

Vorlesung Programmieren

14. Best Practices

PD Dr. rer. nat. Robert Heinrich



Organisatorisches: Präsenzübung



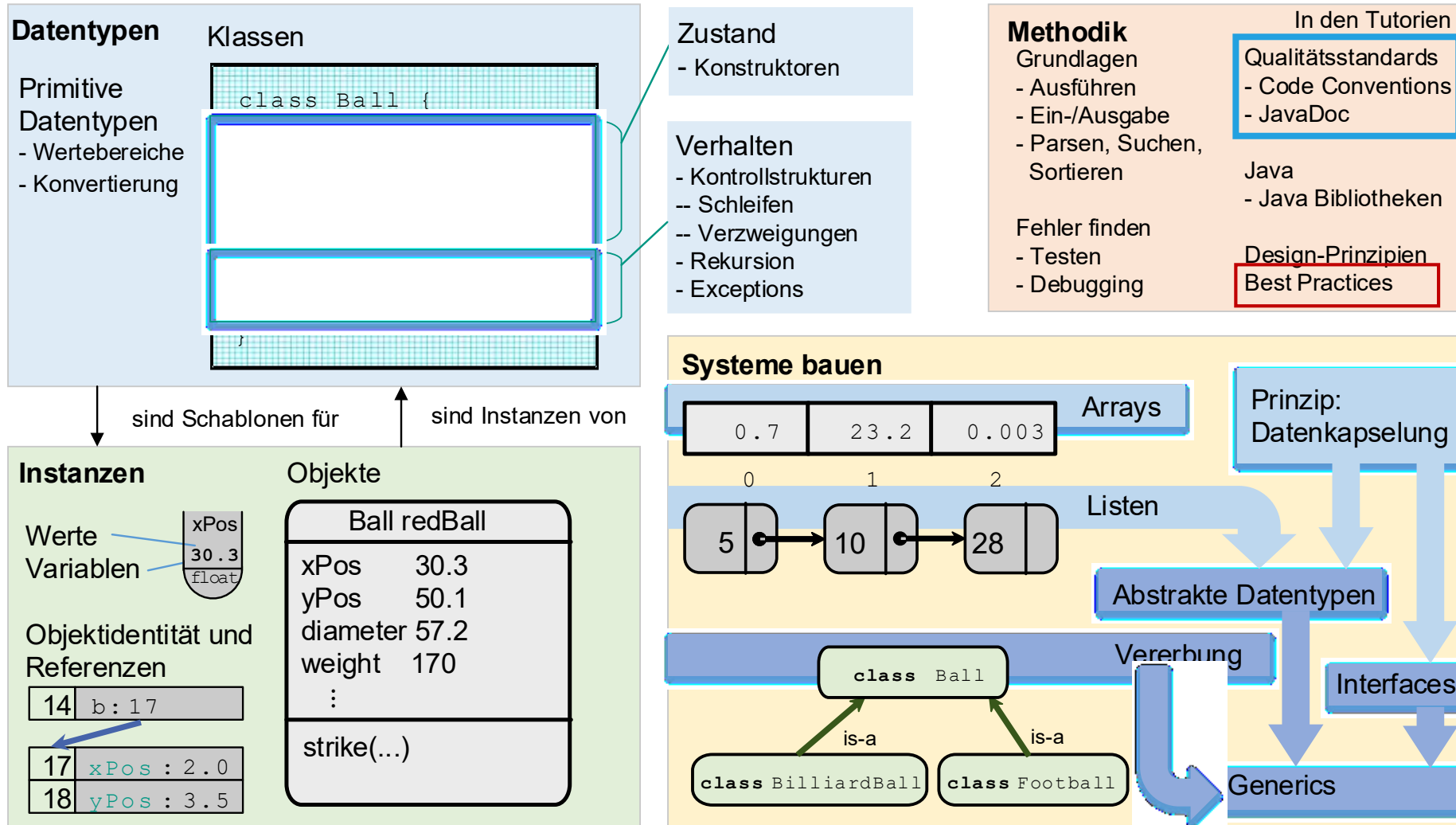
Beachten Sie die
Ankündigungen in
ILIAS

- **Diesen Freitag, der 21.01.2022**
 - Zeitfenster A: 18:20 – 18:40 Uhr
 - Zeitfenster B: 19:20 – 19:40 Uhr

