Scala III

Muhammad Tahir

Lecture 13

Contents

Traits

- Traits are a fundamental code reuse block in Scala
- A trait encapsulates method and field definitions
- A class can mix-in any number of traits
- Traits are commonly used for
  - widening thin interfaces to rich interfaces
  - defining stackable modifications

Traits and Multiple Inheritance

- Traits allow to inherit from multiple class-like constructs
- One important difference between Traits and multiple inheritance: *The interpretation of* `super`
  - With multiple inheritance, the method called by a super call can be determined right where the call appears
  - With traits, the method called is determined by a linearization of the classes and traits that are mixed into a class.
- When you call a method on a class with mixins, the method in the trait furthest to the right is called first

When to Use Traits

When implementing a behavior and

- the behavior is not going to be reused, make it a concrete class
- the behavior may be used by multiple different (possibly unrelated) classes, make it a trait
- the behavior will be inherited by other classes, make it an abstract class

Linearization

- Scala avoids multiple inheritance by using a technique called linearization
- Linearization flattens calls to super classes
- Linearization solves the diamond problem resulting from multiple inheritance

Linearization Cont'd

```
class BaseClass{
  def print{ println("Base Class") }
}

trait Trait1 extends BaseClass{
    override def print(){ println("Trait 1")
      super.print
    }
}

trait Trait2 extends BaseClass{
    override def print(){ println("Trait 2")
      super.print
    }
}

trait Trait3 extends Trait1{
    override def print(){ println("Trait 3")
      super.print
    }
}

// Derived class extending base class and mixins with traits
class DerivedClass extends BaseClass with Trait2 with Trait3{
    override def print(){ println("Derived Class")
      super.print
    }
}

val printFun = new DerivedClass
printFun.print
```

Linearization Cont'd

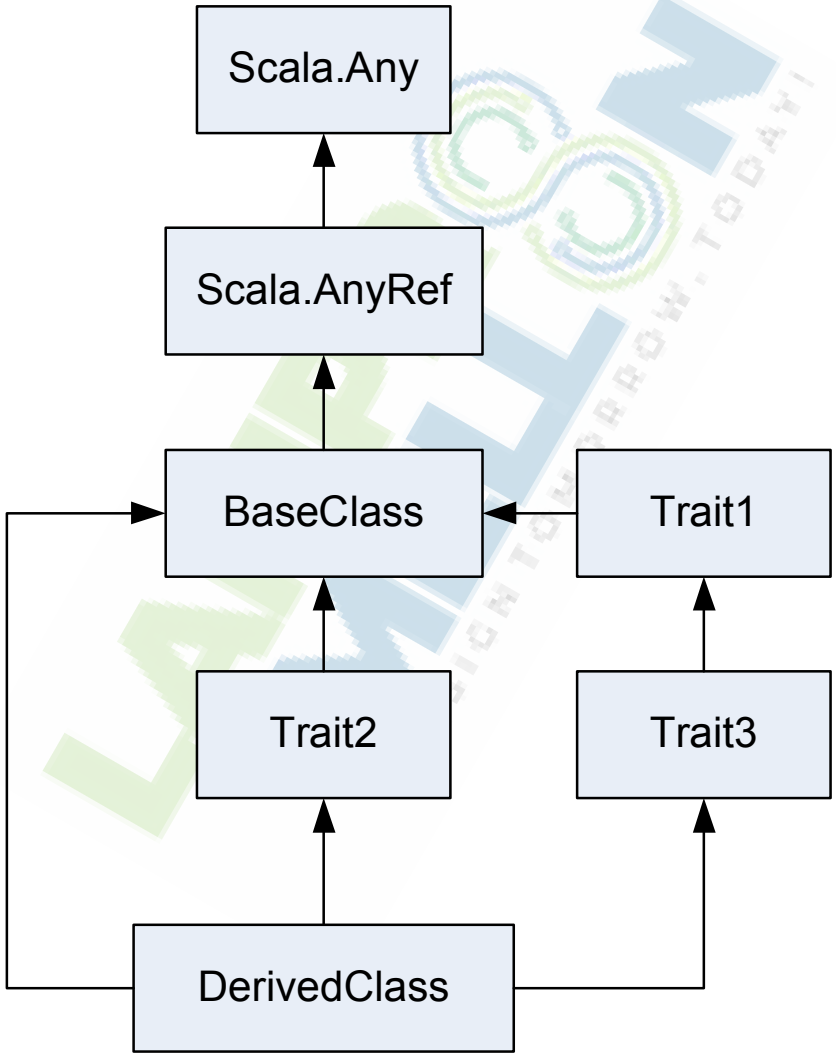

Figure: Inheritance structure.

