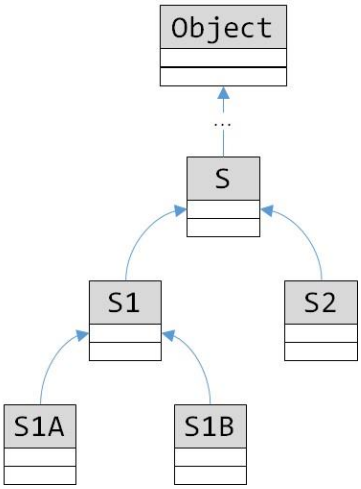
Problem Set 04 Worksheet

0 Exploration

Attempt the following on your own and try to explain the result.

0.a Bounded Wildcards

Consider the class diagram on the right. The ... in between `S` and `Object` basically means that we do not know if there are other superclass of `S` in between. Ignoring the ... part, can you write a code corresponding to that class diagram? Make sure that everything is a class and not interface or abstract class.



0.a.1 Upper Bound Wildcards

Consider the type `Array<? extends S1>`. Select ALL types that can be used to substitute `?` in the type.

Object	S	S1	S2	S1A	S1B

0.a.2 Lower Bound Wildcards

Consider the type `Array<? super S1B>`. Select ALL types that can be used to substitute `?` in the type.

Object	S	S1	S2	S1A	S1B

0.a.3 Container

So far our discussions on complex type `C<T>` focuses on the inner type `T`. Now we are going to look at the container type `C`. Recap that `ArrayList <: List`. Try running the following code:

```
public static <T> void f(List<T> lst) { }
public static <T> void g(ArrayList<T> lst) {
  f(lst);
}
```

Does it compile? What can you say about the relationship between `ArrayList<T>` and `List<T>`?

0.b PECS

If you have not done so, please attempt `0.a` from `R03_Worksheet`.

1 Question 1

There are too many classes to consider. So we will discuss quite a rather general approach.

```
List<Integer>   List<?>                  List<? super Number>     List<? extends Object>
List<Object>    List<? super Integer>    List<? extends Number>   ArrayList<Integer>
List<Number>    List<? extends Integer>  List<? super Object>     ArrayList<Object>
                                                                  ArrayList<Number>
```

There are two general approaches:

1.a Top-Down Approach

1. Find all the classes that should be at the top of the hierarchy (*i.e., no superclass from the choice*).
2. Write this down at the top of the diagram and remove them from the choice.
   - If there is a subtyping relationship with a class above, draw a line from this class to the other class.
3. Move to the next line below, repeat step (1) until there are no more classes to consider.

1.a.1 Question

What are the classes at the top of the hierarchy?

1.b Top-Down Approach

1. Find all the classes that should be at the bottom of the hierarchy (*i.e., no subclass from the choice*).
2. Write this down at the bottom of the diagram and remove them from the choice.
   - If there is a subtyping relationship with a class blow, draw a line from the other class to this class.
3. Move to the next line above, repeat step (1) until there are no more classes to consider.

1.b.1 Question

What are the classes at the bottom of the hierarchy?

2 Question 2

The code is reproduced below

```
static <T extends Comparable<T>> T max(List<T> list) {
  T max = list.get(0);
  if (list.get(1).compareTo(max) > 0) {
    return list.get(1);
  }
  return max;
}
```

The classes are also reproduced below.

```
class Fruit implements Comparable<Fruit> {
  public int compareTo(Fruit f) {
    return 0; // stub
  }
}
class Apple extends Fruit {
}
```

Draw the class diagram involving `Fruit`, `Apple`, and `Comparable`. Additionally, write a subtyping relationship of the form `S <: T`.

2.a Fruit f = max(apples);

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	
Type Parameter	
Resolved Type:	

2.b Type Inference Failure

2.b.1 Fruit f = max(apples)

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	
Type Parameter	
Resolved Type:	

2.b.2 Apple a = max(apples)

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	
Type Parameter	
Resolved Type:	

2.b.3 Apple a = max(fruits)

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	
Type Parameter	
Resolved Type:	

2.c PECS

We are trying to derive the answer without using PECS. If you are interested in using PECS, simply ask yourself the question which part is the consumer and which part is the producer. Then map that to a type and change it to either `extends` or `super`.

In the questions below, we cannot change `Fruit` or `Apple`. For obvious reason, we can implement `Comparable<Apple>` in Apple to solve some of the problems below.

Consider the statement `Fruit f = max(apples)`. Using subtyping relationship already known:

- `Apple <: Fruit`
- `Fruit <: Comparable<Fruit>`

We can solve the problem in this statement if we can simply add `Apple <: Comparable<Apple>`. Using the subtyping above, suggest a change to add the subtyping relationship needed without actually implementing `Comparable<Apple>`.

Do the same for all other statements. Can you solve them and make the necessary change?

2.d Another Type Inference

Use the modified code. Redo the type inference.

2.d.1 Fruit f = max(apples)

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	

Type Parameter	
Resolved Type:	

2.d.2 Apple a = max(apples)

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	
Type Parameter	
Resolved Type:	

2.d.3 Apple a = max(fruits)

Write down all type constraints based on the following and write down the resolved type:

Target Typing	
Argument Typing	
Type Parameter	
Resolved Type:	

Can you make it even more general?