- **Es gilt 3G (Genesen, Geimpft, Getestet) + FFP2-Maskenpflicht:**
Zertifikate müssen nach EU-Norm digital (QR-Code) auslesbar sein
- **Hörsaaleinteilung** in der SDQ-News-List: <https://s.kit.edu/newslist>
- Bitte seien Sie pünktlich vor dem Hörsaal: **Sie sollten mind. 20 Minuten früher da sein**
- Bitte merken Sie sich die **Nummer Ihres Tutoriums** und bringen Sie einen amtlichen Lichtbildausweis (z.B. Personalausweis) und Studierendenausweis (KITCard) mit
- Verwenden Sie bitte einen nicht-roten, dokumentenechten Stift (z.B. Kugelschreiber)

- **Schülerstudierende** sind automatisch angemeldet: Gottlieb-Daimler HS ab 19:20 Uhr

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Lernziele



- Sie kennen Best Practices für oft auftretende Entscheidungen im Entwurf von Java-Programmen
- Sie können die fünf vorgestellten Prinzipien anwenden
- Sie kennen den Einsatzzweck der Prinzipien
- Sie können gut lesbaren und gut wartbaren Code in Java schreiben



Quelle: <http://phdcomics.com>



Was erwartet Sie heute?

Prinzipien des guten Programmierens in Java:

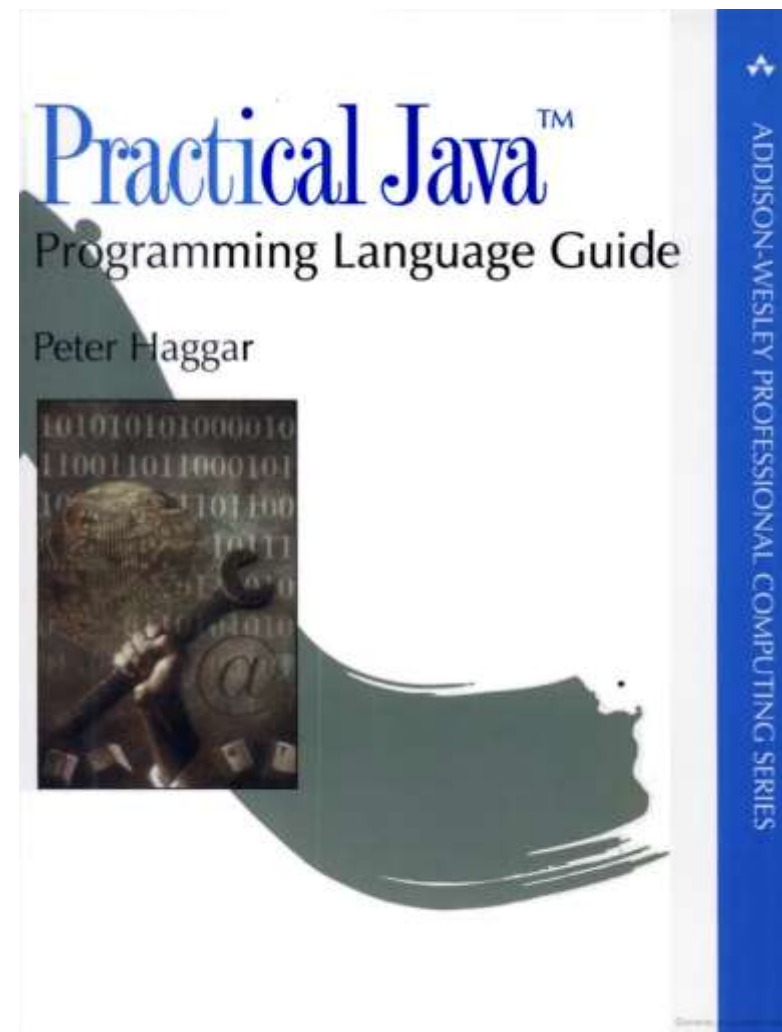
- Polymorphie vs. **instanceof**
- Unterscheidung zwischen `==` und `equals()`
- Standardimplementierung von `equals()` überschreiben
- Vorsicht bei der Implementierung von `equals()`
- **super**.`equals()` verwenden
- Vorsicht bei der Verwendung von **instanceof** in `equals()`