Linearization Cont'd

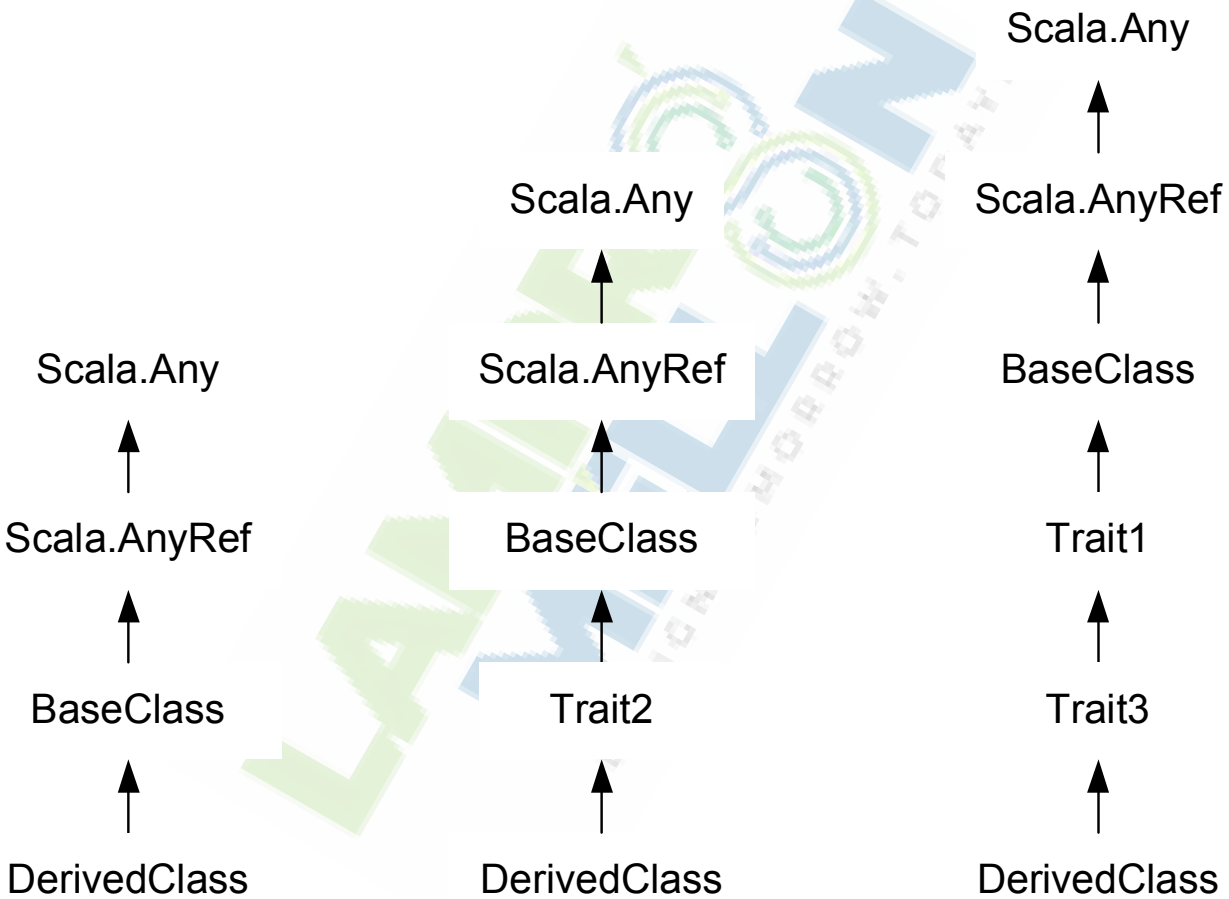

Figure: Linearization Step1.

