Instanz x:

```
Field f = fields[0];
Object value = f.get(x)
```

Reflection erlaubt es auf einen Konstruktor zuzugreifen und über diesen eine neue Instanz zu erzeugen. Dann können alle Klassen in der Vererbungshierarchie betrachtet werden und die dort deklarierten Objekte können gesetzt werden. Dies funktioniert auch für geschützte und private Felder. Felder mit einem primitiven Typ (`double`, `int`, …) oder Felder die auf ein unveränderbares Objekt zeigen (wie `String`) werden kopiert, bei den anderen Objekten wird rekursiv geklont. Diese Idee ist einfach, aber damit es richtig läuft müssen ein paar Details beachtet werden.

Die Implementierung einer clone-Methode mit Hilfe der cloning-Bibliothek sieht dann wie folgt aus:

```
public Object clone () {
    return Cloner.standard().deepClone(this);
}
```

Weitere Informationen zu Reflection finden Sie unter http://docs.oracle.com/javase/tutorial/reflect/

Aufgaben:

1. Vergleichen Sie die Geschwindigkeit dieser Variante mit der Default `Object.clone` Variante.

2. Können auch finale Felder mit dieser Bibliothek kopiert werden? (in der Klasse `Dictionary` ist z.B. das Feld `size` als final deklariert).

3. Was passiert wenn Klassen keinen Default-Konstruktor enthalten, z.B. wenn Sie in Ihrer Klasse ein Feld vom Typ `UUID` definiert haben (die Klasse `UUID` hat keinen Default-Konstruktor)?

4. Wie verhält sich `deepClone` bei
   a) zyklischen Strukturen?
   b) alias Referenzen?

   Sie finden im Paket `patterns.clone.alias` zwei kleine Klassen, in welchen Datenstrukturen mit einem Zyklus bzw. mit einer Rauten-Struktur erzeugt werden. Kopieren Sie diese Datenstrukturen mit `deepClone` und prüfen Sie, ob die Struktur erhalten bleibt.

5. Kann `deepClone` mit inneren (nicht statischen) Elementklassen umgehen?

6. Versuchen Sie zu verstehen, wie die Methode `deepClone` arbeitet. Neben der Methode deepClone bietet die Klasse `Cloner` auch noch die Methode `shallowClone` an. Wie wird unterschieden ob ein `deepClone` oder ein `shallowClone` gemacht wird?

7. Soll eine Klasse, welche Clone mit `deepClone` implementiert, auch das `Cloneable` Interface implementieren?