Fachhochschule
Münster University of
Applied Sciences

Java Idioms

Basic und Advanced *Java Coding Style.*

Prof. Dr. Nikolaus Wulff

Java Idiome

- Operator == versus equals Methode
- equals und hashCode
- Vermeide NullPointer
- Java Konstruktoren
- Function Pointers, Interfaces und Reflection
- ...

== Operator und equals

- In Java sind alle Bezeichner Referenzen auf Objekte, d.h. verhalten sich wie C Pointer.
- Wird auf zwei Objekte der == Operator angewandt, so wird nicht auf logische Gleichheit verglichen, sondern lediglich ob beide Referenzen auf die selber Speicheradresse verweisen.
- Sollen zwei Objekte auf logische Gleichheit überprüft werden, muss die equals Methode verwendet werden.
- Diese wird von der Superklasse java.lang.Object geerbt und muss geeignet überladen sein.

equals überladen

- Die equals Implementierung der Klasse Object vergleicht lediglich ob die Referenzen gleich sind, d.h. entspricht dem == Operator.
- Die Impementierung von equals muss etwas mit dem (inneren) Status der Instanz zu tun haben.
- equals entspricht einer Äquivalenzrelation:
  - a.equals(null) ist immer false
  - Reflexiv: a.equals(a) ist immer true
  - Symmetrisch: a.equals(b) ⇒ b.equals(a)
  - Transitiv: a.equals(b) und b.equals(c) ⇒ a.equals(c)
  - Das Ergebnis bleibt konstant bei Wiederholungen.

hashCode und equals

- Gerade für die Suche in Collection Klassen ist der sogenannte Hashcode wichtig, der von der hashCode Methode geliefert wird.
- Wenn zwei Objekte gleich hinsichtlich der equals Operation sind, so müssen sie auch beide den selben hashCode haben.
- Dies heisst i.A. nicht, dass zwei ungleiche Objekte auch ungleichen Hashcode haben. Beispiel:
  - hashCode liefert ein int, jedoch bereits die Klasse Long hat mehr mögliche Instanzen, als sich per int referenzieren lassen.
  - $\Rightarrow \exists\ a,b \in Long \mid !a.equals(b)$ und $a.hashCode \equiv b.hashCode$

equals implementieren

```
public class Person {
    private String name;
    private Address address;
    public boolean equals(Object obj) {
        if (obj == null)
            return false;
        if (obj == this)
            return true;
        if (obj.getClass() == this.getClass()) {
            Person p = (Person) obj;
            return this.name.equals(p.name)
                && this.address.equals(p.address);
        } else {
            return false;
        }
    }
```

Weshalb kein instanceof ?

hashCode implementieren

```
public int hashCode() {
    return name.hashCode()
           ^address.hashCode();
}
```

- Die hashCode Implementierung muss passend zu equals implementiert werden. D.h. muss die selben Variablen verwenden.
- Gute Hashcode Algorithmen zu finden in:
  - D.E. Knuth, The Art of Computer Programming Vol. 3
- Häufig reicht eine XOR Verknüpfung der Attribut Hashcodes aus oder *Objects.hash(Object …)*.

JDK1.7

Vermeidung von NullPointern

- Nicht nur in Java führen nicht initialisierte Referenzen zu Problemen.
- Eine defensive Programmierung führt dann dazu Referenzen vor dem Gebrauch immer auf Null zu überprüfen. Dies bläht den Code unnötig auf.
- Wo immer möglich sollen NullPointer daher vermieden werden, insbesondere als Rückgabewert von Methoden.
- Beispiel Suche, anstatt eines NullPointers
  - entweder ein Array der Länge 0
  - oder eine Collection ohne Inhalt

NullPointer bei Feldern

```
public String[] notGood(String query) {
    String retVal[] = null;
    if (query != null) {
       retVal = findArray(query);
    }
    return retVal;
}

public String[] thatsRight(String query) {
    String retVal[];
    if (query != null) {
        retVal = findArray(query);
    } else {
        retVal = new String[0];
    }
    return retVal;
}
```

@NotNull
ab JDK 1.9
überprüft?

java.util.Objects

- Seit dem JDK 1.7 nimmt sich die Klasse Objects dieser Problematiken an. Sie gibt NP-freie Referenzen zurück – oder wirft eine NPException, generiert HashCodes und implementiert deepEquals.
- Sie ist einzusetzen, solange die @NotNull Annotation noch nicht richtig standardisiert ist und von Compiliern berücksichtigt wird.
- @NotNull wird schon zur Compile Zeit bewertet, die Methoden der Klasse Objects tragen erst zur Lauf- dh. Entwicklungs- bzw. Debugzeit zur Verbesserung der Codequalität bei. Beides erzieht zu einem “guten Programmierstil” ...