Linearization Cont'd

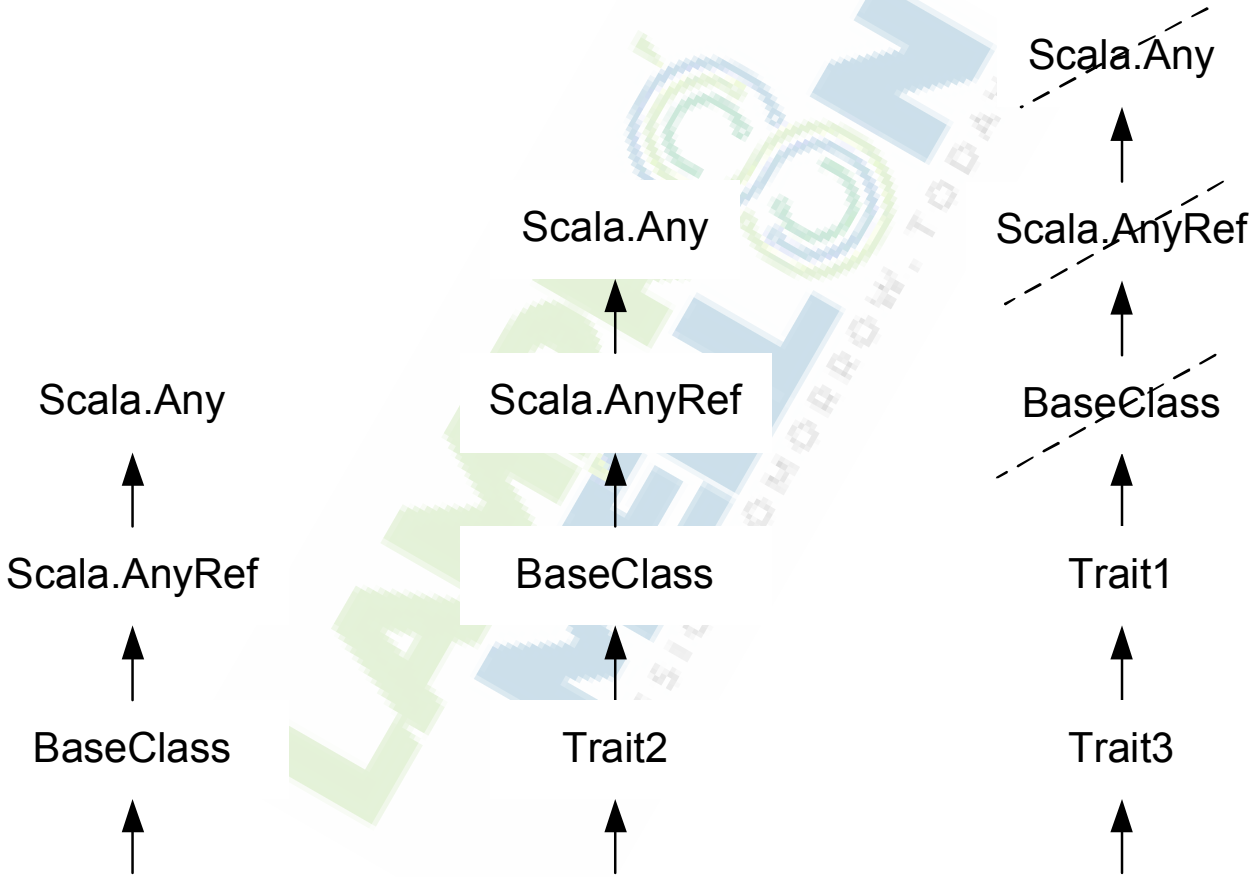

Figure: Linearization Step2.