Java-spezifische Praxistipps (gegenüber allgemeinen OO-Entwurfsprinzipien in Foliensatz 15)

Literaturhinweis – Weiterlesen

 Peter Hagggar: *Practical Java™ Programming Language Guide*, Addison-Wesley Professional, 2000

Kapitel: 5/9/10/11/13/14



Prinzip 1

Bevorzuge Polymorphie gegenüber *instanceof*

Anytime you find yourself writing code of the form “if the object is of type T1, then do something, but if it’s of type T2, then do something else”, slap yourself.

📄 Scott Meyers: *Effective C++, 55 Specific Ways to Improve Your Programs and Design*, Addison-Wesley Professional, 3. Ausgabe, 2005





instanceof

- **Instanceof** bestimmt zur Laufzeit, zu welcher Klasse ein Objekt gehört.
- wird häufig im falschen Kontext eingesetzt.
- → kann in vielen Fällen durch Polymorphie ersetzt werden.



Beispiel für schlechten Entwurf

```
interface Employee {  
    public int salary();  
}  
  
class Manager implements Employee {  
    private static final int mgrSal = 40000;  
    public int salary() { return mgrSal; }  
}  
  
class Programmer implements Employee {  
    private static final int prgSal = 50000;  
    private static final int prgBonus = 10000;  
    public int salary() { return prgSal; }  
    public int bonus() { return prgBonus; }  
}
```

Was passiert, wenn man eine weitere Gehaltsgruppe hinzufügen will?

```
class Payroll {  
    public int calcPayroll(Employee emp){  
        int money = emp.salary();  
        if (emp instanceof Programmer)  
            money+=((Programmer)emp).bonus();  
        return money;  
    }  
  
    public static void main(String args[]) {  
        Payroll pr = new Payroll();  
        Programmer prg = new Programmer();  
        Manager mgr = new Manager();  
        System.out.println("Programmer payroll: "  
            + pr.calcPayroll(prg) );  
        System.out.println("Manager payroll: "  
            + pr.calcPayroll(mgr) );  
    }  
}
```



Beispiel für geeigneten Entwurf

```
interface Employee {
    public int salary();
    public int bonus();
}

class Manager implements Employee {
    private static final int mgrSal = 40000;
    private static final int mgrBonus = 0;
    public int salary(){ return mgrSal; }
    public int bonus() { return mgrBonus; }
}

class Programmer implements Employee {
    private static final int prgSal = 50000;
    private static final int prgBonus = 10000;
    public int salary(){ return prgSal; }
    public int bonus() { return prgBonus; }
}
```

