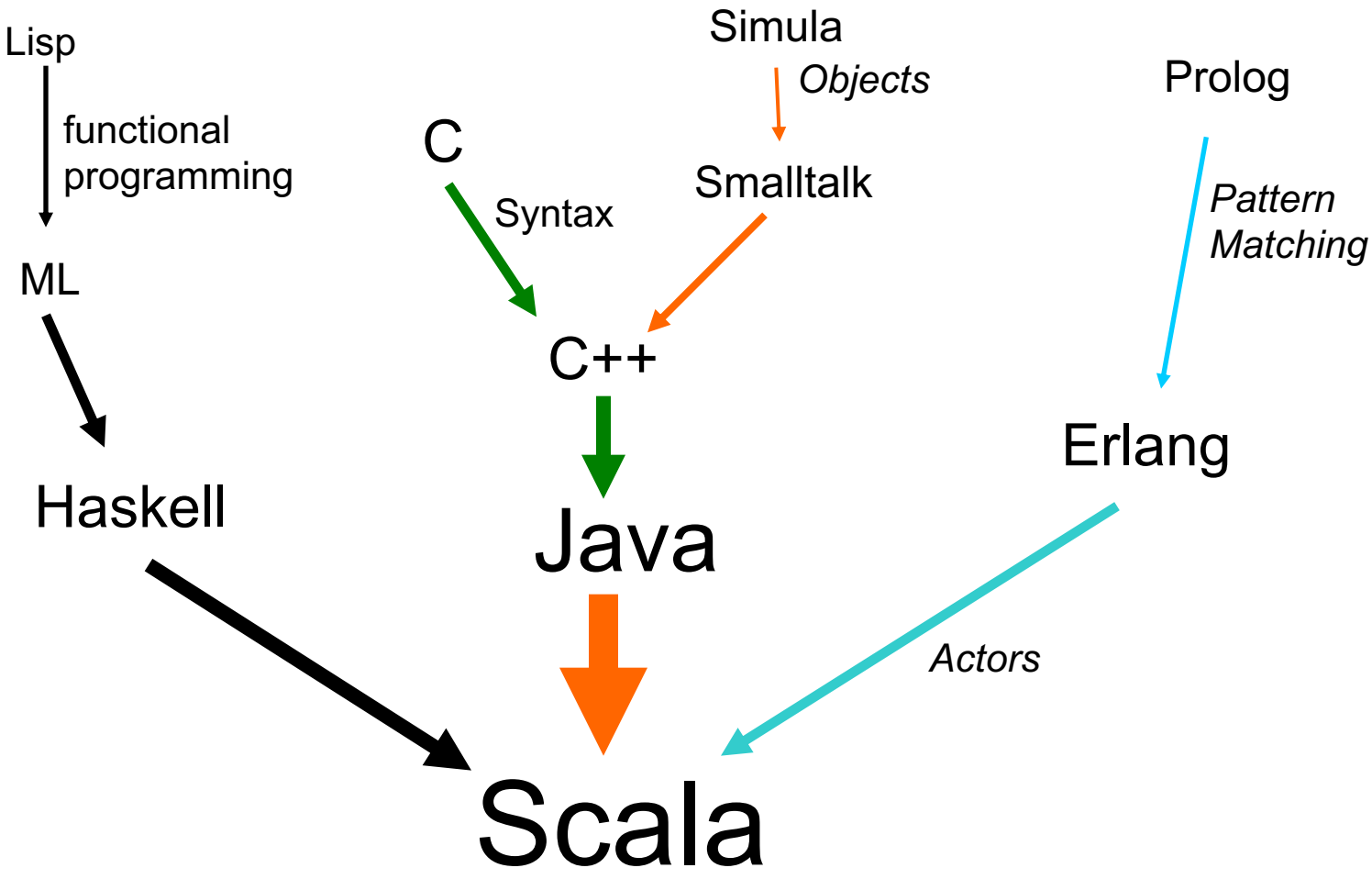
Introduction to Scala

- Overview
- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

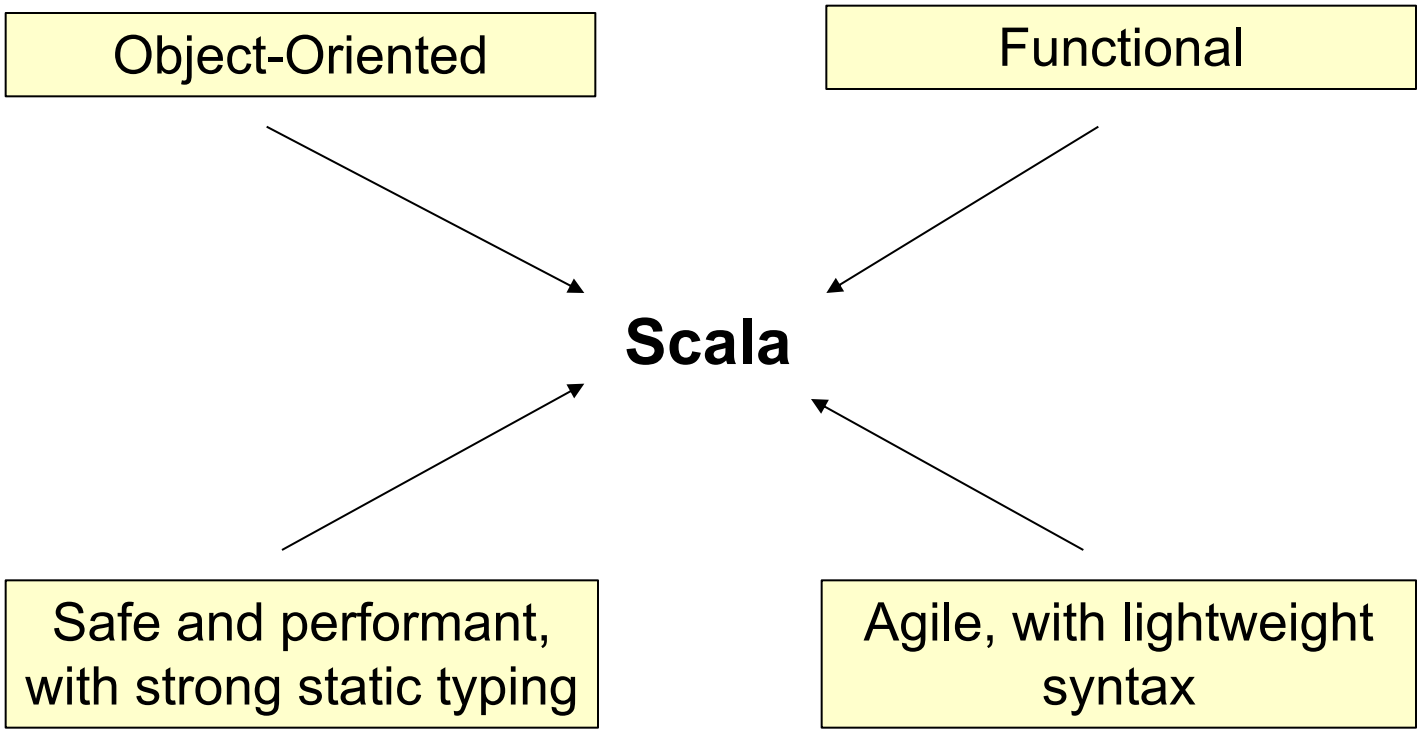
Multiparadigm Programming

- **Multi-paradigm Programming**
  - A multi-paradigm programming language provides “a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms.” [Tim Budd]
- **Programming Expertise**
  - “Research results from the psychology of programming indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual than it is the number of years’ experience in programming.” [Tim Budd]
- **Language Trend**
  - New explicitly multi-paradigm (object-oriented/functional) languages are appearing, e.g.,
    - Java platform: Scala, Kotlin
    - .Net platform: F#
  - https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

