Object-Oriented Programming in Scala

(Tunight Talk)

Paul

May 30, 2020

Tonight

1. What is Scala & OO?
2. Objects: The Essence
3. Traits: Key of Code Reuse
4. Subtyping: The Soul
   - Type Bounds
   - Variances
   - Intersection & Union Types
5. Advanced Typing Features

Contents

Scala

- Invented by Martin Odersky, EPFL
- Scala stands for “scalable language”
- Runs on JVM
- Famous Projects
  - Spark: `https://spark.apache.org`
  - Flink: `https://flink.apache.org`
  - Akka: `https://akka.io`
  - Scalding: `https://github.com/twitter/scalding`
  - Play: `https://www.playframework.com`

Pros & Cons

Pros:

- Seamlessly interact with existing Java libraries
- Write less, synthesize more
- Flexible (syntax, programming paradigm)
- Leading language features that have been C# 9.0, Java 14, etc.

Cons:

- "Academic language"
- Steep learning curve
- Hard to hire people
- Slow compilation and running

Object-Oriented?

Some say:

> *The key features of object-oriented programming are encapsulation, inheritance and polymorphism.*

However, I find that Haskell, a purely functional programming language, also has them:

- functions are encapsulated (black box, no side effect)
- inheritance of type classes (e.g. `Ord` extends `Eq`)
- polymorphism derived from system F (e.g. `id :: a -> a` for all type a)

Contents

Slogan

Objects are first-class citizens!

Operators are Just Methods

```
1 + 2 // 1.+(2)
true && false // true.&&(false)

case class Point(x: Int, y: Int) {
  def +(that: Point): Point = Point(this.x + that.x, this.y + that.y)
}
Point(1, 2) + Point(3, 4) // -> Point(4, 5)
```

Operators are regarded as normal identifiers:

```
val ++ = 1
++ + ++ // -> 2
```

Functions are Just Objects

A binary function is an instance of the following trait:

```
trait Function1[-T1, +R] extends AnyRef {
  def apply(v1: T1): R
}
```

Calling a function means to invoke the apply method:

```
val twice = (x: Int) => x * 2
// new Function1[Int, Int] {
//   def apply(x: Int) = x * 2
// }
twice(5) // twice.apply(5)
```

Singleton

Singleton design pattern in Java:

```
public class Printer {
  private static final Printer instance = new Printer();
  private Printer() { /* ... */ }
  public static Printer getInstance() {
    return instance;
  }
}
```

In Scala:

```
object Printer { /* ... */ }
```

Objects as Parameters

Recall that "`obj f x`" is a short-hand for "`obj.f(x)`". In designing a *domain-specific language* (DSL), this syntax comes in handy:

```
object now
object simulate {
  def once(behavior: => Unit) = new {
    def right(n: now.type): Unit = /* ... */
  }
}

simulate once { someAction() } right now
// simulate.once({ someAction() }).right(now)
```

Singleton Types

A singleton type is a type *inhabited* by exactly one value. We access the type of an object using the ".type" syntax:

```
object A
def foo(x: A.type) = x
foo(A)
```

In Scala 3, a literal itself is also a singleton type:

```
def bar(x: 1, y: 2) = x + y
bar(1, 2)   // -> 3
bar(1, 3)   // type error
bar(1, 1+1) // -> 3
```

Objects as Modules

```
// Scala 2 (methods cannot be top-level components):
object MyLibrary {
  def parseId(s: String): Int = ???
}
// In another file
import MyLibrary._

// Scala 3 (methods can be top-level components):
def parseId(s: String): Int = ???
```

In Scala, object `scala.Predef` is automatically imported into scope:

```
println("Hello, world")
```

Unlike in Java:

```
System.out.println("Hello, world");
```

Static Members v.s. Companion Object

Java people use static members:

```
class User {
  private static int count = 0;

  public final String name;
  public final int id;
  public User(String name) {
    this.name = name;
    this.id = count;
    count++;
  }
}
```

Scala people use companion object:

```
class User(name: String) {
  import User._
  val id: Int = count
  increase()
}

object User {
  private var c = 0
  private def count = c
  private def increase() = c += 1
}
```

Contents

Trait?