Beispiel aus dem JDK

```
/**
 * If the processor class is annotated with {@link
 * SupportedOptions}, return an unmodifiable set with the same set
 * of strings as the annotation.  If the class is not so
 * annotated, an empty set is returned.
 *
 * @return the options recognized by this processor, or an empty
 * set if none
 */
public Set<String> getSupportedOptions() {
    SupportedOptions so = this.getClass().getAnnotation(SupportedOptions.class);
    if  (so == null)
        return Collections.emptySet();
    else
        return arrayToSet(so.value());
}
```

- Dieses Beispiel aus dem JDK zeigt die Rückgabe eines leeren Set<String> anstatt eines Nullzeigers.

NP Vermeidung JDK1.7 ++

```
public String[] thatsRight(String query) {
    String retVal[] = new String[0];
    query = Objects.requireNonNull(query);
    retVal = findArray(query);
    assert retVal != null : “bug detected”;
    return retVal;
}

public @NonNull String[] thatsRight(
          @NonNull String query) {
    String retVal[] = new String[0];
    retVal = findArray(query);
    assert retVal != null : “bug detected”;
    return retVal;
}
```

Java Konstruktor

- Impementiere Konstruktoren so dass:
  - Der Client nicht zu viele Argumente liefern muss,
  - das Objekt in einem definierten Zustand ist und
  - sich die Klasse noch erweitern lässt.
- Konstruktoren lassen sich per protected verstecken und die Klasse ist dennoch offen für Erweiterung...
- GUI Klassen besitzen einen leeren Konstruktor, um der Java Bean Konvention zu genügen.
- Ein Copy-Konstruktor ist meist nützlich...

Mehrere Konstruktoren

Constructor Summary
`JTable()` Constructs a default `JTable` that is initialized with a default data model, a default column model, and a default selection model.
`JTable(int numRows, int numColumns)` Constructs a `JTable` with `numRows` and `numColumns` of empty cells using `DefaultTableModel`.
`JTable(Object[][] rowData, Object[] columnNames)` Constructs a `JTable` to display the values in the two dimensional array, `rowData`, with column names, `columnNames`.
`JTable(TableModel dm)` Constructs a `JTable` that is initialized with `dm` as the data model, a default column model, and a default selection model.
`JTable(TableModel dm, TableColumnModel cm)` Constructs a `JTable` that is initialized with `dm` as the data model, `cm` as the column model, and a default selection model.
`JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)` Constructs a `JTable` that is initialized with `dm` as the data model, `cm` as the column model, and `sm` as the selection model.
`JTable(Vector rowData, Vector columnNames)` Constructs a `JTable` to display the values in the `Vector` of `Vectors`, `rowData`, with column names, `columnNames`.

Immutable Objekte

- Häufig werden unveränderliche Objekte benötigt.
- Diese haben lediglich lesende getter-Methoden.
- Der interne Status wird im Konstruktor initialisiert.

```
public class Address {
    private final String street;
    private final String zipCode;
    private final int countryCode;
    /**
     * Immutable Object construction
     */
    public Address(String street, String zip, int country) {
        this.street = Objects.requireNonNull(street);
        this.zipCode = zip;
         countryCode = country;
    }
```

=>Immutable records
1.st preview JDK 14

Function Pointer

- Überall dort, wo ein C Programmierer einen FunctionPointer verwendet, z.B. Callback Methoden, werden in Java Schnittstellen eingesetzt.
- Schnittstellen erlauben ein sauberes Typsystem und gestatten es Fehler bereits zur Compilezeit zu entdecken.
- Die statische Schnittstelle kann dann zur Laufzeit dynamisch mit beliebigen Implementierungen dieser Schnittstelle bestückt werden.
- ClassCastExceptions/falsches Auflösen von void* Methodenpointern sollte es in Java nicht geben.