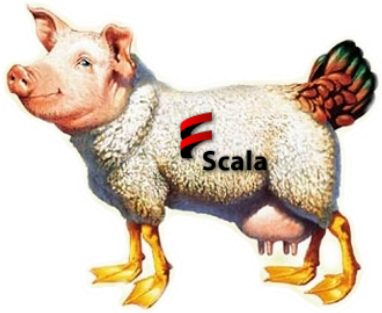
Scala Genealogy

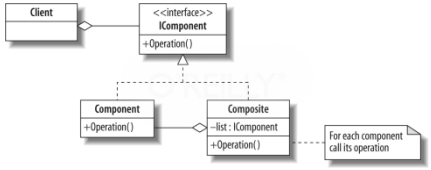


Scala is a Unifier



Why unify FP and OOP?





Functional Programming

- **Makes it easy to build interesting things from simple parts, using**
  - Higher-order functions
  - Algebraic types and pattern matching
  - Parametric polymorphism (Generics)

Object-oriented Programming

- **Makes it easy to adapt and extend complex systems, using**
  - Subtyping and subclassing (inheritance)
  - Classes as partial abstractions (abstract classes)
  - Dynamic configurations (component software)

Scala History

- **Martin Odersky**
  - Professor at EPFL, Programming Methods Laboratory
  - PhD from ETHZ (Supervisor: Niklaus Wirth)
- **Scala History**
  - 1996-1997 Pizza: Java with
- parametric polymorphism
- higher order functions
- abstract data structures
  - 1998-2000 GJ: Generic Types for Java (basis for Java5)

make Java better

  - 2001-2006 The Scala Experiment
  - 2006-2012 An Industrial strength programming language

make a better Java

Java Pos & Cons

- **Cons**
  - Very imperative: How instead of What
  - Not designed for highly concurrent programs
  - Verbose, too much boilerplate code (=> Eclipse: Source code generators)
    - Getters & Setters (=> Project Lombok)
- **Pros (The Good Parts of Java)**
  - Popularity and acceptance
  - Object-oriented (largely) and strong typing
  - Library, Tools
  - Java Virtual Machine (JVM)
    - Platform independent
    - Highly optimized: JIT, Garbage Collection
  - Several Languages were defined on top of the JVM:
    - Clojure / Groovy / JRuby / Jython / Scala / Kotlin / Ceylon

Scala Advantages over Java

- **In Scala functions are first class and can be passed around**
  - This encourages functional programming with all its advantages
- **In Scala all values are objects (pure object-oriented)**
  - The compiler turns them into primitives, so no efficiency is lost
- **In Scala operators are just methods**
  - `a * b ⇔ a.*(b)`
- **Scala is statically typed (as Java) but uses type inference**
  - `val m = HashMap[String,List[String]]()`
- **Scala supports the principle of uniform access**
  - A field defined as attribute or as method is accessed uniformly
- **Scala supports concurrency**
  - Actor Library for coarse-grained concurrency
  - Immutable data structures => avoids race conditions

Scala Advantages over Java

- **Scala is concise**
  - Person.java

```
class Person {
   private final String name;
   private int age;
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public String getName() { return name; }
   public int getAge() { return age; }
   public void setAge(int age) { this.age = age; }
}
```

  - Person.scala

```
class Person (val name: String, var age: Int)
```

Scala Advantages over Java

- **Scala is concise**
  - Demo.java

```
public class Demo {
   public static void main(String[] args) {
      List<Person> people = new LinkedList<>();
      people.add(new Person("Romeo", 22));
      people.add(new Person("Julia", 21));
      List<Person> adults = new LinkedList<>();
      for(Person p : people) {
         if(p.getAge() >= 18) {
            adults.add(p);
         }
      }
   }
}
```

Scala Advantages over Java

- **Scala is concise**
  - Demo.scala

```
object Demo {
   def main(args: Array[String]): Unit = {
      val people = List(new Person("Romeo", 22),
                        new Person("Julia", 21))

      val adults = people.filter(p => p.age >= 18)
   }
}
```

Scala Advantages over Java

- **Scala is concise**
  - Demo.java (since Java 8)

```
public class Demo {
   public static void main(String[] args) {
      List<Person> people = Arrays.asList(
         new Person("Romeo", 22),
         new Person("Julia", 21));
      List<Person> adults = people.stream()
         .filter(p -> p.getAge() >= 18)
         .collect(Collectors.toList());
   }
}
```