Traits are a fundamental unit of code reuse in Scala. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.

– Chap. 12, Programming in Scala (2ed)

A Philosophical Frog

```
trait Philosophical {
  def philosophize(): Unit = println("I consume memory, therefore I am!")
}

abstract class Animal {
  def isAnimal: Boolean = true
}

class Frog extends Animal with Philosophical {
  override def toString = "green"
}

object Main {
  def main(args: Array[String]) = (new Frog).philosophize()
  // -> I consume memory, therefore I am!
}
```

Underlying Technique: Mixin

Let's disassemble the compiled JVM byte code:

```
javap -v Philosophical
javap -v Animal
javap -v Frog
javap -v Main$
javap -v Main
```

Trait v.s. Interface

In Java 8 (or higher):

```
@FunctionalInterface public interface Comparator<T> {
  int compare(T o1, T o2);
  default Comparator<T> reversed() { // since 1.8
    return Collections.reverseOrder(this);
  }
  /* ... */
}
```

In Scala:

```
trait Ordering[T] extends Comparator[T] with PartialOrdering[T] {
  def compare(x: T, y: T): Int
  def reverse: Ordering[T] = /* ... */
  def max(x: T, y: T): T = /* ... */
  /* ... */
}
```

Trait Inheritance: Ordering

```
trait Ordering[T] extends Comparator[T] with PartialOrdering[T]
trait PartialOrdering[T] extends Equiv[T]
trait Equiv[T]
```

Like Haskell's type class:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT

class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Trait Inheritance: Abstract Algebra

```
class Group a => Abelian a
class Monoid a => Group a where
  invert :: a -> a
class Monoid a where
  mappend :: a -> a -> a
  mempty :: a
```

can be translated into

```
trait Abelian[T] extends Group[T]
trait Group[T] extends Monoid[T] {
  def invert(x: T): T
}
trait Monoid[T] {
  def (x: T) * (y: T): T // Scala 3
  def _1: T
}
```

Linearlization

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Type	Linearization
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any
FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

Conventions

Should I use a trait, an abstract class or a concrete class?

- the behavior will not be reused: concrete class
- it might be reused in multiple, unrelated classes: trait
- it will be inherited in Java code: abstract class
- still do not know: trait

Trait Parameters

In Scala 3, traits are allowed to have parameters:

```
trait Greeting(val name: String) {
  def msg = s"How are you, $name"
}

class C extends Greeting("Bob") {
  println(msg)
}

class D extends C with Greeting("Bill") // error: parameter passed twice
```

Brainstorming

Q: Which programming languages have traits?

Contents

Objects as Records

An object is a collection of fields and methods, pretty much like ML's record:

```
type point = { x : int, y : int };
val p = { x = 1, y = 2 };
```

Scala supports record types via structural typing:

```
type Point = { val x: Int; val y: Int }
val p: Point = new { val x = 1; val y = 2 }
```

Inheritance, Revisited

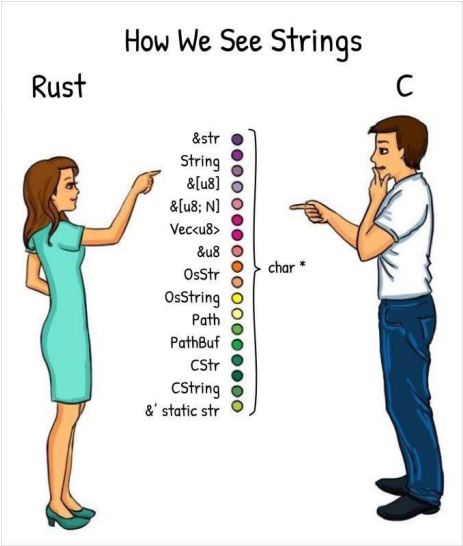
```
class A
class B extends A // all members of A are also members of B, B <: A

def foo(x: A): B = ???
val (a, b) = (new A, new B)
```

Which of the following method invocations are allowed?

```
foo(a)              // (1)
foo(b)              // (2)
val x: B = foo(a)   // (3)
val x: A = foo(a)   // (4)
```

Observation: throwing away some "extra" information is "safe"!

