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.

solution

```
public class List {
   Object[] elements;
   public void add(int i, Object el) {elements[i]=el;}
   public Object get(int i) {return elements[i];}
}
```

Advantages: short implementation.

Limits: the type of the method result of `get` is `Object`. When using such a class, usually we have to dynamically cast the values returned by this method.

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

solution

```
public interface List {
   public void add(int i, Object el);
   public Object get(int i);
}

public class IntList implements List {
   Integer[] elements;

   public void add(int i, Object el) {elements[i]=(Integer) el;}

   public Integer get(int i) {return elements[i];}
}
```

Advantages: method `get` returns an `Integer`, thus we do not need dynamic casting of the values returned by this method.

Limits: in Java, we have the same limits like before, and in addition code duplication and additional type castings and checks in method `add`. Moreover, we do not have behavioural subtyping, since method `add` in `IntList` may not respect the expected contracts in `List`. In particular, if we invoke it passing an object that is not an instance of `Integer`, the runtime environment would raise an exception and the element would not be added to our list.

**C)** Implement the list using generic types.

solution

```
public class List<T> {
   T[] elements;
   public void add(int i, T el) {elements[i]=el;}
   public T get(int i) {return elements[i];}
}
```

Advantages: short implementation, statically type safe.

Limits: nothing! :) we have only advantages...

## 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?

solution

The Oracle and the Open JDK compilers both produce these short errors:

```
illegal generic type for instanceof
illegal generic type for instanceof
```

The Eclipse compiler tries to be more helpful:

```
Cannot perform instanceof check against parameterized type
List<String>. Use the form List<?> instead since further
generic type information will be erased at runtime
```

```
Cannot perform instanceof check against parameterized type
List<Integer>. Use the form List<?> instead since further
generic type information will be erased at runtime
```

This happens because of type erasure in Java.

**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())?

solution

First of all, we follow the output of the compiler, and so we rewrite the method to:

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

The Java compiler will compile this program without any warning. The output of the method is obviously:

```
String: word
String: 1
String: java.lang.Object@3e25a5
```

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

solution

No, in the original program we expected:

```
String: word
Integers:+1
java.lang.Object@3e25a5
```

We can fix it in the following way:

```
String concatenate(List<?> list) {
   String  result="";
   String separator="";
   if(list.size() >= 1 )
      if(list.get(0) instanceof String) {
         result="Strings:";
         separator=" ";
      }
      else if(list.get(0) instanceof Integer) {
         result="Integers:";
         separator="+";
      }
   for(Object el : list)
      result=result+separator+el.toString();
   return result;
}
```

But this requires to have at least one element in the list. Moreover, there is no guarantee that if the first element is, for example, a string, that this is not a list of Objects.

**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?

solution

If we introduce separate methods which differ only by the generic types of their arguments, we get compile-time errors such as:

```
Method concatenate(List<? extends Object>) has the same
erasure concatenate(List<E>) as another method in type C
```

This restriction is imposed to ensure that when choosing which of the overloaded method definitions to call, we always have a "best fit". Java class files do however include generic versions of the method signatures in the class (to enable separate compilation and type-checking of generic code). For this reason, it might seem surprising that we cannot disambiguate between these different overloaded methods, since at compile-time the type information is all available. However, Java also supports raw types - versions of generic classes in which no type parameter is provided (e.g., `List` for a `List<X>` class). These are supported for backwards compatibility with pre-generics Java code. For this reason, we need to consider the possibility that a client calling our method provides an argument of raw type `List`. In this case, we would not be able to choose between our different method overloads.

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

solution

The program is compiled and we obtain the expected results ("String: word", "Integers:+1", "..."), since in C# there is no type erasure and the information about generics is preserved at runtime.

## 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.

solution

We do not have any relation between the wildcard of `List`, and the types of the values that we are going to store.

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

solution

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

We have to use a lower bound constraint because we want the argument of `list.add` to be a supertype of `V`, otherwise we cannot pass it as a parameter.

**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?

solution

This method has exactly the same constraints of the ones obtained using a wildcard. In fact, the type of `value` can be a subtype of the type parameter of `list`, since it is a method argument. In practice, this means that the generic type of `list` is supertype of the type of `value`. For instance, consider the following program.

```
List<Object> list =...
add("x", list);
```

This program is accepted because strings are subtype of objects, thus `V=Object` is inferred by the type checker.

**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.

solution

```
List<String> list = new ArrayList();
List<Object> list2 = new ArrayList();
addAllX(list, list2);
addAllY(list, list2);
```

The call to `addAllX` is accepted by the compiler, while the one to `addAllY` is rejected, since it requires that the parametric type of `List` is exactly `String`. This happens because of invariance on type parameters in Java, so `V` has to be `String`, but the generic type of `list2` is `Object`.

