ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 8

Parametric polymorphism and information hiding

November 15, 2013

Task 1

Consider the following Java method:

```
String concatenate(List<?> list) {
   String  result="";
   String separator="";
   if(list instanceof List<String>) {
      result="String:";
      separator=" ";
   }
   else if(list instanceof List<Integer>) {
      result="Integers:";
      separator="+";
   }
   for(Object el : list)
      result=result+separator+el.toString();
   return result;
}
```

- This program is rejected by Java compiler. Why?
- Using the advice given by the Java compiler, rewrite and compile the program. What are the results of executing the method passing each of the following:
  - A list of strings containing only one element "word"?
  - A list of Integers containing only one element Integer(1)?
  - A list of Objects containing only one element (initialized by new Object())?
- Is this behaviour consistent with what you would expect from the initial program? If not, how can you fix it?
- What would happen if you tried to implement the different cases using method overloading instead of just one method. Why is this the case?
- What happens if you compile and execute the initial program in C#? Why?

Task 2

Consider the following Java method:

```
public void add(Object value, List<?> list) {
   list.add(value);
}
```

The Java compiler rejects this program, with the following message:

```
The method add(capture#1-of ?) in the type List<capture#1-of ?> is not
applicable for the arguments (Object)
```

- Explain why we obtain such an error.
- Fix the program by using a generic type for the parameter of method `add` and constraining the wildcard appropriately.
- We can use the following alternative signature for `add`:

  ```
  public <V> void add(V value, List<V> list)
  ```

  Is this solution more restricted than the one obtained using the wildcard?
- Consider the following methods:

  ```
  public <V> void addAll(List<V> v, List<? super V> l) {
     for(V el : v) l.add(el);
  }
  public <V> void addAll1(List<V> v, List<V> l) {
     for(V el : v) l.add(el);
  }
  ```

  Method `addAll` is less restrictive than `addAll1`. Provide an example to prove this claim.

Task 3

A C++ template class can inherit from its template argument:

```
template <typename T>
class SomeClass : public T { ... }
```

Using this technique and given the following class definition

```
class Cell {
public:
   virtual void setVal(int x) { x_ = x; }
   virtual int value() { return x_; }
private:
   int x_{};
}
```

write two template classes that can be used as "mixins" for class `Cell`

- `Doubling` - doubles the value stored in the cell.
- `Counting` - counts the number of times the value of the cell was read.

Do not use multiple inheritance. It should be possible to use the classes like this:

```
auto c = new Doubling<Counting<Cell>>(); // instantiation
c->setVal(5);
c->value(); // returns 10
c->numRead(); // returns 1
```

- Describe how the instantiation above will look like.
- How does this concept of mixins in C++ differ from Scala traits?
- Can the code above be implemented using Java generics? If yes, show how. If no, explain why not.
- What if we used C# instead of Java, does anything change?

Task 4

Consider the following Scala code:

```
class A[-T]
class B[ ... T] {
   def m(in : A[T]) : int = {...}
}
```

We want to annotate the generic type of `B`. If we use a covariant or a contravariant annotation for the generic type parameter to `B`, what would that annotation be? Why? Justify your answer with an example.

Task 5

Java allows an object of a class `C` to access the private fields of other objects declared in `C`. Discuss the resulting level of information hiding, its advantages and limits, and provide some examples.

What is the policy concerning the visibility of protected fields of other objects?

Task 6

Consider the following Java code:

```
file A.java:
package p;
public class A {}

file B.java:
package p;
class B extends A {}

file C.java:
package p;
public class C extends B {          //1
  public B get(){...}               //2
  public void set(B d){...}         //3
}

file client.java:
package client;
import p.*;
class Client
  void f(){
    C c = new C();

    c.set(c.get());     //4
    c.set(c);           //5

    A a = c;            //6
  }
```

The code is accepted by the Java compiler.

- What seems inconsistent with information hiding in this code? Which lines would you expect the compiler to reject? Could allowing this still be useful in some cases?
- Does this problem exist in C++? C#? Scala? If not explain which line the respective compiler would complain about.
- Suggest a rule the Java compiler could enforce in order to be more consistent with information hiding - the rule should reject all marked lines but (1).

- Now suppose we wanted to support also line (6) (could you suggest why this might be useful?) how can we relax our new rule to allow this?