Type Refinement



Subtype Relation

A subtyping is a pre-order relation, i.e., a binary relation that is reflexive and transitive. We write $S <: T$ to pronounce "S is a subtype of T", or "T is a supertype of S".

$$\text{S-Refl}\ \frac{}{S <: S}$$

$$\text{S-Trans}\ \frac{S <: U \qquad U <: T}{S <: T}$$

Simply-Typed Lambda Calculus with Subtyping ($\lambda_{<:}$)

$$\text{Term } t ::= x \mid (t_1 t_2) \mid (\lambda x : T.t)$$
$$\text{Type } T ::= B \mid \top \mid T_1 \to T_2$$

- Subtyping rules: S-Refl, S-Trans and

$$\text{S-Top}\frac{}{S <: \top} \qquad \text{S-Arrow}\frac{T_1 <: S_1 \quad S_2 <: T_2}{(S_1 \to S_2) <: (T_1 \to T_2)}$$

- Typing rules:

$$\text{T-Var}\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash (\lambda x : T.t) : T \to T'}$$

$$\text{T-App}\frac{\Gamma \vdash t_1 : T \to T' \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 t_2) : T'} \qquad \text{T-Sub}\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

Record Extension with Subtyping ($\lambda^{\{\}}_{<:}$)

$$\text{Term } t ::= \cdots \mid \{l_1 = t_1, \ldots, l_n = t_n\} \mid t.l_i$$
$$\text{Type } T ::= \cdots \mid \{l_1 : T_1, \ldots, l_n : T_n\}$$

New subtyping rules:

$$\text{S-RecWidth}\ \frac{}{\{l_1 : T_1, \ldots, l_n : T_n, l_{n+1} : T_{n+1}, \ldots, l_m : T_m\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\text{S-RecDepth}\ \frac{\forall i. S_i <: T_i}{\{l_1 : S_1, \ldots, l_n : S_n\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\text{S-RecPerm}\ \frac{\{k_1 : S_1, \ldots, k_n : S_n\} \text{ is a permutation of } \{l_1 : T_1, \ldots, l_n : T_n\}}{\{k_1 : S_1, \ldots, k_n : S_n\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

Exercise Time!

- How many different supertypes does $\{l_1 : \top, l_2 : \top\}$ have?
  Six. $\{l_1 : \top\}$, $\{l_2 : \top\}$, $\{\}$, $\{l_1 : \top, l_2 : \top\}$, $\{l_2 : \top, l_1 : \top\}$, $\top$.
- Can you find an infinite ascending chain in the subtype relation?
  $\{\} \to \top <: \{l_1 : \top\} \to \top <: \{l_1 : \top, l_2 : \top\} \to \top <: \cdots$
- Is there a type that is a subtype of every other type?
  No. By inversion.
- Is there an arrow type that is a supertype of every other arrow type?
  No. If there were such an arrow type $T_1 \to T_2$, then T_1 would have to be a subtype of every other type, which we have just seen is impossible.