Scala Scalability

- **Scala is scalable**
  - It can be used for *programming in the small* and *programming in the large*
    - *The way it helps you is by not having to mix many specialized languages. You can use the same language for very small as well as very large programs, for general purpose things as well as special purpose application domains [Odersky]*
- **Scala is extensible**
  - Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries
    - Example: Actor library
  - Guy Steele has formulated a benchmark for measuring language extensibility [Growing a Language, OOPSLA'98]
    - *Can you add a type of complex numbers to the library and make it work as if it was a native number type?*

Scala Scalability

- **Adding new data types: Complex**

```
class Complex(val re: Double, val im: Double) {
   def + (x: Complex) = new Complex(re+x.re, im+x.im)
   def - (x: Complex) = new Complex(re-x.re, im-x.im)
   def unary_- = new Complex(-re, -im)
   def * (x: Complex) =
      new Complex(re*x.re-im*x.im, re*x.im+im*x.re)
   def / (y: Complex) = {
      val d = y.re*y.re+y.im*y.im;
      new Complex((re*y.re+im*y.im)/d, (im*y.re-re*y.im)/d)
   }
   override def toString =
      if ( im == 0) re.toString
      else if (re == 0) im + "i"
      else re + (if(im < 0) "" else "+") + im + "i"
}
```

Scala Scalability

```
object Complex {
   def apply(re: Double, im: Double) = new Complex(re, im)
   implicit def DoubleToComplex(d: Double) = Complex(d, 0)
}
```

– Sample usage:

```
scala> import Complex._
import Complex._
scala> val c1 = Complex(1,2)
c1: Complex = 1.0+2.0i
scala> val c2 = 4 - c1
c2: Complex = 3.0-2.0i
scala> c1 / c2
res0: Complex = -0.07692307692307693+0.6153846153846154i
scala> -c1
res1: Complex = -1.0-2.0i
```

Scala Scalability

- **Adding new control structures: `using` (try-with in Java)**

```
def using[A, B <: { def close(): Unit }]
   (closeable: B) (f: B => A): A =
       try { f(closeable) } finally { closeable.close() }
```