## 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.

solution

```
class Cage{
   ...
   Animal<?> getAnimal() {
      if (animal!=null)  return animal;
      else{
         animal = new Sheep();
         Wolf wolf = new Wolf();
         wolf.lunchBag=wolf;
         return wolf;
      }
   }
}
class Zoo{
   ...
   void manage(){
      feedAnimal(new Cage(null));
}}
```

**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.

solution

```
class Fox extends Animal<Meat>{
   Fox(){}
   void eat(Meat m){}
   Wolf getLunchBox(){ cage.animal=new Sheep();return new Wolf(); }
   Cage cage;
}
class Zoo{
   ...
   void manage(){
      Fox fox = new Fox();
      Cage cage = new Cage(fox);
      fox.cage=cage;
      feedAnimal(cage);
   }
}
```

**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.

solution

Here we cannot make a sheep eat a wolf.

The reason is that `cage.animal` evaluates to the same value in both expressions `cage.animal` and `cage.animal.getLunchBox()` and so type safety is not broken and the `Sheep` can only be fed with `Grass`, which the `Wolf` is not.

**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.

solution

This is safe as no methods are called during the evaluation of arguments, so `cage.animal` cannot change.

**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.

solution

The version of `feedAnimal` in section D is unsafe as another thread might modify `Cage.animal` between the evaluation of the two expressions.
The version in section C is safe.

**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.

solution

We could allow wildcard capture to happen only once per access path in the same statement (rather than once per occurrence in the statement), if either

1. It includes no method calls and there are no method calls evaluated between any two instances of the path (only in the sequential case).
2. It begins with a local variable and follows only final fields (no method calls) - works also in the concurrent case.

## 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
```

solution

```
template <typename T>
class Doubling : public T {
public:
   virtual void setVal(int x) override {
      T::setVal(x * 2);
   }
}

template <typename T>
class Counting : public T {
public:
   virtual int value() override {
      ++numRead_;
      return T::value();
   }
   int numRead() { return numRead_; }
private:
   int numRead_{};
}
```

**B)** Describe how the instantiation above will look like.

solution

When the mix-ins are instantiated the following two classes will be generated:

```
class Counting_Cell : public Cell {
public:
   virtual int value() override {
      ++numRead_;
      return Cell::value();
   }
   int numRead() { return numRead_; }
private:
   int numRead_{};
}
```

```
class Doubling_Counting_Cell : public Counting_Cell {
public:
   virtual void setVal(int x) override {
      Counting_Cell::setVal(x * 2);
   }
}
```

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

solution

While this concept is similar to Scala traits there are some notable differences. In Scala it is possible to mix any number of traits in a class and use this in any location of the code that requires the same class and a subset of the traits:

```
var x = new X with A with B with C with D
var x1: (X) = x // OK
var x2: (X with A) = x // OK
var x3: (X with B) = x // OK
var x4: (X with A with D with C) = x // OK
```

Using the proposed solution in C++ however is more restrictive, as there is no way to refer to the class `X` with arbitrary mix-ins:

```
auto x = new D<C<B<A<X>>>>();
X* x1 = x; // OK
A<X>* x2 = x; // OK
B<X>* x3 = x; // Does not compile
C<D<A<X>>>* x4 = x; // Does not compile
```

This is particularly important for traits that introduce new methods like `Counting.numRead()` since any client code that uses this new behavior would have to know exactly how the trait was mixed-in.

Another problem of the C++ solution is object construction. If the base class does not have a default constructor then the mix-ins should know to call the correct constructor and provide appropriate parameters. An alternative here is for the mixin to just inherit the base class constructors: `using T::T;` which will allow clients of the mixin to use all constructor available in the base class. This works fine if the state of the mixin can be initialized with default values.

A further difference to Scala is that in the C++ solution it is possible to include the same "trait" more than once:

```
auto x = new Doubling<Doubling<X>>();
x->setVal(5);
x->value(); // returns 20
```

An advantage of the C++ solution is that we do not need to declare the base class that the mix-ins extend. Thus it is possible to use them with different base classes as long they have matching virtual methods.

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

solution

No, it is not possible to implement this with generics. The core reason for this is erasure. In Java each class must have a known supertype. However if we could translate the code above to Java and apply erasure, it will turn out that the supertype of `Doubling` and `Counting` is `Object` which is clearly not what we want.

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

solution

The code cannot be implemented using C# generics either - the standard explicitly forbids a generic class to inherit directly from a type argument. Although the type argument would be known at run-time and it is theoretically possible to allow inheriting from it, that would have complicated and slowed down the handling of generics by the C# virtual machine. The reason is that unlike C++, in C# only a single class would be generated for all possible type arguments and a lot of dynamic checks and method call adjustments would be required to make this work. Thus in this case the designers of C# chose safety and efficiency at the expense of expressiveness.