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 3

Date of Issue: 30.04.13
Deadline: 07.05.13
(before the lecture as PDF via E-Mail)

Exercise 1 Reflection and Annotations

The `Proxy` class (`java.lang.reflect.Proxy`) allows to intercept method calls.

a) Inform yourself about the `Proxy` class using the JDK documentation. Is it a *normal* JDK class? Also inform yourself about Java annotations (for example under `http://docs.oracle.com/javase/tutorial/java/annotations/`).

b) Develop a generic wrapper using the `Proxy` class, which allows, for a certain object, upon invokation of one of its methods, to check annotated method parameters for non-nullness. An example of using the wrapper could look as follows:

```
public interface Example {
   public void print(@NonNull String s);
}

...

Example e = new Example() {
   public void print(String s) {
      System.out.println(s);
   }
};
e = (Example)Wrapper.wrap(e);
e.print("Hello"); // prints out "Hello"
e.print(null); // should throw an exception
```

The annotation type declaration is given as follows (does not need to be modified).

```
import java.lang.annotation.*;

@Target(ElementType.PARAMETER)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface NonNull {}
```

c) For which types and methods does the presented technique work?

Exercise 2 Inheritance and Subtyping

a) Explain the relation between inheritance and subtyping.

b) Explain the output of the following program:

```
class Dog {
   public static void bark() {
      System.out.print("woof␣");
   }
}

class Basenji extends Dog {
   public static void bark() { }
}

public class Bark {
   public static void main(String args[]) {
      Dog woofer = new Dog();
      Dog nipper = new Basenji();
      woofer.bark();
      nipper.bark();
   }
}
```

c) Define the role and semantics of the `@Override` annotation in Java. Explain the relation between overriding and dynamic dispatch. *Hint: use the Java Language Specification*

d) Analyze the following program:

```
package p;

public class Person {
  void print() {}
}

package q;
import p.Person;

public class Assistant extends Person {}

package p;
import q.Assistant;

public class PhDAssistant extends Assistant {
  public void print() {}
}
```

Does the `print` method in class `PhDAssistant` override the `print` method in class `Person` ? *Hint: look at the Java Language Specification in Section 8.4.8.*

Exercise 3 Super-Calls

Write a program, that would behave differently under the assumption of dynamically bound `super`-calls than it does with statically bound `super`-calls.

Exercise 4 *StoJas* Extension

In this exercise, we are going to build extensions to the Java subset *StoJas* that was presented in the lecture.

a) Extend the syntax as well as the semantics (static & dynamic) of *StoJas* to support a "`for(...) { ... }`" statement and give a detailed explanation for the necessary adjustments.

b) Extend the syntax as well as the semantics (static & dynamic) of *StooJas* to support static methods and give a detailed explanation for the necessary adjustments.

c) Extend the syntax as well as the semantics (static & dynamic) of *StooJas* to support static variables and give a detailed explanation for the necessary adjustments.

Exercise 5 Prototyping (optional)

The following exercise will not be discussed in the exercise hours.

In this exercise, we will encode the class-based approach to object-orientation using prototyping. First of all, have a look at the wikipedia page about prototype-based programming:
(`http://en.wikipedia.org/wiki/Prototype-based_programming`).

In pure prototyping there are no visible pointers or links to the original prototype from which an object is cloned. The prototype object is copied exactly, but given a different name (or reference). Behavior and attributes are simply duplicated as-is.

We want to implement the University Administration System from the lecture using prototyping with a class-based approach. As we want to do this in the (familiar) Java language, which offers no direct support for prototyping, we present a small library to provide prototyping facilities. Furthermore, we restrict the library user from using certain Java features. In fact, you are only allowed to use pre-defined (Java) types. However, it is allowed to create anonymous classes which are direct subtypes of the `Function` class. You are not allowed to use reflection.

```
public final class ProtoObj {
   /** Initial object */
   public static final ProtoObj INIT_PROTO = new ProtoObj();

   /** Tests whether the object has the given property */
   public final boolean hasProperty(String name) { ... }

   /** Set the property with name propertyName to the given value.
      If no such property exists, it is created. */
   public final void setProperty(String propertyName, Object value) { ... }

   /** Gets the value of the property with name propertyName */
   public final Object getProperty(String propertyName) { ... }

   /** Gets the value of the (function) property with name functionName.
    *  This is just a convenience method to avoid casting. */
   public final Function getFunction(String functionName) { ... }

   /** clone this object (copy all properties) */
   public final ProtoObj clone() { ... }
}

/** This class represents a function. To create a new function, override the run method */
public abstract class Function {
   public abstract void run(ProtoObj self, Object...args);
}
```

a) To get familiar with the prototype-based programming approach, you can look at the following example (similar to the one in the lecture):

```
ProtoObj vehicle = ProtoObj.INIT_PROTO.clone();
vehicle.setProperty("name", "␣");
ProtoObj sportsCar = vehicle.clone();
sportsCar.setProperty("driveToWork", new Function() {
  @Override
  public void run(ProtoObj self, Object... args) {
     System.out.println(self.getProperty("name") + "␣drives␣to␣work");
  }
});
ProtoObj porsche911 = sportsCar.clone();
porsche911.setProperty("name", "Bobs␣Car");

// call the driveToWork method on the porsche911 object
porsche911.getFunction("driveToWork").run(porsche911);
```

b) Implement the University Administration System in the spirit of the provided Java sources (UAS.zip). You should use the following template to implement the system (e.g. implement the methods `call` and `newObject`).

```
static ProtoObj PERSON_TYPE = ...
static ProtoObj STUDENT_TYPE = ...
static ProtoObj PROFESSOR_TYPE = ...
static ProtoObj ASSISTENT_TYPE = ...

public static void main(String...args) {
  ProtoObj michi = newObject(PERSON_TYPE, "Michi");
  ProtoObj john = newObject(STUDENT_TYPE, "John", 12345);
  ProtoObj einstein = newObject(PROFESSOR_TYPE, "Prof.␣Einstein", 402, "AG␣Relativ");
  ProtoObj peter = newObject(ASSISTENT_TYPE, "Peter", true);

  ProtoObj[] people = {john, michi, einstein, peter};

  for (ProtoObj p : people) {
     call(p, "print");
  }
}

public static void call(ProtoObj self, String functionName, Object...args) { ... }

public static ProtoObj newObject(ProtoObj type, Object...args) { ... }
```