

Exercise 7

Parametric polymorphism

1)

a)

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

b)

```
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];}
}
```

c)

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

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

Limits of b: in Java, we have the same limits of a, 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 instance of `Integer`, the runtime environment would raise an exception and the element would not be added to our list. The advantage is that method `get` returns an `Integer`, thus we do not need dynamic casting of the values returned by this method.

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

Concepts of Object-Oriented Programming

- 2) 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 waits for A[AnyRef] and gets A[String]
           // due to contravariance of A, this is
           // type incorrect
}
```

- 3) Class P1 can be instantiated with any type, while P2 has to be instantiated with subtypes of A.

```
val x : P1[AnyRef]; //correct
```

```
val y : P2[AnyRef] //wrong: AnyRef is not a subtype of A
```

Furthermore, class P1 is covariant in its argument:

```
val x : P1[A]=new P1[B] //correct
```

```
val y : P2[A]=new P2[B] //wrong: found P2[B], required P2[A]
```

- 4)

- 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 non-invariance on type parameters in Java, so `V` has to be `String`, but the generic type of `list2` is `Object`.

5) The typechecker has to prove the following formula:

```
∃ T1 >: B. ∃ T2 <: B.
T2 <: T1 // list1.add(0, list2.get(0))
AND T2 <: B // return list2.get(0);
```

The hypotheses are generated from the method signature, while the right part is generated from the method body. This implication is validated since `T2 <: T1` because of the transitive property of (`<:` ), and `T2` directly from the hypothesis. Thus, the program is compiled.

6)

- 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:";
```

## Concepts of Object-Oriented Programming

```
 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 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.

- 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.

7) The Scala approach is completely unsafe. It does not check at all if an object respects a lower type, and anyway it's impossible to check it using the current bytecode instruction set.

For instance, the following example is normally executed:

```
class Foo[X >: String](val str : X)
val a=new Foo(1)
a.str
```

with the following output

```
defined class Foo
a: Foo[Any] = Foo@968f9
res7: Any = 1
```