CSCI 2113.30 Lab 12

May 1, 2017

Goals

1. Learn about variance
2. Implement List Data structure in scala
3. Practice using List

Activity

1. Study the following code:

```
class Dog
class Corgi extends Dog
val corgi = new Corgi
def feed(dog: Dog) = ...
feed(corgi)
```

   Notice how the feed function takes Dog as its parameter but we can apply the function with an instance of a Corgi class.

   Based on this observation, we can derive a simple rule, if a function takes an parameter of type S, it is possible to apply the function with an argument with type T where T is a subtype of S (T <: S).

   This rule has a name: The Liskov Substitution Principle, which is bit more general than our rule. It states:

   > *If A <: B, then everything one can do with a value of type B one should also be able to do with a value of type A.*

2. Now study the following function:

```
def feed(dogs: List[Dog]) = ...
val corgis: List[Corgi] = List(...)
feed(corgis)
```

   Is the rule from 1 applicable in this case as well? In other words, is List[Corgi] a subtype of List[Dog]?

   Intuitively, this makes sense. List of Corgis is a special case of List of arbitrary dogs. But is it always the case when T <: S then List[T] < List[S] or in more general terms C[T] <: C[S]?

3. Imagine that the List class is mutable like in Java. Rather, since Array is mutable, let's consider an Array and let's explore if Array[Corgi] <: Array[Dog]. Consider the following code:

```
class Poodle extends Dog
val corgis: Array[Corgi] = Array(new Corgi, new Corgi)
val dogs: Array[Dog] = corgis
dogs(0) = new Poodle
val c: Corgi = corgis(0)
```

   What is the type of corgis? What is the value of corgis(0)? Does this make sense to store it as a Corgi? Can you see what went wrong?

   It seems like Array[T] <: Array[S] relationship doesn't hold even though T <: S.

4. We'll derive our second rules of types: If T <: S then C[T] < C[S], only if C is immutable.

5. There are actually three different possibilities for type that are parameterized with generic.

   If C[T] is a parameterized type and A, B are types such that A <: B, then there are three possible relationships between C[A] and C[B]:

Relationship	Name
C[A] <: C[B]	C is covariant
C[A] >: A[B]	C is contravariant
Neither	C is invariant

6. In Scala you can declare your variance of a type by annotating the type parameter:

```
class C[+A] { ... } C is covariant
class C[-A] { ... } C is contravariant
class C[A]  { ... } C is invariant
```

7. Now consider following two function types:

```
type A = Dog => Corgi
type B = Corgi => Dog
```

   According to the Liskov Substitution Principle, which of the following should be true?

   a. A <: B
   b. B <: A
   c. A and B are unrelated

   Try to create mock-up function that has the above types and see what makes sense.

8. We can generalize the observation from #7 and come up with a new rule:

   If `A2 <: A1` and `B1 <: B2`, then

   ```
   A1 => B1 <: A2 => B2
   ```

   Know this!

   Or expressed as differently:

   ```
   Function[A1, B1] <: Function[A2, B2]
   ```

9. This means that functions are contravariant in their argument types and covariant in their return type. So in other words:

   Covariant type parameters can only appear as a function return type
   Contravariant type parameters can only appear as function parameters
   Invariant type parameters can appear anywhere

10. Try the following code:

   ```
   trait List[A]
   case object Nil extends List[Nothing]
   case class Cons[A](head: A, tail: List[A])
   val list: List[String] = Nil
   ```

   You will get an error that looks something like this:

   ```
   Note: Nothing <: String (and Nil.type <: List[Nothing]), but trait List
   is invariant in type A.
   You may wish to define A as +A instead. (SLS 4.5)
          val list: List[String] = Nil
                                   ^
   ```

   It's saying `Nothing <: String` true but the thing that you are trying to do is impossible because List is invariant.

   We can now fix this by saying List should be covariant:

   ```
   trait List[+A]
   ```

11. There's one more thing to consider. Take a look at the following code:

   ```
   trait List[+A] {
     def prepend(elem: A): List[A] = Cons(elem, this)
   }
   ```

   This does not compile. Why? Think back to the Liskov Substitution Principle.

12. If above code is legal then we can do the following:

```
        val dogs: List[Dog] = List(new Corgi)
        val corgis: List[Corgi] = List(new Corgi)

        // If we can do this with List[Dog]
        dogs.prepend(new Poodle)

        // We should be able to do the same with List[Corgi] but this is an
error!
        corgis.prepend(new Poodle)
```

Another way to think about this is that the function arguments must be contravariant, but we declared A to be covariant. Therefore, error is produced.

13. But we still want to prepend! We can correct this by using lower bounds:

```
        def prepend[B >: A](elem: B): List[B] = new Cons(elem, this)
```

In other words, we can append an element that's more generic than current type. The rule is:

> **Covariant** type parameters may appear in lower bounds of method type parameters.
> **Contravariant** type parameters may appear in upper bounds of method

14. Finally take a look at the following code:

```
        val corgis: List[Corgi] = List(new Corgi)
        val poodle: Poodle = new Poodle
        val result = corgis.append(poodle)
```

What is the type of result? Why do you think it is?

Assignment

1. Download Lab12.scala
2. Implement all the List functions as methods. You may use the lecture slides as your reference. Most of the functions were already implemented during the lecture.
3. Download Amazon.scala
4. Implement the 10 functions defined in the file. The main method has a simple test but you should write more test cases to make sure your functions are correct.
5. Submit the two scala files.