Concepts of Programming Languages:
Scala Introductory Overview

Ragnar Mogk

Announcements

- https://cast.informatik.tu-darmstadt.de/

Why (we) use Scala?

- General purpose programming language
- Powerful type system.
- Succinct: programmers want to be more productive.
- Integrates object-oriented and functional paradigms.
- It's fully interoperable with Java (and JavaScript).
- Scalable: from scripting(*) to large projects.
  - (*) well … small projects
- Extensible: new control structures, DSLs.

Enterprise Experience

- Companies who depend on Java are turning to Scala to boost productivity, applications scalability and reliability.

- For example, at Twitter, the social networking service, Robey Pointer moved their core message queue from Ruby to Scala. This change was driven by the companies need to reliably scale their operation to meet fast growing Tweet rates, already reaching 5000 per minute during the Obama Inauguration.

- LinkedIn, Twitter, Foursquare, Netflix, Tumblr, The Guardian, precog, Sony, AirBnB, Klout, Apple

- Mostly server applications, some Android (seems to be less relevant since Kotlin), ScalaJS

Integration with Java

- Scala runs on the JVM.
- You can call Scala from Java and vice versa.
- Familiar Environment:
  - IDEs: Eclipse, Intellij
  - Libraries, Frameworks
- Gradual migration in programming style is possible.
- .NET support not currently maintained, but:
  - I don't want to give you an estimated time of arrival but it should be certainly this year including visual studio support. (Odersky, Jan 2011)
- 2017: .NET support does not exist (and no one cares anymore)
  - But we now have JavaScript (stable) and native (experimental) backends

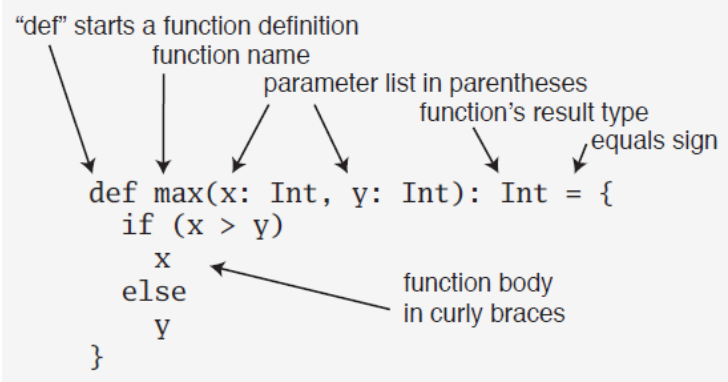
Scala is a functional language

- Every function is a value.
- Anonymous / Higher-order functions
- Functions can be nested
- Tail recursion
- Currying
- Closures

```
(x: Int) => x + more
```

- Typical operations on lists

```
scala> someNumbers.filter((x: Int) => x > 0)
res6: List[Int] = List(5, 10)
```

