Scala, how it manages to support multiple inheritance in an easier-to-understand way, without the headaches incurred by C++’s technique, adding complexity only when you need to express complex delegation sequences. In order to understand multiple inheritance in Scala, though, we first have to talk about a feature called traits, since Scala does not permit a class to inherit from more than one other class (we will see that only traits are multiple-inheritable).

Traits

Traits in Scala are best described as “interfaces that can provide concrete members.” Read more about [traits in scala](https://www.safaribooksonline.com/library/view/scala-in-action/9781935182757/) in Scala in Action. First, let’s look at the “interfaces” part. Traits can have abstract members (both fields and methods), so you can use them like you use interfaces in Java. In fact, Scala offers traits instead of interfaces (with all-method pure-abstract traits and Java interfaces being bytecode-compatible for interoperability). However, traits, unlike interfaces in Java, can provide concrete members. This is a pretty major difference, so let’s take a minute to think about the implications of allowing interfaces to provide concrete members

Solving the Diamond Problem

Scala’s solution to the Diamond Problem is actually fairly simple: it considers the order in which traits are inherited. If there are multiple implementors of a given member, the implementation in the supertype that is furthest to the right (in the list of supertypes) “wins.” Of course, the body of the class or trait doing the inheriting is further to the right than the entire list of supertypes, so it “wins” all conflicts, should it provide an overriding implementation for a member.

Let’s see Scala’s multiple inheritance in action with a classic example:

trait Drawable {

def draw() { }

}

trait Cowboy extends Drawable {

override def draw() { println("I am a cowboy!") }

}

trait Artist extends Drawable {

override def draw() { println("A pretty painting") }

}

// Note: the "with" keyword goes between type names for multiple inheritance

// You can chain on multiple "with X" clauses to mix in more traits. If a

// class is to be part of the extended type, it must be specified \_before\_

// the first "with X" clause.

class CowboyArtist extends Cowboy with Artist // Note: the "with" keyword goes between type names for multiple inheritance.

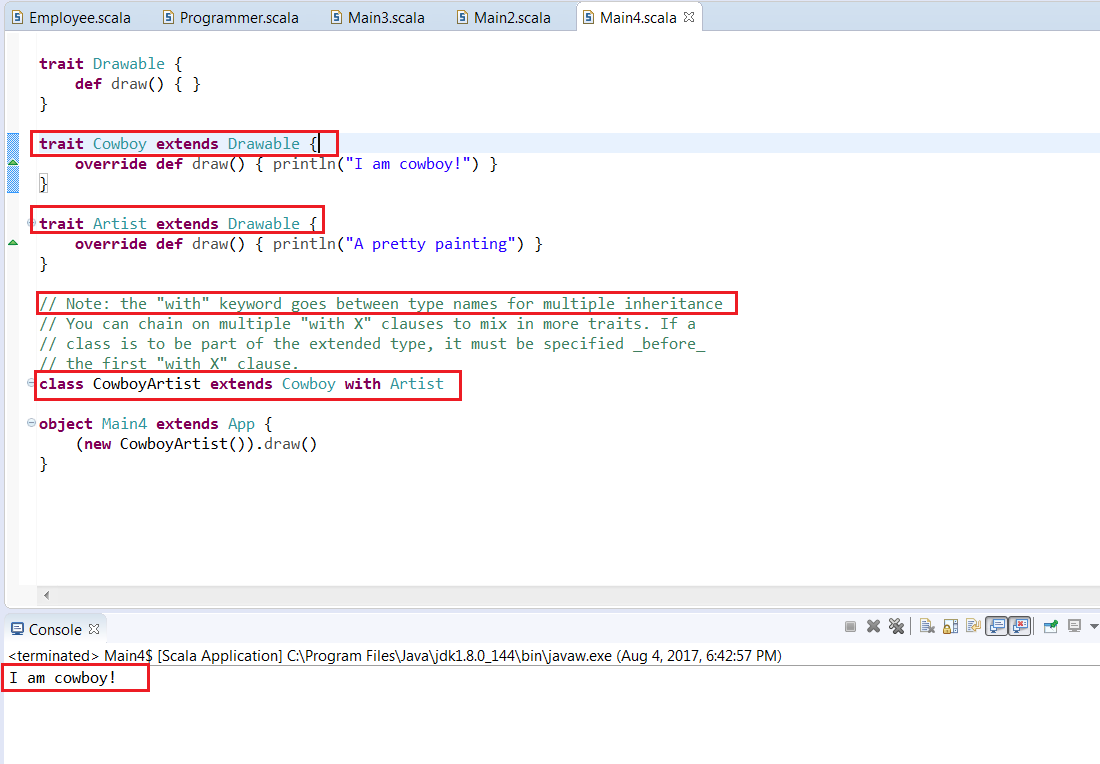
object Main extends App {

(new CowboyArtist()).draw()

}

When executed, this code will print “A pretty painting!,” since a CowboyArtist mixes in Artist after it mixes in Cowboy.

In the above example cowboyArtist class has implemented both cowboy and Artist traits. And both traits implemented draw() method of drawable trait. So multiple inheritance occurs here.



If there are multiple implementors of a given member, the implementation in the supertype that is furthest to the right (in the list of supertypes) “wins.” In scala thus overcome the multiple inheritance problem.

## Delegating to Super

In classes that inherit, it is common to delegate to members of super. In Scala, traits can also delegate to super. This leads one to wonder what type of object super actually refers to within a trait. At compile-time, this is easy to determine: it is whatever type the trait extends (or AnyRef if it does not specify a type to extend). As is often the case, though, super’s runtime type is often a subtype of its compile-time type.

As a rule of thumb, the runtime type for super can be found by looking “left” in the supertype list of the instantiated (leaf) class. To be sure of super’s exact runtime type, though, we have to apply a process called “type linearization.” In performing a linearization, you start with the type of the instantiated (leaf) class and recursively expand each of its supertypes into a list of their supertypes (the resulting list should be flat, not nested). You then scan that left-to-right and, whenever you come across a type that you’ve already seen, you delete it from the list.

Example 2:

