1. In the Java Collections Framework, `List` is an interface that is implemented by `ArrayList`. For each of the statements below, indicate if it is a valid statement with no compilation error. Explain why.

   (a) Problem #A

   ```
   1  void foo(List<?> list) { }
   2  // code omitted
   3  foo(new ArrayList<String>());
   ```

   (b) Problem #B

   ```
   1  void foo(List<Object> list) { }
   2  // code omitted
   3  foo(new ArrayList<String>());
   ```

   (c) Problem #C

   ```
   1  void foo(List<? super Integer> list) { }
   2  // code omitted
   3  foo(new List<Obejct>());
   ```

   (d) Problem #D

   ```
   1  void foo(List<? extends Object> list) { }
   2  // code omitted
   3  foo(new ArrayList<Object>());
   ```

   (e) Problem #D

   ```
   1  void foo(List<? super Integer> list) { }
   2  // code omitted
   3  foo(new ArrayList());
   ```

Suggested Guide:

(a) **Yes**, since `ArrayList<String> <: List<String> <: List<?>`

(b) **No**, as `ArrayList<String>` is not a subtype of `List<Object>`

(c) **No**, `List` is an interface. It would be fine if we changed it to `ArrayList<Object>` since

```
ArrayList<Object> <: List<Object> <: List<? super Object>
                  <: List<? super Integer>
```

(d) **Yes**, since

```
ArrayList<Object> <: ArrayList<? extends Object>
                  <: List<? extends Object>
```

(e) **Compiles, but with unchecked conversion warning**. The use of raw type should also be generally avoided.

2. The following static generic method `max3` that takes in an array of generic type `T` such that `T` implements the `Comparable` interface.

```
1  static <T extends Comparable<T>> T max3(T[] arr) {
2    T max = arr[0];
3    if (arr[1].compareTo(max) > 0) {
4      max = arr[1];
5    }
6    if (arr[2].compareTo(max) > 0) {
7      max = arr[2];
8    }
9    return max;
10 }
```

What happens if we replace the method header with each of the following:

(a) `static <T> Comparable<T> max3(Comparable<T>[] arr)`

> **Suggested Guide:**
> Realize that now the method returns a `Comparable<T>` object. If we change the type of `max` to `Comparable<T>`, then we are required to typecast in the argument of the `compareTo` method as it expects an argument of type `T` (*e.g.*, `arr[1].compareTo((T)max)`).

(b) `static <T> T max3(Comparable<T>[] arr)`

> **Suggested Guide:**
> The above preserves the return type as `T`. Since `arr[0]` has a type of `Comparable<T>` there is a type mismatch. An explicit typecasting is therefore required when assigning an element of `arr` to `max` (*e.g.*, `T max = (T) arr[0]`).

(c) `static Comparable max3(Comparable[] arr)`

> **Suggested Guide:**
> As `Comparable` is a generic interface, by not passing any type argument we have created a raw type. Indeed, this code fragment shows the effect of type erasure. When the compiler replaces the type-parameter information with the bound in the method declaration, it also inserts explicit cast operations in front of each method call to ensure that the returned value is of the type expected by the caller (*e.g.*, `(Integer) max3(new Integer[]{2,3,1})`).

3. Suppose a `Fruit` class implements `Comparable` interface, and an `Orange` is a subclass of `Fruit`. How would you change the `max3` method header in Question 2 such that the parameter type of `max3` is `List<T>` instead? You should aim to make the method as flexible as you can.

Suggested Guide:

Suppose we have:

```
1 class Fruit implements Comparable<Fruit> {
2   @Override
3   public int compareTo(Fruit f) { return 0; }
4 }
5 class Orange extends Fruit { }
```

Simply declaring the following:

```
1 static <T extends Comparable<T>> T max3(List<T> list)
```

would work for `List<Fruit>` only, but not for `List<Orange>`, since `Orange extends Comparable<Orange>` does not hold.

The first solution is to modify the argument:

```
1 static <T extends Comparable<T>> T max3(List<? extends T> list)
```

Now what can `T` be bound to? Can it be `Orange`? Notice that `<T extends Comparable<T>>` would not for `List<Orange>`, since the type constraint `Orange extends Comparable<Orange>` does not hold. Observe that we have the following type constraints:

`Orange <: Fruit <: Comparable<Fruit>`

but `Comparable<Fruit>` and `Comparable<Orange>` are ***invariant***.

As such, `Orange <: Comparable<Orange>` does not hold.

How about binding `T` to `Fruit`? Clearly, `Fruit extends Comparable<Fruit>` holds. And is `List<Orange>` a subtype of `List<? extends Fruit>`?

YES! This is a ***covariant*** relation. Thus, it is possible to put `List<Orange>` as an argument for `max3` in this case because `List<Orange> <: List<? extends Fruit>` and `Fruit extends Comparable<Fruit>`.

Another way is to declare `max3` as follows:

```
1 static <T extends Comparable<? super T>> T max3(List<T> list)
```

Now what can be `T` be bound to? Notice that we have the following type constraints:

```
Orange <: Fruit <: Comparable<Fruit>
       <: Comparable<? super Orange>
```

So `T` can be bound to `Orange`! In this case, the constraint `Comparable<Fruit> <: Comparable<? super Orange>` is ***contravariant***.

To be even more general, we should have:

```
1 static <T extends Comparable<? super T>>
2   T max3(List<? extends T> list)
```