Contents

$\lambda^{\{\}}_{<:}$ with Bottom Types

$$\text{Type } T ::= \cdots \mid \bot$$

New subtyping rule:

$$\text{S-Bot}\ \frac{}{\bot <: T}$$

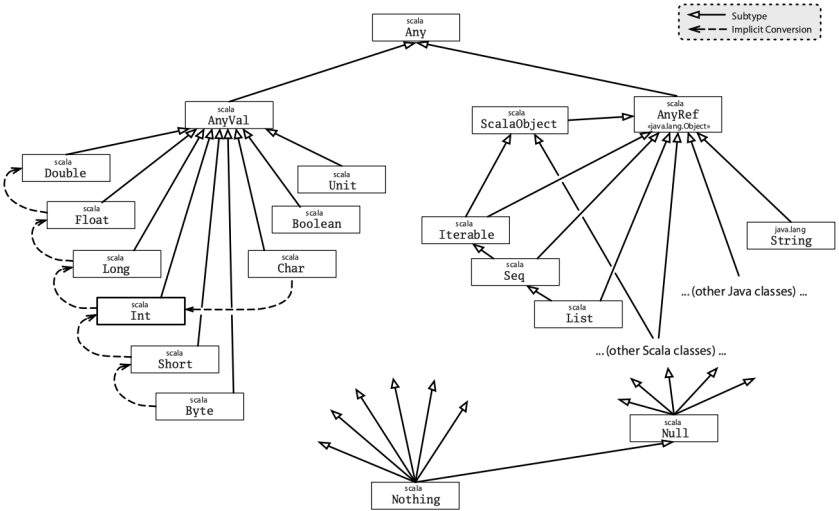
Remarks:

- There are no closed values of type $\bot$. Suppose there is one, say v, then $\vdash v : \top \to \top$, which is impossible.
- In practice, $\bot$ indicates an exception, or whatever value.

Top & Bottom Types in Scala

```
// top
abstract class Any
class AnyRef extends Any // AnyRef = java.lang.Object
final class AnyVal extends Any

// bottom
abstract final class Nothing extends Any
def ???: Nothing = throw new NotImplementedError
def yourMethod(str: String): String = ???
```

Class Hierarchy of Scala



Type Bounds

Type parameters can be restricted to be a subtype/supertype of some type.

```
// Implicit conversions from a Java Array into collection.Mutable.ArrayOps
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): ArrayOps[T] =
  new ArrayOps.ofRef[T](xs)

// getOrElse method in Option
sealed abstract class Option[+A] extends Product with Serializable {
  @inline final def getOrElse[B >: A](default: => B): B = { /* ... */ }
}
```

Type Bounds with Structural Typing

How to implement a method using that will automatically close a resource after using it?

```
using(new PrintWriter("date.txt")) { writer =>
  writer.println(new Date)
}

def using[T <: { def close(): Unit }, S](obj: T)(operation: T => S) = {
  val result = operation(obj)
  obj.close()
  result
}
```

Contents

Classification

A *type constructor* X is

- covariant if it preserves the ordering of types ($S <: T \to X[S] <: X[T]$);
- contravariant if it reverses this ordering ($S <: T \to X[T] <: X[S]$);
- invariant if neither of the above applies.

In Scala and Java:

Variances	Scala	Java
Covariant	`+T`	`? extends T`
Contravariant	`-T`	`? super T`
Invariant	`T`	`T`

Function Trait, Again

Recall that

$$\text{S-Arrow}\frac{T_1 <: S_1 \qquad S_2 <: T_2}{(S_1 \to S_2) <: (T_1 \to T_2)}$$

In scala, a function is implemented as a trait

```
trait Function1[-T1, +R] extends AnyRef {
  def apply(v1: T1): R
}
```

where the parameter is contravariant and the return value is covariant.