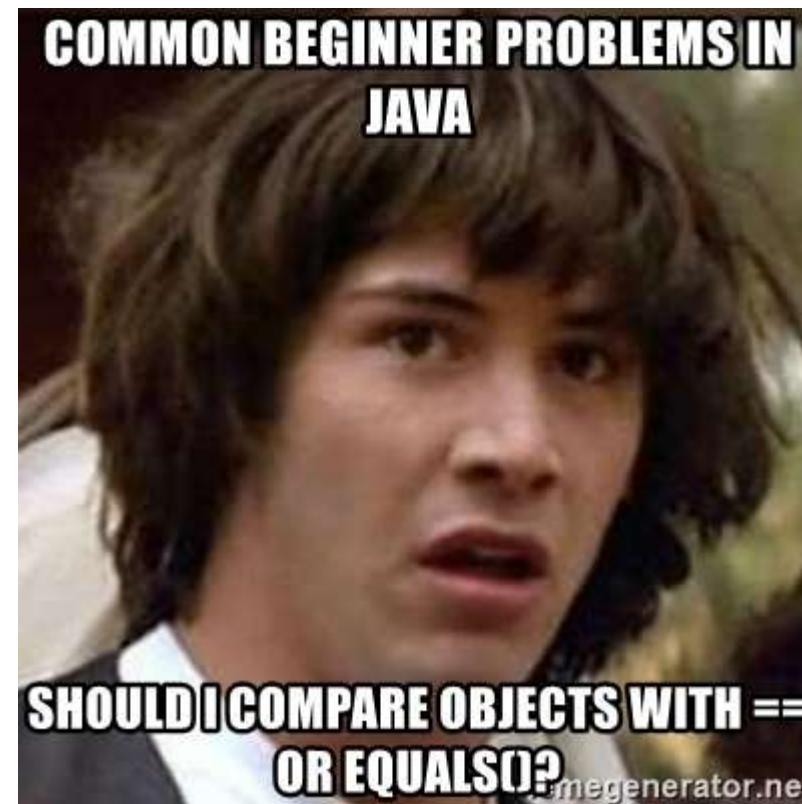
```
class Payroll {
    public int calcPayroll(Employee emp){
        return emp.salary() + emp.bonus();
    }

    public static void main(String args[]) {
        Payroll pr = new Payroll();
        Programmer prg = new Programmer();
        Manager mgr = new Manager();
        System.out.println("Programmer payroll: "
                           + pr.calcPayroll(prg) );
        System.out.println("Manager payroll: "
                           + pr.calcPayroll(mgr) );
    }
}
```

Prinzip 2

Unterscheidung zwischen `==` und `equals()`

- Was ist der Unterschied zwischen `==` und `equals()`?
- Kann ich alles mit `==` vergleichen?
- Wo brauche ich `equals()`?





Unterscheidung zwischen == und equals()

```
class Test {  
    public static void main(String args[]) {  
        int a = 10;  
        int b = 10;  
        System.out.println("a==b is " + (a==b));  
  
        Integer ia = new Integer(10);  
        Integer ib = new Integer(10);  
        System.out.println("ia==ib is " + (ia==ib));  
    }  
}
```

Programmausgabe:

a==b is true

ia==ib is false

Warum verhält sich das Programm so?

- a und b sind vom Typ int
(**primitiver Datentyp**)
- a und b haben keine korrespondierenden Objekte
→ a==b liefert also den Vergleich zwischen Werten
- ia und ib sind hingegen vom Typ Integer, somit **Objektreferenzen**
- ia und ib zeigen auf unterschiedliche Objekte
→ ia==ib liefert also den Vergleich zwischen den Objektreferenzen!



`equals()`

- **Frage:** Wie vergleicht man dann die gespeicherten Werte in Objekten?
- **Antwort:** `equals()` liefert einen semantischen Vergleich!
- Da `equals()` eine Methode ist, kann sie nur zum Vergleich von Objekten, **nicht** aber zum Vergleich von primitiven Datentypen benutzt werden

Weiteres Beispiel

```
class Test {  
    public static void main(String args[]) {  
        int a = 10;  
        float b = 10.0f;  
        System.out.println("a==b is " + (a==b));  
  
        Integer ia = new Integer(10);  
        Integer ib = new Integer(10);  
        Float fb = new Float(10.0f);  
        System.out.println("ia.equals(ib) is " + (ia.equals(ib)));  
        System.out.println("ia.equals(fb) is " + (ia.equals(fb)));  
    }  
}
```

Erweiternde Primitivkonvertierung
(*widening primitive conversion*)

Objekte unterschiedlicher Klassen
werden meistens nicht als gleich
betrachtet (Ausnahme s. später).