FP per Reflection

- Dynamisch lassen sich FunctionPointer in Java per Reflection erzeugen.
- Beliebige Methoden können mit Hilfe der Java Method-Klasse dynamisch an einem Objekt aufgerufen werden.
- Dies ist allerdings i.A. nicht so performant und wird z.B. bei automatischen Proxy Klassen verwendet.
  - Nicht jede JVM bietet Reflection an, z.B. auf Mobiles oder anderen embedded Geräten.
  - Reflection lässt sich durch den Java Security Manager abschalten.
  - Lego Robots unterstützen Reflection nicht.

Java Closure

- Mit dem JDK1.8 ist die Java Sprache um sogenannte Closure Ausdrücke erweitert worden.
- Funktionen lassen sich so direkt als Objekte referenzieren und an Methoden weitergeben.
- Mehr dazu in einem eigenständigen Block zum Thema Closures...

Methodenaufruf per Reflection

```
Class argTypes[] = null;
Object argList[] = null;
Person p = new Person();
System.out.println("Person " + p.toString());

// Java FunctionPointer
Method m = Person.class.getMethod(
                    "toString", argTypes);
// Aufruf der Methode am Objekt
Object ret = m.invoke(p,argList);
System.out.println("ByReflection: " + ret);
```

"Normaler Methodenaufruf"

FunctionPointer per Reflection

Schnittstelle und Implementierung

- Java unterstützt aktiv die Verwendung von Schnittstellen. Es gibt die Bezeichner **class** und **interface**, um beide Konzepte auf Sprachebene zu realisieren.
- Wo immer möglich sollten Klassen gegen die Schnittstelle implementiert werden, ohne von der Implementierungsklasse abzuhängen.
- Dies erleichtert den transparenten Austausch der Implementierung.
- Durch geeignete Design Muster ist dann auch der **new** Operator zu verstecken.

Immutable Schnittstelle

- Nicht immer ist es möglich Immutable Objekte per Konstruktor zu erzeugen.
- Dann bietet es sich an eine Schnittstelle mit nur read-only getter-Methoden zu implementieren.
- An Clienten wird nur diese Schnittstelle veröffenlicht, intern können entsprechende setter-Methoden verwendet werden.
- Um einen Cast auf der Clientseite zu verhindern lassen sich diese setter-Methoden mit package oder protected scope nach außen verstecken ...

PackageScope Implementierung

```
public interface Money {
    double getValue();
}
class MoneyImpl implements Money {
    private double value;
    public double getValue() {
        return value;
    }
    void setValue(double amount) {
        value = amount;
    }
}
public class MoneyFactory {
   public static Money createMoney(double amount){
      MoneyImpl m = new MoneyImpl();
      m.setValue(amount);
      return m;
   }
```

Final Mutable

- Das Gegenteil zu nichtveränderbaren Immutables sind final Variable, die dennoch änderbar sind.

```
public class Mutable {
    public static class Integer {public int value;}
    public static class Long    {public long value;}
    public static class Double  {public double value;}
}

  final Mutable.Integer counter = new Mutable.Integer();
  for(int i=0;i<10;i++) {
       ++counter.value;
  }
  System.out.println("counter: " + counter.value);
```

Zusammenfassung

- Idioms sind auf der jeweils untersten Sprachebene angesiedelt und weder sprach- noch architekturneutral.
- Dies unterscheidet sie von den Design Mustern. Idiome sind dennoch wichtig für den Projekterfolg.
- Einheitliche Berücksichtigung von Idiomen innerhalb eines Projekt(teams) vereinfacht den Programmierstil und hilft bei der Fehlererkennung und -vermeidung.
- Tools wie FindBugs und Checkstyle helfen schon jetzt.
- Vermutlich mit JDK 1.9+ werden neue Annotations Überprüfungen eingeführt, so dass ***@NotNull, @Nullable*** etc. vom Compiler überprüft werden...