Lecture 9

1 Reading

Programming in Scala, Chapters 12 and 19.

2 The Singleton Pattern

At times, it is often desirable to only have a single instance of a particular class. For example, if you have a class that lets you read input from the user, it doesn’t make sense to have two instances, since the user just has one keyboard. You can create a singleton object in Java by following the design pattern in fig. 18.1. The only instance of Singleton that can exist is the one stored in the private field. (Notice that the constructor is private.) The only way to get the singleton is by calling the Singleton.getInstance method.

Here is the equivalent in Scala:

```scala
object Singleton {
  def myMethod() = println("It works")
}
```

So, all the top-level functions we’ve created so far (in class and in assignment) can be thought of as methods of singleton objects.

3 Case Objects

We’ve seen several examples of case classes that take no arguments. For example, to represent binary trees with values at nodes (and not at the leaves), we could write the following type:

```scala
sealed trait BinTree
case class BinTree(lhs: BinTree, value: Int, rhs: BinTree) extends BinTree
case class Leaf() extends BinTree
```

However, we can also represent leaves using a case object, which is essentially a singleton that can be used in pattern-matching:

```scala
sealed trait BinTree
case class BinTree(lhs: BinTree, value: Int, rhs: BinTree) extends BinTree
case object Leaf extends BinTree
```

Since Leaf is an object and not a class, it cannot take any arguments, which why we can simply write Leaf instead of Leaf(). Scala uses case objects extensively. For example, Nil and None are case objects. If they were case classes, we’d have to write Nil() and None() instead.
public class Singleton {
    private static Singleton instance = null;

    private Singleton() { }

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new ClassicSingleton();
        }
        return instance;
    }

    void myMethod() {
        System.out.println("It works");
    }
}

Figure 18.1: The Singleton Pattern in Java

4 Subtyping

Subtyping in Scala is very similar to subtyping in Java.

**Notation** We write $A <: B$ to mean $A$ is a subtype of $B$.

The intuition behind subtyping is that a subtype always “adds more features” to its super-type, and doesn’t “subtract features”. Therefore, if $A <: B$, then $A$ can be used in any context where $B$ is expected. The context simply won’t try to use the extra features of $A$. However, $B$ cannot be used in contexts where $A$ is expected, because the context may rely on the “added features” of $A$ that $B$ does not provide.

For example, consider the following hierarchy of types, all of which implement the Animal trait:

```scala
trait Animal {
    def sound(): String
}

class Dog extends Animal {
    def sound() = "woof"
    def bite() = "ouch"
}

class Poodle extends Dog {
    override def sound() = "yelp"
}

class Cat extends Animal {
    def sound() = "purr"
    def scratch() = "yow"
}
```

A function that expects an Animal can be applied to an object of any class defined above. Similarly, a function that expects a Dog can be applied to a Dog or a Poodle, but not to a

---

1Think of a trait as an interface in Java. Traits are actually more flexible, but we are only going to use them like interfaces for now.
def useDog(c: Single[Dog]) = {
    c.get().bite()
}

useDog(new Single(new Cat()))

(a) Cats can't bite.

def useAnimal(c: Single[Animal]) = {
    c.get().sound()
}

val c = new Single[Dog](new Dog())
useAnimal(c)

(b) Scala is being conservative.

Figure 18.2: None of these programs type-check.

def useAnimal2(c: Single[Animal]) = {
    c.set(new Cat())
    c.get().sound()
}

val c = new Single[Dog](new Dog())
useAnimal2(c)

(c) Sneaky function confuses our code.

Cat. For example, the following function takes dogs and makes them bite:

def dontBite(x: Dog) = {
    x.bite()
}

The expression dontBite(new Cat()) will not type-check, which is good, because cats don’t have a bite method.

Given the traits and types defined above, we can say that:

- Dog <: Animal because Dog extends Animal
- Poodle <: Dog because Poodle extends Dog
- Cat <: Animal because Cat extends Animal
- Poodle <: Animal because Poodle <: Dog and Dog <: Animal (subtyping is transitive),
- X <: X for all X (subtyping is reflexive).

In addition, scala has two special types:

- X <: Any for all types X.
- Nothing <: X for all types X.

A peculiar property of Nothing is that there are no values with type Nothing! It may seem pointless to have a type with no values. For example, the following function cannot be applied to anything (not even to null):

def useless(x: Nothing): Unit = {
    println("Cannot call me")
}

But, we’ll see why Nothing is useful in a moment.

5 Generics and Subtyping

Subtyping becomes more complicated when we working with generics types, such as lists, sets, or any other kind of container. To illustrate, we’ll work with the following generic type, which is the simplest possible container:
Unsurprisingly, we can’t store cats in \( c1 \). If we did, a function that consumes a container with a dog, might try to \( .get() \) the dog and call the \( .\text{bite}() \) method, which cats don’t have. Therefore, the code in fig. 18.2a does not type-check. However, since \( \text{Dog} <: \text{Animal} \), can we send a dog-container to a function that expects an animal container. For example, section 5 appears to be safe. Unfortunately, this code does not type-check either. Although this particular example is safe, consider the variation in section 5.

The problem above is that useAnimal2 sneakily stores a cat in the container. After the function returns, \( .get() \) would produce a cat even though the type indicates that it should produce a dog. Therefore, the Scala type-checker does not allow this program to type-check.

Unfortunately, section 5 (which was safe) does not type-check either, just so that this kind of unsafe example can be ruled out. Individual methods and functions are type-checked only once. Similarly, when a function or method call is type-checked, the body of the function is not re-examined.

## 6 Variance

However, lists and other immutable data structures in Scala are not constrained this way. For example, the following code does type-check:

```scala
def useAnimals(alist: List[Animal]) = {
  alist.map(animal => animal.sound()).mkString(",
"")
}
val lst: List[Cat] = List(new Cat(), new Cat())
useAnimals(lst)
```

The reason this works is because there are no methods on lists to update their contents. The problem with our \( \text{Container}[T] \) class is that it writes to \( T \)-typed values. However, if we had a functional container class, we can use a \textit{covariance annotation} to indicate that \( T \)-typed values are never updated.

```scala
class Container[+T](private val x: T) {
  def get(): T = x
}
```

The \( +T \) annotation indicates \( \text{Container}[A] <: \text{Container}[B] \) when \( A <: B \). For this to be safe, Scala ensures that \( T \)-typed values are never updated by the class.\(^2\) (The class may have other kind of state, but it can’t update \( T \)-s.)

As a rule of thumb, almost any immutable data structure can be made covariant, which makes it more flexible. E.g., we can a Set[Cat] where a Set[Animal] is expected or a List[Dog] where a List[Animal] is expected.

In fact, the list type in Scala is covariant:

\(^2\)Covariance is actually more subtle than this. The reading discusses it in more depth, but you don’t have to remember all the details.
sealed trait List[A]

Cons is easy to define as follows:

case class ::[A](head: A, tail: List[A]) extends List[A]

However, the definition of Nil is trickier. If it were a case-class, we could write:

case class Nil[A]() extends List[A]

However, recall that Nil is a case-object. So, we may try to write this:

case object Nil[A] extends List[A]

However, objects can’t have type parameters (or any parameters for that matter).

As discussed earlier, Scala has a special type Nothing that has no values and is the subtype of all other types: i.e., Nothing <: A. Therefore, by covariance of lists, List[Nil] <: List[A], so we can write:

case object Nil extends List[Nothing]

Although there a no values of type Nothing, an empty list of of type List[A] doesn’t contain any values of type A. Therefore, the Nil object can be given the type List[Nothing].