Linearization Cont'd

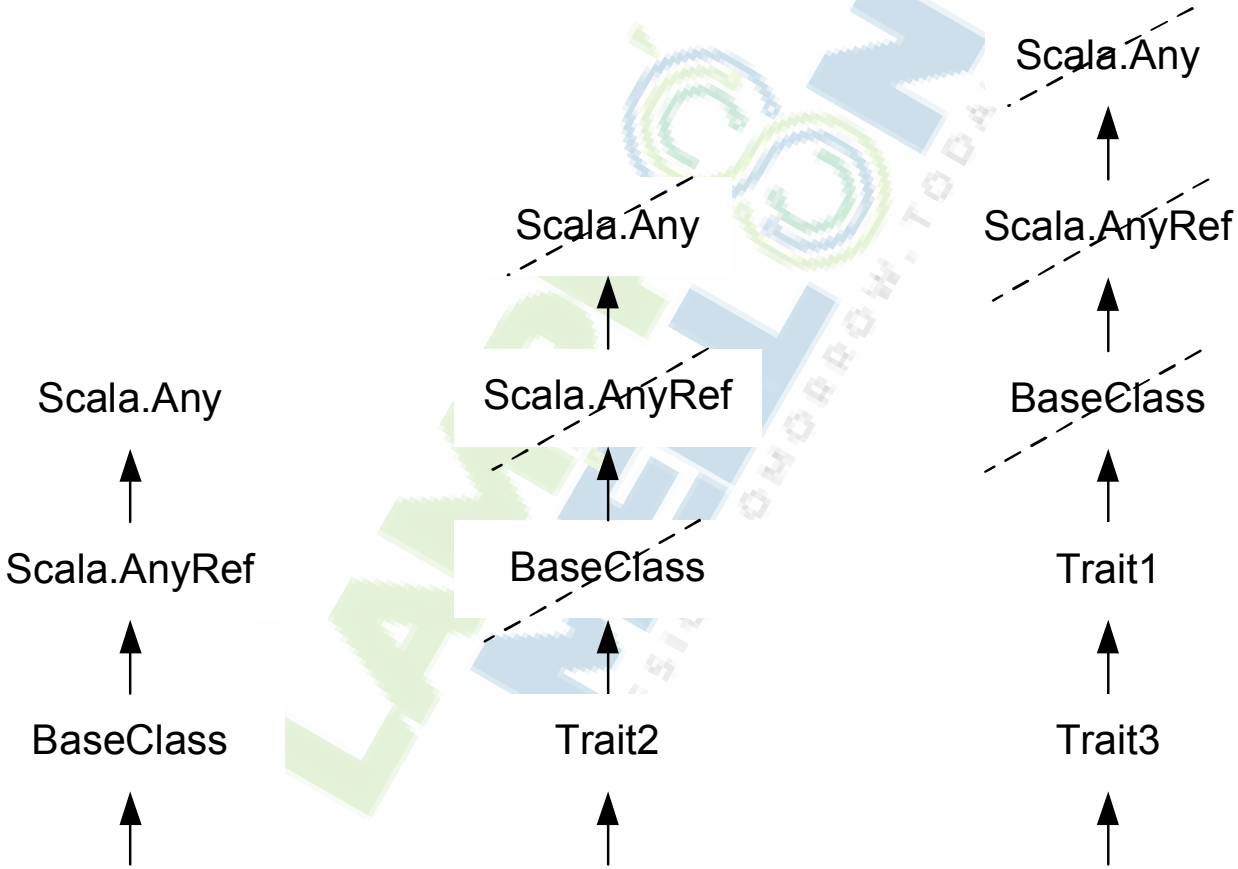

Figure: Linearization Step3.

