Exercise 7

Parametric polymorphism

1)

- The right answer is 3. Answer 4 and 2 are wrong simply because `T` is covariant and not contravariant. Answer 1 is wrong since we can have a covariant type as return type of a method.

- The right answer is 3. In fact we cannot assign and object of type `A[B, C]` to a variable of static type `A[B, B]` since the second parameter type of `A` is contravariant and `C <: B`. All the other assignments are right.

2)

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 behavioral 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 reaise 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...

3) The following Java code will compile because of the covariance rule on arrays:

```
String[] s = new String[1];
Object[] o = s;
o[0]=new Object();
String s1=s[0];
```

but it will cause the following runtime exception:

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Object
```

while executing `o[0]=new Object()`.
If a language would adopt a controvariant approach to array, we could write the following code:

```
Object[] o = new Object[1];
String[] s = o;
o[0]=new Object();
String s1=s[0];
```

In this case, we expect that at runtime an exception is raised when `String s1=s[0]` is executed.
We can imagine that the choice of covariant array was done expecting that usually developers exploit this rule (that is, they assign/pass an array of type A[] to a variable/argument of type B[] where A <: B) only when they want to read the elements stored in the array and not to add new elements. This assumption could be reasonable in a scenario in which usually we pass arrays to procedures to perform some computations on them. A typical example can be:

```
class A {
    int i;
}
class B extends A{}

class Foo {
    static int add(A[] a) {
          int result=0;
          for(int i=0; i<a.length; i++)
                result +=a[i].i;
          return result;
    }
    public static void main(String[] args) {
          B[] b = ...;
          Foo.add(b);//allowed by covariant arrays
    }
}
```

Invariant arrays are statically safe and they do not require any runtime type check, but they are more restrictive.

4) We need to adopt a covariant annotation (`class B[+T]`).
In negative positions (arguments) we need contravariant types. Formally, we want that `B[T1] <: B[T2]` => `A[T1] :> A[T2]` because `A[T]` is the type of an argument of `B`, and `B` has covariant annotation. Since `T` has covariant annotation in `B`, we have that `B[T1]<:B[T2]` ⇔ `T1<:T2`. In addition, `T1<:T2` => `A[T1] :> A[T2]` holds as generic type `T` in `A` has controvariant 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.

5)

- 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`.

6) 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.

7)

- 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 overloadings.

- 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.

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
```