Elements of Programming Languages
Tutorial 6: Classes, subtyping, and comprehensions
Solution notes

Starred exercises are more challenging. Please try all unstarred exercises before the tutorial meeting.

1. **Covariant and contravariant type parameters** Notice that the Box classes have no content — they are just to demonstrate covariance and contravariance.

$A =$	Any	Nothing	Super	Sub1	Sub2
g1(new Box1[A]))	Error	OK	OK	OK	OK
g2(new Box1[A]))	Error	OK	Err	OK	Error
h1(new Box2[A]))	OK	Error	OK	Error	Error
h2(new Box2[A]))	OK	Error	OK	OK	Error

   The OK cases are those where the subtyping relationship holds. The Error cases are those where the relationship doesn't hold. It may also be helpful to draw a simple lattice diagram (i.e. a tree with Super at the top and Sub1 and Sub2 as children) and show the subsets of the tree corresponding to the types that are valid for each call.

2. **Parameterized traits** The trait should look something like this:

```
trait Ordered[T] {
  def compare(that: T): Int
  def < (that: T): Boolean = this.compare(that) < 0
  def <= (that: T): Boolean = this.compare(that) <= 0
  def equalTo (that: T): Boolean = this.compare(that) == 0
  def nequalTo (that: T): Boolean = this.compare(that) != 0
  def > (that: T): Boolean = this.compare(that) > 0
  def >= (that: T): Boolean = this.compare(that) >= 0
}
```

3. **List comprehensions**

   (a)

```
  Result = List(2,3,4)
  List(1,2,3).map{x => x + 1}
// or equivalently
  List(1,2,3).flatMap{x => List(x + 1)}
```

   (b)

```
  Result = List(1)
  List(1,2,3).filter{x => x % 2 == 0}.map{x => x / 2}
// or equivalently
  List(1,2,3).flatMap{x => if (x % 2 == 0) {List(x/2)} else {Nil}}
```

   (c) ($\star$)

```
  Result = List((1,2),(1,3), (2,3))
  List(1,2,3).flatMap{x => List(1,2,3).filter{y => x < y}.map{y => (x,y)}}
// or
```

```
List(1,2,3).flatMap{x => List(1,2,3).flatMap{y =>
   if (x < y) {List((x,y))} else {Nil} }}
```

4. (⋆) **Covariant lists**

   (a) Something like

```
  Cons(1,Cons("abc",Nil))
```

   (b) Something like this:

```
abstract class List[+A] {
 def append[C >: A, B <: C] (m:List[B]):List[C]
}
case object Nil extends List[Nothing] {
 def append[C >: Nothing, B <: C] (m:List[B]): List[C] = m
}
case class Cons[+A](head: A, tail: List[A]) extends List[A] {
 def append[C>: A,B<:C](m: List[B]): List[C] = Cons[C](head, tail.append[C,B](m))
}
```

   This is a little tricky. For the Nil case, we need to say that `C >: Nothing` since `A` has been instantiated to `Nothing`. For the `Cons` case, the type parameters on `append` seem necessary to make the typechecker happy. The type parameters on `Cons` are not necessary, but included to help clarify what is going on.

   The supertype and subtype bounds are both necessary; if we remove `C >: A` then we don't know that we can put `A`'s into the result list and if we remove `B <: C` then we don't know that we can put `B`'s into the result list.