Concepts of Object-Oriented Programming
AS 2015

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 8

Parametric polymorphism

November 13, 2015

Task 1

Implement a list in Java or C# with two methods:

```
public void add(int i, Object el)
public Object get(int i)
```

Implement the list and discuss the advantages and the limitations of the three different approaches below.

A) Implement the list using only one class without generics.

B) Implement the list using one abstract class/interface and then (some) subclasses that implement it for different types.

C) Implement the list using generic types.

Task 2

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;
}
```

A) This program is rejected by the Java compiler. Why?

B) Using the advice given by the Eclipse Java compiler (replace `List<...>` with `List<?>`), 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())?

C) Is this behaviour consistent with what you would expect from the initial program? If not, how can you fix it?

D) What would happen if you tried to implement the different cases using method overloading instead of just one method. Why is this the case?

E) What happens if you compile and execute the initial program in C#? Why?

Task 3

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)
```

A) Explain why we obtain such an error.

B) Fix the program by using a generic type for the parameter of method `add` and constraining the wildcard appropriately.

C) 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?

D) Consider the following methods:

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

Method `addAllX` is less restrictive than `addAllY`. Provide an example to prove this claim.

Task 4 Wildcards

This is an extended version of a previous exam question.

Consider the following Java code:

```
interface Food {}
interface Grass extends Food {}
interface Meat extends Food{}
```

```
abstract class Animal<F extends Food> implements Meat{
   abstract void eat(F food);
   F getLunchBag(){ return lunchBag; };
   F lunchBag;
}

final class Sheep extends Animal<Grass>{ void eat(Grass f){} }
final class Wolf extends Animal<Meat>  { void eat(Meat  f){} }

class Cage { //You are allowed to modify this class
   Cage(Animal<?> animal){ this.animal = animal; }
   Animal<?> getAnimal() { return animal; }
   Animal<?> animal;
}

class Zoo{
  void feedAnimal(Cage cage){ /*code given in each section*/ }
  <F extends Food>void feed(F food, Animal<F> animal){animal.eat(food);}

   void manage(){ /*your code here*/ }
}
```

Clearly a `Wolf` can eat a `Sheep` but not the other way around. In the following subtasks we explore if relaxing some of the Java type rules can lead to a situation where a `Sheep` can eat a `Wolf` - that is, the method `eat` is called on an object of the dynamic type `Sheep` with an argument object of the dynamic type `Wolf`. All the code you give in the answers to the following sections is in the same package as the code above, and must type check in standard Java. The top method that is called in all sections is `Zoo.manage`. You can assume that method call arguments are evaluated left to right, and that the program is sequential. You are not allowed to use reflection, raw types, or type-casts.

A) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{ feed(cage.getAnimal().lunchBag,cage.getAnimal()); }
```

Make a `Sheep` eat a `Wolf` assuming the body of `feedAnimal` is exempted from the type checker. Show all necessary code. You are only allowed to change the `Cage` class and provide the body of the `Zoo.manage` method.

B) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{feed(cage.animal.getLunchBag(),cage.animal);}
```

Can you make a `Sheep` eat a `Wolf` if the body of `feedAnimal` is exempted from the type checker? If so, show all necessary code, otherwise explain why not. You are only allowed to change the `Cage` class, provide the body of the `Zoo.manage` method, and add new classes.

C) Answer the question in the previous section, assuming the field `Cage.animal` is `final`. Explain your answer. Reminder: Java's `final` fields can be assigned to only in the constructor of the class that declares them.

D) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type-checker:

```
{feed(cage.animal.lunchBag,cage.animal);}
```

Can you make a `Sheep` eat a `Wolf` if the body of `feedAnimal` is exempted from the type-checker? If so, show all necessary code, otherwise explain why not.

E) Which of the above cases, that is safe in the sequential case, is unsafe in a multithreaded program? For each such case, explain what can happen in the multithreaded case that cannot in the sequential case.

F) The current Java rule for evaluating an expression (including a method call) with wildcard typed arguments is to capture each wildcard in the arguments separately. Propose a more lenient wildcard capture rule than current Java, that is typesafe and accept all the above cases that you deem safe.
Hint: define "stable" paths that cannot be modified by calls.

Task 5

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

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

A) 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
```

B) Describe how the instantiation above will look like.

C) How does this concept of mixins in C++ differ from Scala traits?

D) Can the code above be implemented using Java generics? If yes, show how. If no, explain why not.

E) What if we used C# instead of Java, does anything change?