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

Limits of b: in Java, we have the same limits of a), code duplication and additional type castings and checks in method add. In addition, we do not have behavioral subtyping, since method add in `IntList` may not respect the expected contracts in List if we invoke passing an object that is not instance of Integer (in which case we would have an exception and the element would not be added to our list). The advantage is that the method `get` returns an Integer object, thus we do not need dynamic casting of the values returned by this method.

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

2)

If we adopt a covariance annotation on T (Matrix[+T]), the compiler rejects the program with the following error:

```
error: covariant type T occurs in contravariant position in type T of
value elem
           def set(i : int, j : int, elem : T) : Unit = {
                                                ^
```

This happens because a covariance annotation is allowed only for types that do not occur in the types of parameters.

On the other hand, if we adopt a contravariance annotation (Matrix[-T]), the compiler still rejects the program, this time with the following error:

```
error: contravariant type T occurs in covariant position in type
(int,int)T of method get
          def get(i : int, j : int) : T  = {return m(i)(j);}
              ^
```

This happens because a covariance annotation is allowed only for types that do not occur in the return type.

The following assignments are rejected:

```
val z : Matrix[String, String, String]=
         new Matrix[Object, Object, Object](Array(Array(new Object())))
val y : Matrix[Object, Object, Object]=
         new Matrix[String, String, String] (Array(Array("foo")))
val x : Matrix[String, String, Object]=
         new Matrix[String, String, String] (Array(Array("foo")))
```

while the following ones are accepted

```
val y1 : Matrix[Object, Object, String]=
         new Matrix[Object, Object, Object] (Array(Array(new Object())))
val z1 : Matrix[String, Object, String]=
         new Matrix[String, String, String] (Array(Array("foo")))
```

In this way we can hide some information to the client (i.e. y1 can add only strings to a matrix of object, while z1 expects to receive Object values from method get).

3) We need to adopt a covariant annotation (class B[+T]).
In negative positions (arguments) we need contravariant types. Formally, B[T1] <: B[T2] => A[T1] :> A[T2] because A[T] is the type of an argument of B. Since T has covariant annotation, we have that B[T1]<:B[T2] ⇔ T1<:T2. T1<:T2 => A[T1] :> A[T2] is proved as type T in A has contravariant annotation. So we prove that B[T1]<:B[T2] => T1<:T2 => A[T1] :> A[T2]
If we had a contravariant annotation on generic type T of class B, we would have that B[T1]<:B[T2] ⇔ T1:>T2, but T1:>T2 => A[T1] :> A[T2] cannot be proven as type T has not covariant annotation.

The following example illustrates why we cannot have contravariant annotation of the generic type of B.

```
val x : A[String]=new A[String]
val y : B[String]=new B[Object] //we can do it since  B[-T]
y.m(x)
```

The method call is allowed by the compiler, as method m on B[String] is trivially defined for an argument of type A[String]. On the other hand, B[Object] expects as argument an object subtype of A[Object], but A[String] :> A[Object] since A has contravariant annotation on the generic type.

4)

- Class P1 can be instantiated with any type, while P2 has to be instantiated with subtypes of A.
- P1[T1] <: P1[T2] ⇔ T1<:T2, while P2 is invariant (P2[T1] <: P2[T2] ⇔ T1=T2).

Thus T<:A ⇔ P1[T] <: P1[A], while P2[T] <: P2[A] ⇔ T=A
Therefore, P1 is less restrictive than P2.

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

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

5)

- We do not have any relation between the wildcard of List, and the types of the value 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 constraints because we need that the argument of list.add is supertype of V, otherwise we cannot invoke it passing value.

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

This method has exactly the same constraints of the one written using a wildcard. In fact, the type of value can be a subtype of the generics of list, since it is are method arguments, i.e., that the generic of list is supertype of the type of value. For instance,

```
List<Object> list =…
add("x", list);
```
This program is accepted because `String` is a subtype of `Object`, 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 of type parameters in Java, so V has to be String, but the generic type of list2 is Object.

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 erasure in Java, i.e. information about generics is erased during the compilation and it cannot be used by dynamic checks

- 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
```

```
String concatenate(List<Object> 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! If we don't have such element, we cannot know at runtime the type of the objects that should be stored in the list, thus we cannot correctly initialize `result`.

- 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 type-checker has to prove the following:

∀T1, T2:
T1 >: B // List<? super B> list1
∧ T2 <: B // List<? extends B> list2
⇒
T2 <: T1 // list1.add(0, list2.get(0));
∧ T2 <: B // return list2.get(0);

This implication is easy to prove, since T2 <: T1 by the transitivity of <:, and T2 <: B directly from the hypothesis. Thus the compile-time checks are satisfied.

8) 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
```