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?

Solution

- We obtain two errors:

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

Cannot perform instanceof check against parameterized type
```

```
List<String>. Use instead its raw form List since generic
type information will be erased at runtime.
```

This happens because of type erasure in Java.

- 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
```

- 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.

- If we introduce separate methods which differ only be 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.

- 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 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.

Solution

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

- ```
  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.

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

  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 generic type 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.

- ```
  List<String> list = new ArrayList();
  List<Object> list2 = new ArrayList();
  addAll(list, list2);
  addAll1(list, list2);
  ```

  The call to `addAll` is accepted by the compiler, while the one to `addAll1` 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 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?

Solution

Here is the C++ implementation of the mixin classes:

```
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_{};
}
```

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

```
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);
   }
}
```

- 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.

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.

- 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.
- 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.

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.

Solution

The only annotation that may be used here is + (covariant).

If we use contravariance, then the following code may break:

```
def foo(x:B[String])
   // due to contravariance x may be of type B[AnyRef]
{
```

```
    var a = new A[String]
    x.m(a) // crash!
       // x expects A[AnyRef] and gets A[String]
       // due to contravariance of A, this is
       // type incorrect
}
```

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?

Solution

Drawback: we cannot check the consistency of an object considering only the current instance.

```
public class Foo {
    int a=0; /// invariant a>=0
    public class Foo broken() {
       Foo result=new Foo();
       result.a=-1;
       return result;
    }
}
```

Advantage: we can access the internal structure of other objects of the same class. Note that we already know their internal structure, since it is exactly the same as that of the current object.

```
public class List {
    private Object el;
    private List next;

    public removeSecondElement() {this.next=this.next.next;}
}
```

Similarly to private fields, an object can access protected fields of other objects of the same type. However there are some restrictions. Consider the code below:

```
class A {
    protected int x;
}
class B : A {
    public void foo() { this.x = 2; } // OK
    public void bar(B b) { b.x = 2; } // OK
    public void foobar(A a) { a.x = 2; } // Does not compile
}
```

The `foo` method compiles without errors - an object has access to its own protected fields. `bar` also compiles - an object has access to private and protected fields of other objects of the *same* type. However, as we see from `foobar`, an object does not have access to the protected fields of another object of a supertype. If this were allowed it would be possible to break the internal state of an arbitrary class by simply inheriting from it:

```
class A {
    protected int x;
}
final class B {
    /// invariant x==0
    public B() { this.x = 0 };
```

```
}

class MaliciousClass extends A {
   public void foo(A a) {a.x = 42;}
   public void evil() {
      B b = new B();
      foo(b);
      // The invariant of b would now be broken if
      // method foo could be compiled.
   }
}
```

Note that in the examples above we assume that all classes are in different packages.

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?

Solution

- The code has a package access class `B` in the public interface. We would expect the com-

piler to complain about that. This could, however, prove useful, for example (disregarding the inheritance) if `C` was a `LinkedList` class and `D` a `ListNode` - we might want to let the client hold Nodes, but not be able to do anything with them except pass them back to the list. We would expect the compiler to reject all numbered lines except number 1, and perhaps also number 6 - as explained in the last section.

- C++ does not have access modifiers for classes, and private inheritance does not allow upcasting, so this problem does not exist - however, if we look at inner classes, C++ has access modifiers and we get:

```
class E{
public: class A{};
protected: class B : A{};
public: class C : B{
   public:
      B* get(){...}
      void set(B* b){...}

   private: B b;
   };
};

void f() {
   E::C* c = new E::C();
// E::B* b = c->get();       //rejected by compiler
// E::A* a1 = c->get();      //rejected by compiler
// E::A* a2 = c;             //rejected by compiler
   c->set(c->get());
// c->set(c);                //rejected by compiler
}
```

  C# requires all types mentioned in a method signature to have at least the visibility of the method (for access modifiers for classes they would have to be inner classes, as namespaces only support public members), and similarly for inheritance - so the problem is prevented

  Scala allows the `set` method but not the `get` method and not the inheritance - so `set` would only be callable from inside the package - not very useful

- We could require that:

  For inheritance, the superclass (and interfaces) must be at least as visible as the inheriting class

  For method signatures, all types mentioned in the signature must be at least as visible as the method being declared

- We could relax the inheritance rule, and allow an annotation of the sort `C extends* A` that would mean to the client that `C` can be assigned to `A` but does not mention `B` and does not state the length of the inheritance path. This could be useful, for example, if `B` was an abstract private class with some implemented methods, inherited by several public classes (that should be subtypes of `A`) that use its method implementations, but we didn't want to expose `B` as it is an implementation detail.