Comparer

Suppose we have a comparer for comparing two objects of some class `A`. Then we should also have this comparer for all the subclasses of `A`. Contravariant comes in rescue:

```
public interface IComparer<in T> {
  int Compare(T left, T right);
}
```

Sink

In real-world event-driven software systems, when an event is fired, all its "super levels" should also be notified. To achieve this, we can declare the trait Sink as a contravariant. [1]

```
trait Event
trait UserEvent extends Event
trait SystemEvent extends Event
trait ApplicationEvent extends SystemEvent
trait ErrorEvent extends ApplicationEvent

trait Sink[-In] { def notify(o: In) }

def appEventFired(e: ApplicationEvent, s: Sink[ApplicationEvent]) = s.notify(e)
def errorEventFired(e: ErrorEvent, s: Sink[ErrorEvent]) = s.notify(e)
```

[1] http://blog.petruescu.com/programming/types/scala-types-contravariance/

Sink (Cont.)

```
trait SystemEventSink extends Sink[SystemEvent]
val ses = new SystemEventSink {
  override def notify(o: SystemEvent): Unit = ???
}

trait GenericEventSink extends Sink[Event]
val ges = new GenericEventSink {
  override def notify(o: Event): Unit = ???
}

// You can call:
appEventFired(new ApplicationEvent {}, ses)
errorEventFired(new ErrorEvent {}, ges)
appEventFired(new ApplicationEvent {}, ges)
```

Contents

Motivation

- A type can be interpreted as a set, which contains all closed values of that type.
- Since we have intersection and union operations on sets, can we add them to types?

Intersection Types

The *inhabitants* of an intersection type $T_1 \cap T_2$ are terms belonging to both T_1 and T_2:

$$\text{S-}\cap\text{Proj1}\ \frac{}{T_1 \cap T_2 <: T_1} \qquad \text{S-}\cap\text{Proj2}\ \frac{}{T_1 \cap T_2 <: T_2}$$

$$\text{S-}\cap\text{Form}\ \frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \cap T_2}$$

Remark: in words of lattice theory, $\cap$ is a *meet* operator.

Intersection Types in Scala 3

```
trait Resettable { def reset(): this.type }
trait Growable[T] { def add(x: T): this.type }
def f(x: Resettable & Growable[String]) = {
  x.reset()
  x.add("first")
}

trait A { def children: List[A] }
trait B { def children: List[B] }
class C extends A with B {
  def children: List[A & B] = ???
}
val x: A & B = new C
val ys: List[A & B] = x.children
// Since List is covariant, List[A] & List[B] = List[A & B]
```

Union Types

The inhabitants of a union type $T_1 \cup T_2$ are terms belonging to either T_1 or T_2, formulated by the following subtyping rules:

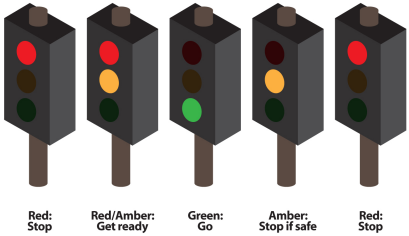
$$\text{S-}\cap\text{Form1}\ \frac{}{T_1 <: T_1 \cup T_2} \qquad \text{S-}\cap\text{Form2}\ \frac{}{T_2 <: T_1 \cup T_2}$$

$$\text{S-}\cap\text{Proj}\ \frac{T_1 <: S \qquad T_2 <: S}{T_1 \cup T_2 <: S}$$

Remark: in words of lattice theory, $\cup$ is a *join* operator.

Union Types in Scala 3

```
trait Light
case object Red extends Light
case object Yellow extends Light
case object Green extends Light

type Ready = Red.type | Yellow.type

def foo(x: Ready) = x match {
  case Red => ???
  case Yellow => ???
}
```



Contents

Path-Dependent Types

```
import scala.collection.mutable

case class Board(length: Int, height: Int) {
  case class Coordinate(x: Int, y: Int) {
    require(0 <= x && x < length && 0 <= y && y < height)
  }
  val occupied = scala.collection.mutable.Set[Coordinate]()
}

val b1 = Board(20, 20)
val b2 = Board(30, 30)
val c1 = b1.Coordinate(15, 15)
val c2 = b2.Coordinate(25, 25)
b1.occupied += c1
b2.occupied += c2
```

```
b1.occupied += c2 // not compile
val b3: b1.type = b1
val c3 = b3.Coordinate(10, 10)
b1.occupied += c3 // compiles
```