Programmausgabe:

```
a==b is true  
ia.equals(ib) is true  
ia.equals(fb) is false
```

Prinzip 3



Die Standardimplementierung von
`equals()` überschreiben





equals() überschreiben

```
class Golfball {  
    private String brand;  
    private String make;  
    private int compression;  
    public Golfball(String brd, String mk, int comp) {  
        brand = brd;  
        make = mk;  
        compression = comp;  
    }  
    public String brand() { return brand; }  
    public String make() { return make; }  
    public int compression() { return compression; }  
}
```

Zwei Golfbälle sollen genau dann gleich sein, wenn alle Attribute jeweils denselben Wert haben.

```
class Warehouse {  
    public static void main(String args[]) {  
        Golfball gb1 = new Golfball(  
            "BrandX", "Professional", 100);  
        Golfball gb2 = new Golfball(  
            "BrandX", "Professional", 100);  
        if (gb1.equals(gb2)) {  
            System.out.println(  
                "Ball 1 equals Ball 2");  
        } else {  
            System.out.println(  
                "Ball 1 does not equal Ball 2");  
        }  
    }  
}
```

Programmausgabe:

Ball 1 does not equal Ball 2



Implementierung der equals()-Methode

- Warum funktionierte equals() im Beispiel von Prinzip 2, jedoch hier nicht?
- Das Beispiel benutzte Objekte der Integer-Klasse. Diese Klasse hat ihre eigene Implementierung der equals()-Methode
- Golfball hat hingegen keine eigene Implementierung der equals()-Methode. In diesem Fall wird die Implementierung der equals()-Methode der Klasse java.lang.Object verwendet.
- Wie sieht die Implementierung der equals()-Methode der Klasse java.lang.Object aus?

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```



Abhilfe durch eigene Implementierung

```
class Golfball {  
    // (...), wie zuvor  
    public boolean equals(Object obj) {  
        if (this == obj) { return true };  
        if (obj != null && getClass() == obj.getClass()) {  
            Golfball gb = (Golfball) obj;  
            if (brand.equals(gb.brand()) &&  
                make.equals(gb.make()) &&  
                compression == gb.compression()) {  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

Programmausgabe:

Ball 1 equals Ball 2

Erwartetes Verhalten

... solange die Attribute vom
Typ String sind.



Probleme der vorherigen Implementierung

```
class Golfball {  
    private StringBuffer brand;  
    private StringBuffer make;  
    private int compression;  
    public Golfball(StringBuffer brd, StringBuffer mk, int comp) {  
        brand = brd;  
        make = mk;  
        compression = comp;  
    }  
    public StringBuffer brand() { return brand; }  
    public StringBuffer make() { return make; }  
    public int compression() { return compression; }  
    // (...), wie zuvor  
}
```

Ändern des Typs der
Attribute `make` und `brand`
von `String` zu
`StringBuffer`.

Programmausgabe:

Ball 1 does not equal
Ball 2

Problem: `equals()`-
Implementierung der
`StringBuffer`-Klasse



Lösungsidee 1

```
class Golfball {  
    // (...), wie zuvor  
    public boolean equals(Object obj) {  
        if (this == obj) { return true; }  
        if (obj != null && getClass() == obj.getClass()) {  
            Golfball gb = (Golfball) obj;  
            if (brand.toString().equals(gb.brand().toString()) &&  
                make.toString().equals(gb.make().toString()) &&  
                compression == gb.compression()) {  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

Problem:

Aufrufe von `toString()` führen zur Erzeugung vieler neuer String-Objekte
→ Lösung ist sehr teuer!

- Benutzen von `String` statt `StringBuffer`
- `equals()`-Methode von `Golfball` wird so verändert, dass sie `StringBuffer`-Objekte zu `String`-Objekten konvertiert
→ `equals()`-Methode von `String` wird zum Vergleich verwendet



Lösungsidee 2

- Eigene Klasse `MyStringBuffer` schreiben, die `equals()`-Methode enthält
- Eigene Implementierung von `equals()`-Methode verwenden
- Implementierung ist aufwendiger, jedoch sehr vorteilhaft
- Implementierung auf der nächsten Folie zeigt eine Enthaltensein-Beziehung (Komposition) zwischen `MyStringBuffer` und `StringBuffer`
- Grund: `StringBuffer` ist eine **final class**
→ Andere Klassen können von ihr nicht weiter erben