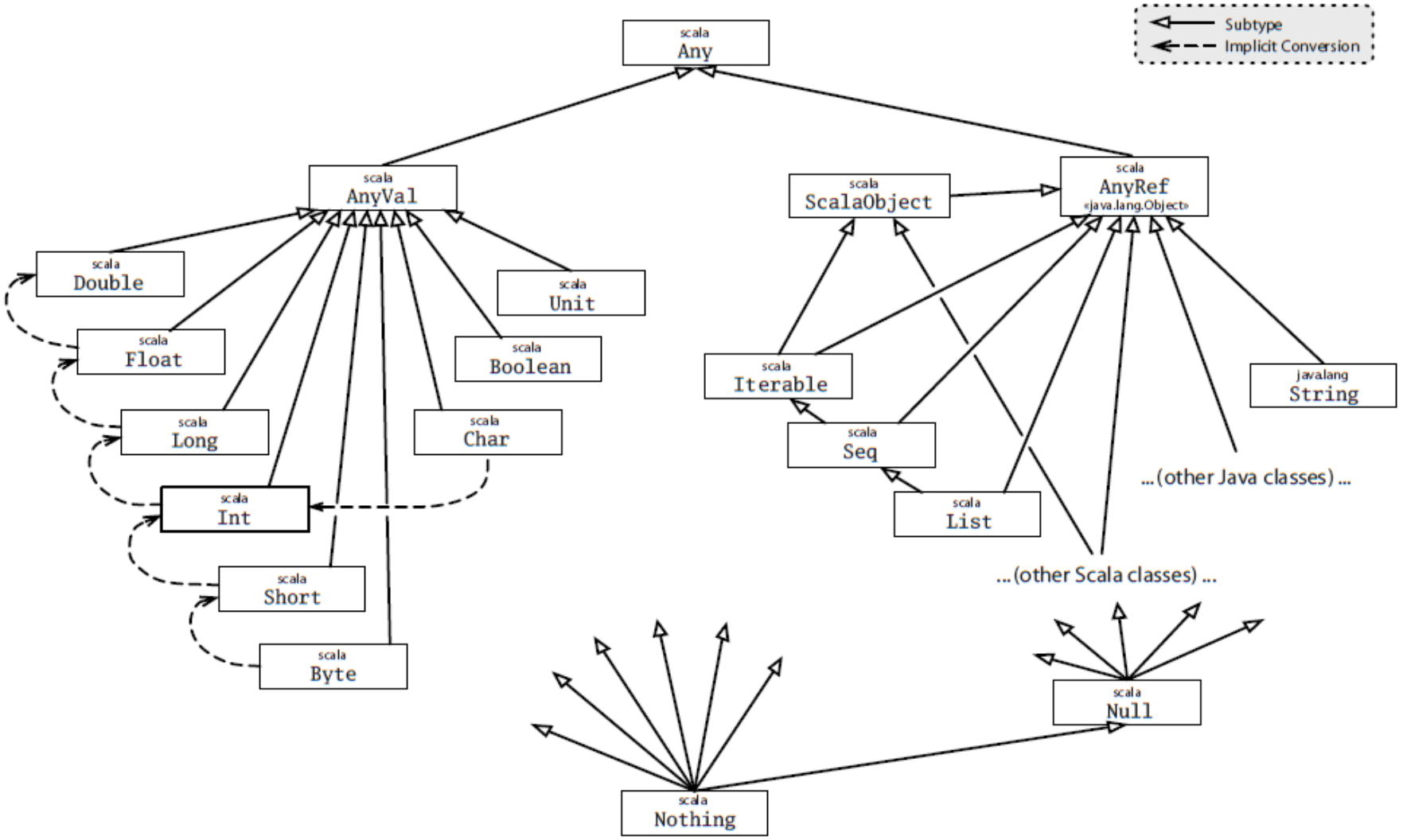


Case Classes

- Pattern matching
- Accessors
- Equality & Hashcode

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

Types


Subtype
Implicit Conversion
scala
Any
scala
AnyVal
scala
ScalaObject
scala
AnyRef
«java.lang.Object»
scala
Double
scala
Float
scala
Long
scala
Int
scala
Short
scala
Byte
scala
Unit
scala
Boolean
scala
Char
scala
Iterable
scala
Seq
scala
List
javalang
String
...(other Java classes)...
...(other Scala classes)...
scala
Null
scala
Nothing

