Prof. Dr. A. Poetzsch-Heffter
Mathias Weber, M.Sc.

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2013)

Practice Sheet 1

Date of Issue: 16.04.13
Deadline: 23.04.13
(before the lecture as PDF via E-Mail)

Exercise 1 Java Programming

Implement the University Administration System from the lecture slides (in subsection 1.2) in Java, inluding the extension regarding assistants.

Exercise 2 The `equals` Method

The JDK description of the equals-Method is as follows:

The `equals` method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value x, `x.equals(x)` should return true.
- It is symmetric: for any non-null reference values x and y, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
- It is transitive: for any non-null reference values x, y, and z, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value x, `x.equals(null)` should return false.

The equals method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (`x == y` has the value true).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

a) Analyze the following code fragment checking for the aforementioned properties.

```
public class Date {
  private int y,m,d;

  public Date(int y, int m, int d) {
    this.y = y;
    this.m = m;
    this.d = d;
  }

  public boolean equals(Object obj) {
    if (obj instanceof Date) {
      Date date = (Date)obj;
      return date.y == y &&
             date.m == m &&
             date.d == d;
    }
    return false;
  }
}
```

```
public class NamedDate extends Date {
  private String name;

  public NamedDate(int y, int m, int d, String name) {
    super(y,m,d);
    this.name = name;
  }
  public boolean equals(Object other) {
    if (other instanceof NamedDate &&
        !name.equals(((NamedDate)other).name))
      return false;
    return super.equals(other);
  }
}
```

b) Consider the following implementation for `Date.equals`. What are the advantages and disadvantages of this solution?

```
public boolean equals(Object obj) {
  if (getClass() == obj.getClass()) {
    Date date = (Date)obj;
    return date.y == y && date.m == m && date.d == d;
  }
  return false;
}
```

c) Use the Java Language Specification (`http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`) to explain the output of the following program

```
public class IdentityEquality
{
  public static void main(String ... args)
  {
    String a1="A";
    String a2="A";

    String b1=new String("B");
    String b2=new String("B");

    System.out.println("a1==a2:␣"+(a1==a2));
    System.out.println("b1==b2:␣"+(b1==b2));

    System.out.println("a1.equals(a2):␣"+(a1.equals(a2)));
    System.out.println("b1.equals(b2):␣"+(b1.equals(b2)));
  }
}
```

Exercise 3 Happy Birthday

In a software project, the classes `Person`, `AgePerson` und `AgeManager` were implemented; their source is given in Figure 1.

a) For testing purposes, the following code is used.

```
...
AgeManager ageManager = new AgeManager();
AgePerson agePerson = new AgePerson("Jane","Doe",28);

ageManager.add("02.02.1980",agePerson);

while(true) {
  System.out.println("Is␣"+agePerson+"␣managed:␣"+ageManager.isManaged(agePerson));
  Thread.sleep(5000);
}
...
```

Explain the following output:

```
Is Jane Doe managed: true
Jane Doe: Age updated.
Is Jane Doe managed: false
Jane Doe: Age updated.
Is Jane Doe managed: false
Jane Doe: Age updated.
Jane Doe: Age updated.
Is Jane Doe managed: false
...
```

Hint: Look into the documentation for `Object.hashCode` *and* `Map`

b) Give a general programming guideline that helps to avoid theses problems.

c) Explain why the parameter *birthday* in `AgeManager.add` has to be *final*.

```
public class Person {
  protected String firstname;
  protected String lastname;

  public Person(String firstname,
                String lastname) {
    this.firstname=firstname;
    this.lastname =lastname;
  }

  public boolean equals(Object second) {
    if (second.getClass()==getClass()) {
      Person person=(Person)second;

      return(person.firstname.equals(firstname) &&
             person.lastname.equals(lastname));
    }

    return(false);
  }

  public int hashCode() {
    return(firstname.hashCode() ^
           lastname.hashCode());
  }

  public String toString() {
    return(firstname+"␣"+lastname);
  }
}

public class AgePerson extends Person {
  private int age;

  public AgePerson(String firstname,
                   String lastname,
                   int age) {
    super(firstname,lastname);

    this.age=age;
  }

  public void updateAge() {
    age++;

    System.out.println(this+":␣Age␣updated.");
  }

  public boolean equals(Object second) {
    if (second.getClass()==getClass()) {
      AgePerson agePerson=(AgePerson)second;

      return(agePerson.firstname.equals(firstname) &&
             agePerson.lastname.equals(lastname) &&
             agePerson.age==age);
    }

    return(false);
  }

  public int hashCode() {
    return(super.hashCode()^age);
  }
}
```

```
import java.util.*;
import java.util.Map.*;

import java.text.*;

public class AgeManager {
  private Map<AgePerson,Calendar>
    persons=new HashMap<AgePerson,Calendar>();

  private Calendar
    currentDate=new GregorianCalendar();

  public AgeManager() {
    new Timer().schedule(new TimerTask() {
      public void run() {
        currentDate.add(Calendar.DAY_OF_MONTH,1);

        updateAges(currentDate);
      }
    },new Date(),10);
  }

  protected synchronized void updateAges(Calendar date) {
    int month=date.get(Calendar.MONTH);
    int day =date.get(Calendar.DAY_OF_MONTH);

    for(Entry<AgePerson,Calendar> entry : persons.entrySet())
      if ((entry.getValue().get(Calendar.MONTH)==month) &&
          (entry.getValue().get(Calendar.DAY_OF_MONTH)==day))
        entry.getKey().updateAge();
  }

  public synchronized void add(final String birthday,
                               AgePerson person)
                               throws ParseException {
    persons.put(person,new GregorianCalendar() {{
       setTime(DateFormat.getDateInstance()
                         .parse(birthday));
    }});
  }

  public synchronized boolean isManaged(AgePerson person) {
    return(persons.containsKey(person));
  }
}
```

Figure 1: Sources for the classes `Person`, `AgePerson` and `AgeManager`.