Lösungsidee 2: Eigene Klasse

```
class MyStringBuffer {
    private StringBuffer stringBuffer;

    public MyStringBuffer(String str) {
        stringBuffer = new StringBuffer(str);
    }

    public int length() {
        return stringBuffer.length();
    }

    public synchronized char charAt(int index) {
        return (stringBuffer.charAt(index));
    }
    // ...
}
```

```
public boolean equals(Object obj) {
    if (this == obj) { return true; }
    if (obj != null && getClass() == obj.getClass()) {
        MyStringBuffer sb = (MyStringBuffer) obj;
        int len = length();
        if (len != sb.length()) {return false; }
        int index = 0;
        while (index != len) {
            if (charAt(index) != sb.charAt(index)) {
                return false;
            } else { index++; }
        }
        return true;
    }
    return false;
}
```



Lösungsidee 2: Beispiel Golfball

```
class Golfball {
    private MyStringBuffer brand;
    private MyStringBuffer make;
    private int compression;
    public Golfball(MyStringBuffer brd,
        MyStringBuffer mk, int comp) {
        brand = brd;
        make = mk;
        compression = comp;
    }
    public MyStringBuffer brand() {return brand;}
    public MyStringBuffer make() {return make;}
    public int compression() {
        return compression;
    }
    // ...
}
```

```
public boolean equals(Object obj) {
    if (this == obj) { return true; }

    if (obj != null && getClass() ==
obj.getClass()) {
        Golfball gb = (Golfball) obj;
        if (brand.equals(gb.brand()) &&
            make.equals(gb.make()) &&
            compression == gb.compression()) {
            return true;
        }
    }
    return false;
}
```



Lösungsidee 3

- `equals()`-Methode zunächst nicht weiter berücksichtigen
- Eigene `compare()`-Methode implementieren, die zwei `StringBuffer`-Objekte vergleicht.
- `equals()`-Methode der `Golfball`-Klasse so verändern, dass sie statt `equals()`-Methode `compare()`-Methode verwendet.
- **Vorteile:** Implementierung weiterer Klassen nicht nötig
- **Nachteile:** `equals()`-Methode der `Golfball`-Klasse ruft eine `compare()`-Methode statt eine `equals()`-Methode auf.
→ Wird in Zukunft beispielsweise wieder ein `String`-Objekt benutzt, muss die Implementierung der `equals()`-Methode geändert werden.

Prinzip 4

Vorsicht bei der Implementierung
von equals()





Wichtige Fragen bei der Implementierung

Beantworten Sie folgende Fragen, bevor Sie `equals()`-Methode neu implementieren.

Benötigt die Klasse überhaupt eine `equals()`-Methode?

- Nur wenn die Objekte der Klassen logisch miteinander verglichen werden müssen, d.h. wenn der Vergleich zweier Referenzen nicht genügt.
- Die zuvor gezeigte Implementierungen der `equals()`-Methode können nicht ohne weiteres auf alle Implementierungen angewendet werden.

Welche Objekte sollen verglichen werden?

- Nur Objekte derselben Klasse?
 - Objekte der Unterklassen mit Objekten der Basisklasse?
- Ihre Implementierung muss dies widerspiegeln



Prinzip 5

`super.equals()` verwenden

- Was passiert mit der `equals()`-Methode bei Vererbung?

**SUPER
EQUALS**



Beispiel für fehlerhaften Code

```
class MyGolfball extends Golfball {  
    public enum Construction  
        {TWO_PIECE, THREE_PIECE};  
    private Construction ballConstruction;  
  
    public MyGolfball(String brd, String mk,  
        int comp, Construction construction)  
    {  
        super(brd, mk, comp);  
        ballConstruction = construction;  
    }  
  
    public Construction construction() {  
        return ballConstruction;  
    }  
    // ...  
}
```

```
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj != null && getClass() ==  
        obj.getClass()) {  
        MyGolfball gb = (MyGolfball) obj;  
        if (ballConstruction == gb.construction())  
            return true;  
    }  
    return false;  
}
```