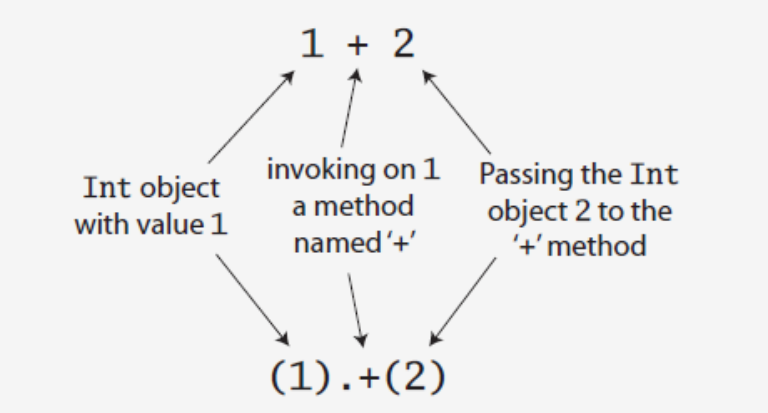
Pattern Matching

- Pattern match includes a sequence of alternatives

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e))  => e   // Double negation
  case BinOp("+", e, Number(0)) => e   // Adding zero
  case BinOp("*", e, Number(1)) => e   // Multiplying by one
  case _ => expr
}
```

Extensibility: DSLs

- Flexible method invocation syntax.
  - a.m(b) ==> a m b
- Tricks:
  - precedence based on first character
  - receiver based on last character



Extensibility: control structures

- To make the code look like a built-in control structure you can use curly brackets instead of parenthesis for the argument list
  - This curly brackets technique will work only if you are passing in one argument
  - Similar to Ruby code blocks.

```
def dont(code: => Unit) = new DontCommand(code)

class DontCommand(code: => Unit) {
  def unless(condition: => Boolean) =
    if (condition) code
} }

dont {
  println("Yep, 2 really is greater than 1.")
} unless (2 > 1)
```

```
val echoActor = actor {
  while (true) {
    receive {
      case msg =>
        println("received message: "+ msg)
    }
  }
}
```

Implicit conversions

- Implicit conversions support conversions into the target type, a type that's needed at some point in the code.
- Well structured conversions are achievable with implicit classes

```
implicit def stringWrapper(s: String) =
  new RandomAccessSeq[Char] {
    def length = s.length
    def apply(i: Int) = s.charAt(i)
  }

scala> stringWrapper("abc123") exists (_.isDigit)
res0: Boolean = true
scala> "abc123" exists (_.isDigit)
res1: Boolean = true
```

Implicit Parameters

- Implicit parameters

```
object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt) {
    println("Welcome, "+ name +". The system is ready.")
    println(prompt.preference)
  }
}
```

Inheritance Model

- 2017: Similar to Java8 Interfaces

- Solutions in other programming languages:
  - Multiple inheritance (e.g. Common Lisp).
    - Too complex. E.g. the linearization algothithm.
  - Single inheritance + Interfaces
    - Limits reuse.
    - Java interfaces are more often thin than rich: implementing an interface requires to implement each method.
  - Scala: traits (mixin-based inheritance).

Inheritance Model: Traits

- Traits are like Java interfaces with concrete methods.
  - But traits can declare fields and maintain state
  - You cannot instantiate a Trait
- Adding a method to a Scala trait is a one-time effort.
- In classes, super calls are bounded at class creation.
- In a trait the implementation is determined each time the trait is mixed into a concrete class (statically).

Type Parametrization

- Variance is explicit.
- Generic types are by default nonvariant (“rigid”).
  - If S is a subtype of type T, then should Queue[S] be considered a subtype of Queue[T]? If so, Queue is covariant in its type parameter T.
  - Queue[String] can be passed to a method with parameter Queue[AnyRef].
- Variance annotations:

```
trait Queue[+T] { ... }
trait Queue[-T] { ... }
```

```
class Queue[+T] (private val leading: List[T],
    private val trailing: List[T] ) {
  def append[U >: T](x: U) =
    new Queue[U](leading, x :: trailing) // ...
}
```

Concurrency

- TL;DR: Functional + JVM = Concurrency <3

- Number of processors per machine is increasing
- Common model of concurrency in imperative languages: shared, mutable state with access synch (locks, ecc...).
  - Difficult kind of programming.
- Functional language: no side effects, no shared/mutable state.
  - Synchronization isn't necessary for reading immutable objects
  - There is nothing to synch.
  - Resurgent interest in function programming recently.
- The Actor Model makes concurrency far easier to manage: share-nothing, message-passing style.

Exercises

- Demo exercise available at https://submission.st.informatik.tu-darmstadt.de/

- Questions... ?