Type Projection

In the above example, how can I manipulate a `Coordinate` of an arbitrary `Board`?

```
def distance(c: Board#Coordinate) = Math.sqrt(c.x * c.x + c.y * c.y)
def distance(c1: Board#Coordinate, c2: Board#Coordinate) = {
  val dx = c1.x - c2.x
  val dy = c1.y - c2.y
  Math.sqrt(dx * dx + dy * dy)
}

distance(c1)
distance(c2)
distance(c1, c2)
```

Type Members

Like methods and fields, types can also be a member of a class/object/trait:

```
abstract class Value {
  type T
  val value: T
}

class IntValue(override val value: Int) extends Value { type T = Int }
class StringValue(override val value: String) extends Value { type T = String }

type ValueList = List[Value]
```

Higher-Kinded Types

Higher-kinded types are also called *type constructors* because they are used to construct types.

```
type Callback[T] = T => Unit
def id[M[_]](f: M[Int]) = f
val g = id[Callback] { x => println(x) }

// Scala 3: type lambdas
type Callback = [T] =>> (T => Unit)
```

Foundation of Scala 3: Dependent Object Types (DOT)

> *While hiking together in the French alps in 2013, Martin Odersky tried to explain to Phil Wadler why languages like Scala had foundations that were not directly related via the Curry-Howard isomorphism to logic. This did not go over well. As you would expect, Phil strongly disapproved. He argued that anything that was useful should have a grounding in logic. In this paper, we try to approach this goal.*

Amin, Nada, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. *The essence of dependent object types*. WadlerFest 2016.

Take-Home Messages

Scala is not a “purely functional language”, instead, “functional guys” hate it because

- it runs on the JVM,
- it allows mutable data and side effects, and
- the standard library has poor support for monadic programming.

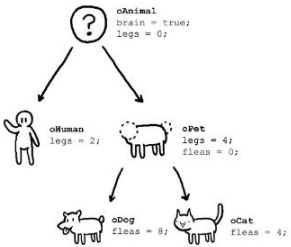
OO is not the antonym of FP. Many language features can be shared:

- immutable data structures,
- lazy evaluation,
- pattern matching,
- monadic programming, etc.

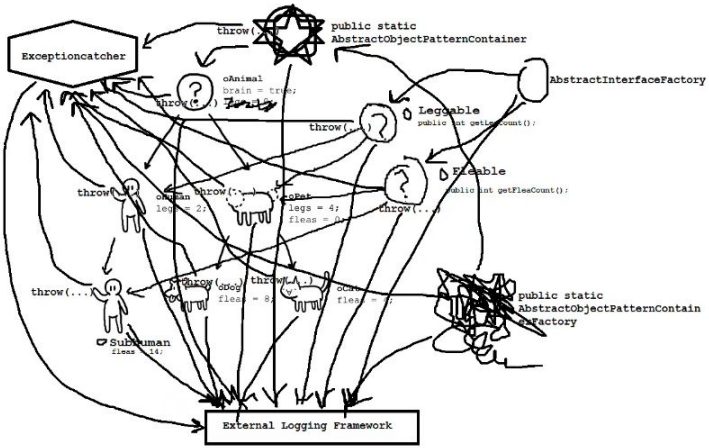
OO is not a cure-all, neither is FP!

OO is Much More Complex Than You Thought!

What OOP users claim



What actually happens



Further Reading

- Martin Odersky, Lex Spoon, Bill Venners. Programming in Scala, 2ed.
- Joshua D. Suereth. Scala in Depth.
- Nilanjan Raychaudhuri. Scala in Action.
- Coursera: Functional Programming Principles in Scala
- Dotty: `https://dotty.epfl.ch`
- DOT papers on OOPSLA, FOOL, etc.

Thanks!

Q & A