Linearization Cont'd

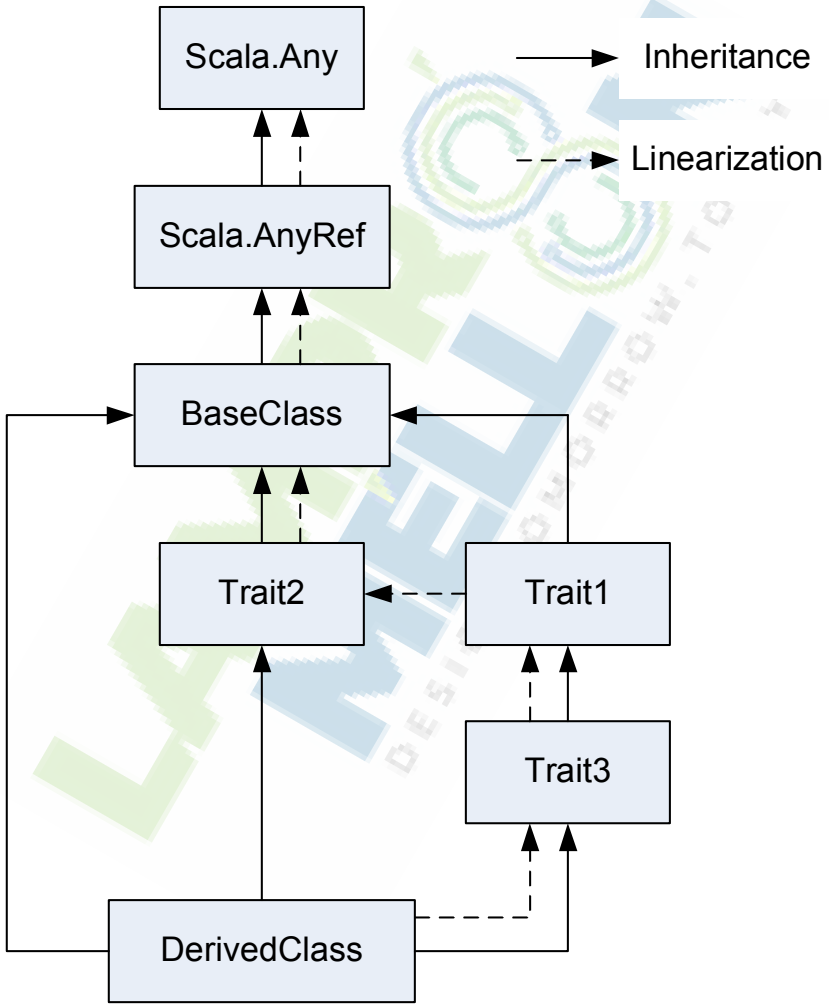

Figure: Inheritance and linearization.

Linearization Cont'd

```
// Derived class extending base class and mixins with traits
class DerivedClass extends BaseClass with Trait2 with Trait3{
      override def print(){
            println("Derived Class")
            super.print
      }
}

// The output at the terminal
Derived Class
Trait 3
Trait 1
Trait 2
Base Class

// Derived class extending base class and mixins with traits
class DerivedClass extends BaseClass with Trait3 with Trait2{
      override def print(){
            println("Derived Class")
            super.print
      }
}

// The output at the terminal
Derived Class
Trait 2
Trait 3
Trait 1
Base Class
```

Partial Functions

- Partial functions are partial implementations and are also termed as *unary* functions
- Do not evaluate the function for every possible value of input parameters
- In Scala, partial functions can be defined by using `case` statement
- Method `orElse` allows chaining other partial function(s)

Partial Functions Cont'd

```
trait PartialFunction[-A, +B] extends (A) => B
```

The `(A) => B` can be interpreted as a function that transforms (a functional mapping) type A input to type B result

Partial Function Example

- Implementing square-root as partial function

```
val squareRoot: PartialFunction[Double, Double] = {
    case d: Double if d > 0 => Math.sqrt(d)
}
```

- Output with an example list

```
val list: List[Double] = List(4, 16, 25, -9)
val result = list.collect(squareRoot)
// OR
val result = list collect squareRoot

// The output is shown below
result: List[Double] = List(2.0, 4.0, 5.0)
```

- Doing the same with the map method

```
// Doing the same with the map method
val list: List[Double] = List(4, 16, 25, -9)
val result = list.map(Math.sqrt)
result: List[Double] = List(2.0, 4.0, 5.0, NaN)
```

Module as Parameter

```
import chisel3._
import chisel3.util._
import chisel3.experimental.{
      BaseModule
}

// Define IO interface as a Trait
trait ModuleIO {
      def in1: UInt
      def in2: UInt
      def out: UInt
}

class Add extends RawModule with ModuleIO {
      val in1 = IO(Input(UInt(8.W)))
      val in2 = IO(Input(UInt(8.W)))
      val out = IO(Output(UInt(8.W)))
      out := in1 + in2
}

class Sub extends RawModule with ModuleIO {
      val in1 = IO(Input(UInt(8.W)))
      val in2 = IO(Input(UInt(8.W)))
      val out = IO(Output(UInt(8.W)))
      out := in1 - in2
}
```

Module as Parameter Cont'd

```
class Top [T <: BaseModule with ModuleIO] (genT: => T) extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })
    val sub_Module = Module(genT)
    io.out := sub_Module.out
    sub_Module.in1 := io.in1
    sub_Module.in2 := io.in2
}

// Generate verilog for two modules, one for addition, second for subtraction
println((new chisel3.stage.ChiselStage).emitVerilog(new Top(new Add)))
println((new chisel3.stage.ChiselStage).emitVerilog(new Top(new Sub)))
```

Reading List I

- Read Chapters 12 and 19 of [Odersky et al., 2016] for in-depth understanding of Scala Collections
- The tutorial available at [Tutorial, 2020] is good resource for quick reference

References

Odersky, M., Spoon, L., and Venners, B. (2016).
Programming in Scala.
Artima Incorporation.

Tutorial (2020).
Scala tutorial.
https://www.tutorialspoint.com/scala/index.htm.