Zwei Golfbälle werden bereits als gleich identifiziert, wenn nur das Attribut **ballConstruction** dieselben Werte hat



Beispiel für fehlerhaften Code

```
class Warehouse {  
    public static void main(String args[]) {  
        MyGolfball gb1 = new MyGolfball("BrandX", "Professional", 100, MyGolfball.TWO_PIECE);  
        MyGolfball gb2 = new MyGolfball("BrandX", "Professional", 90, MyGolfball.TWO_PIECE);  
        // ...  
        if (gb1.equals(gb2))  
            System.out.println("Ball 1 equals Ball 2");  
        else  
            System.out.println("Ball 1 does not equal Ball 2");  
    }  
}
```

Obwohl die beiden Objekte einen unterschiedlichen Wert für Kompression besitzen, liefert `equals()`:

Ball 1 equals Ball 2



Lösung

Warum verhält sich das Programm so?

- Die equals()-Methode von MyGolfBall **überschreibt** jene von Golfball
- Soll der Code der Basisklasse auch ausgeführt werden, muss **super.equals()** explizit aufgerufen werden:

```
public boolean equals(Object obj) {  
    if (this == obj) { return true; }  
    if (obj != null && getClass() == obj.getClass() && super.equals(obj)) {  
        MyGolfball gb = (MyGolfball) obj;  
        if (ballConstruction == gb.construction()) {  
            return true;  
        }  
    }  
    return false;  
}
```



Alternative Lösung

```
public boolean equals(Object obj) {  
    if (super.equals(obj)) {  
        MyGolfball gb = (MyGolfball) obj;  
        if (ballConstruction == gb.construction()) {  
            return true;  
        }  
    }  
    return false;  
}
```

- **Vorsicht** beim Benutzen dieser Lösung!
- Hier wird die Annahme getroffen, dass die Überprüfung auf **null** und auf korrekten Typ in der equals()-Methode der Basisklasse geschieht
- Sie müssen also wissen, wie die equals()-Methode in der Basisklasse implementiert ist

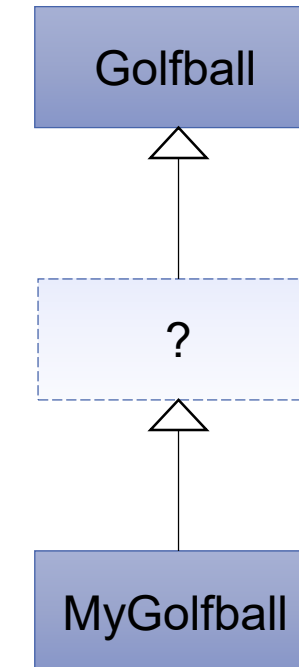


Mehrere Ebenen in der Vererbungshierarchie

- Was passiert, wenn weitere Klassen in der Vererbungshierarchie zwischen den beiden Klassen Golfball und MyGolfball hinzugefügt wird?
- Da MyGolfball `super.equals()` aufruft, funktioniert alles fehlerfrei.

Daumenregel

Sobald eine Basisklasse die `equals()`-Methode implementiert, muss die `equals()`-Methode der abgeleiteten Implementierung `super.equals()` aufrufen. (Ausnahme: Basisklasse ist `java.lang.Object`)



Prinzip 6

Vorsicht bei der Verwendung von `instanceof` in `equals()`

- Können Objekte der abgeleiteten Klassen mit Objekten der Basisklasse verglichen werden?





Vergleich der Objekte

Bisher wurden Objekte von Klassen in derselben Vererbungsebene miteinander verglichen.

→ Verwendung von `getClass()`

Wann können die Objekte der abgeleiteten Klassen mit den Objekten der Basisklasse verglichen werden?

- Wenn die abgeleitete Klasse nur das Verhalten der Basisklasse ändert
- Die relevanten Attribute bleiben also gleich

Wie können die Objekte der abgeleiteten Klassen mit den Objekten der Basisklasse verglichen werden?

- Die Basisklasse benutzt in der Implementierung der `equals()`-Methode `instanceof` statt `getClass()`
 - Zur Erinnerung: `getClass()` → Die Objekte der abgeleiteten Klassen sind niemals gleich mit den Objekten der Basisklasse
- Die abgeleitete Klasse darf `equals()`-Methode nicht implementieren.



Beispiel mit Beibehaltung der relevanten Attribute

```
class Golfball {  
    public boolean equals(Object obj) {  
        if (obj instanceof Golfball) {  
            // compare brand and make  
        }  
    }  
}
```

```
class MyGolfball extends Golfball {  
}
```

```
Golfball g =  
    new Golfball("BrandX", "Professional", 100);  
MyGolfball m =  
    new MyGolfball("BrandX", "Professional", 100,  
        MyGolfball.TWO_PIECE);
```

Falls Attribute gleich:

`g.equals(m)` liefert **true**

`m.equals(g)` liefert **true**

Symmetrie-Eigenschaft ist gegeben:

`g.equals(m) == m.equals(g)`

→ kann so implementiert werden, wenn
für die Domäne sinnvoll



Beispiel mit hinzugefügten Attributen

```
class Golfball {  
    public boolean equals(Object obj) {  
        if (obj instanceof Golfball) {  
            // compare brand and make  
        } } }
```

```
class MyGolfball extends Golfball {  
    public boolean equals(Object obj) {  
        if (super.equals(obj)) {  
            // compare ball construction  
        } } }
```

```
Golfball g =  
    new Golfball("BrandX", "Professional", 100);  
MyGolfball m =  
    new MyGolfball("BrandX", "Professional", 100,  
        MyGolfball.TWO_PIECE);
```

Falls Attribute gleich:

`g.equals(m)` liefert **true**

`m.equals(g)` liefert **false**

Symmetrie-Eigenschaft geht verloren:

`g.equals(m) != m.equals(g)`

→ Sollte so nicht implementiert werden



Beispiel mit Beibehaltung der relevanten Attribute?

```
class Golfball {  
    public boolean equals(Object obj) {  
        if (obj instanceof Golfball) {  
            // compare brand and make  
        }  
    }  
}
```

```
class MyGolfball extends Golfball {  
    public boolean equals(Object obj) {  
        if (obj instanceof MyGolfball) {  
            // compare brand and make  
        }  
    }  
}
```

```
Golfball g =  
    new Golfball("BrandX", "Professional", 100);  
MyGolfball m =  
    new MyGolfball("BrandX", "Professional", 100,  
        MyGolfball.TWO_PIECE);
```

Falls Attribute gleich:

`g.equals(m)` liefert **true**

`m.equals(g)` liefert **false**

Symmetrie-Eigenschaft geht verloren:

`g.equals(m) != m.equals(g)`

Zwar relevante Attribute beibehalten, aber
`equals(..)` anders implementiert

→ Unterklasse darf `equals(..)` nicht
überschreiben

Beispiel mit *erzwungener* Beibehaltung der relevanten Attribute durch `final`



```
class Golfball {  
    public final boolean equals(Object obj) {  
        if (obj instanceof Golfball) {  
            // compare brand and make  
        }  
    }  
}
```

```
class MyGolfball extends Golfball {  
}
```

```
Golfball g =  
    new Golfball("BrandX", "Professional", 100);  
MyGolfball m =  
    new MyGolfball("BrandX", "Professional", 100,  
        MyGolfball.TWO_PIECE);
```

Falls Attribute gleich:

`g.equals(m)` liefert **true**

`m.equals(g)` liefert **true**

Symmetrie-Eigenschaft ist gegeben:

`g.equals(m) == m.equals(g)`

- kann so implementiert werden,
- garantiert, dass abgeleitete Klassen `equals` nicht überschreiben



Best Practices – Zusammenfassung

Prinzipien des guten Programmierens in Java:

- Polymorphie vs. **instanceof**
- Unterscheidung zwischen `==` und `equals()`
- Standardimplementierung von `equals()` überschreiben
- Vorsicht bei der Implementierung von `equals()`
- **super.equals()** verwenden
- Vorsicht bei der Verwendung von **instanceof** in `equals()`