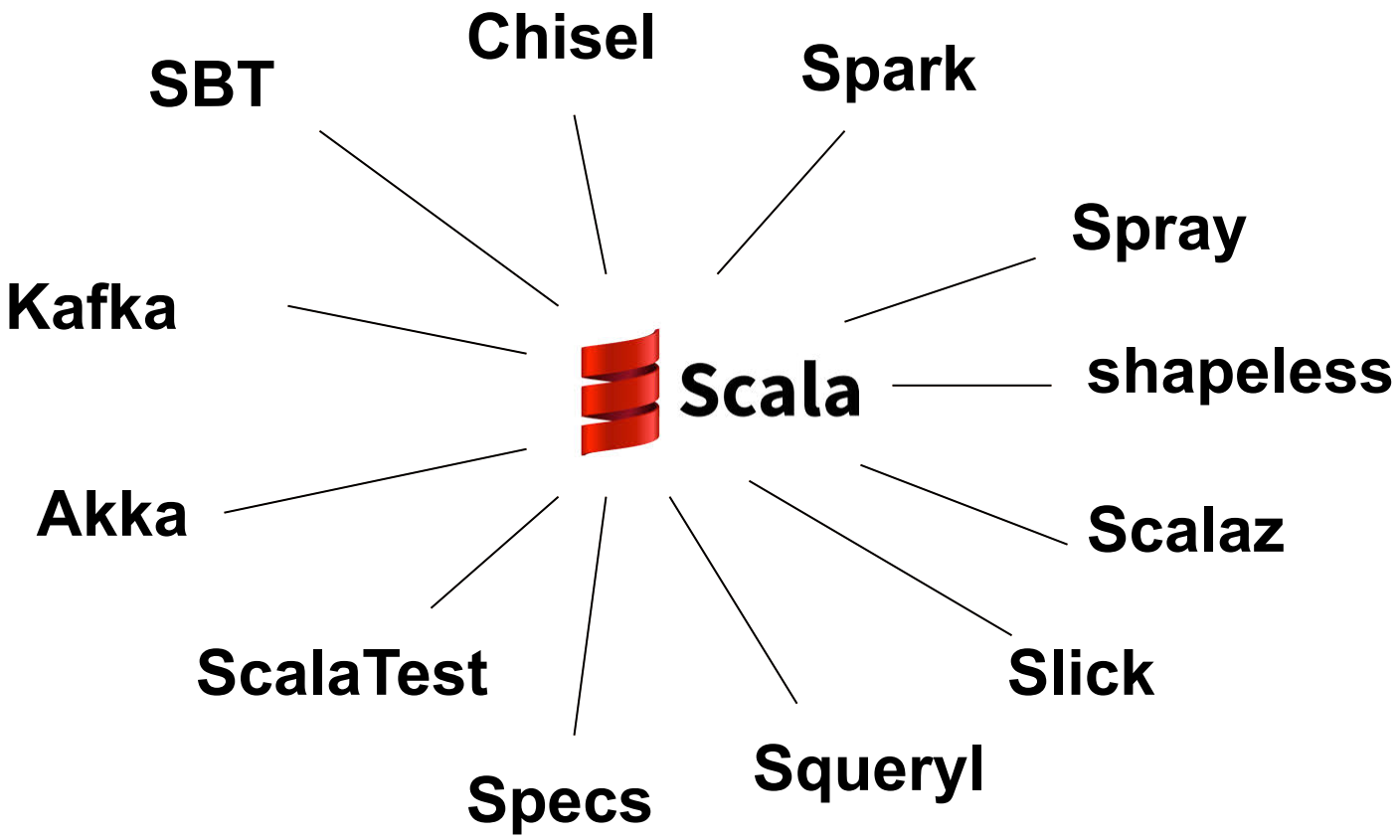
  - Invocation:

```
using(new java.io.PrintWriter("sample.txt")) { out =>
  out.println("hello world!")
}
```

```
val x = new Object() { def close() { println("close") }}
using (x) { x => println(x) }
using (x) ( x => println(x) )
using {x} { x => println(x) }
```

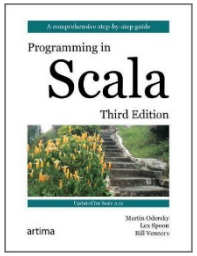
Scala Ecosystem

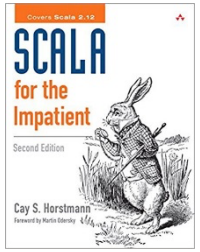
- Libraries on top of Scala

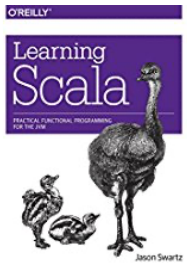


Scala References

- http://www.scala-lang.org
  - Documentation -> Tutorials
  - Documentation -> Overviews/Guides
  - Documentation -> Specification
- **Books**
  - Martin Odersky, Lex Spoon, and Bill Venners;
    Programming in Scala: 3rd edition;
    Artima, Inc., 2016
  - Cay S. Horstmann;
    Scala for the Impatient, 2nd edition;
    Addison Wesley, 2016.
  - Jason Swartz;
    Learning Scala;
    O'Reilly, 30.12.2014







Introduction to Scala

- Overview
- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

Getting Started

- **Scala Interpreter**

```
> scala
Welcome to Scala 2.12.2 (Java HotSpot(TM) 64-Bit Server VM,
Java 1.8.0_102).
Type in expressions for evaluation. Or try :help.

scala> val a = 3
a: Int = 3

scala> val f : Int => Int = x => x*x
f: Int => Int = $$Lambda$1036/934225099@7926352f

scala> f(a)
res0: Int = 9
```

Getting Started

- **Scala Compiler**

```
object HelloWorld {
   def main(args: Array[String]) {
      println("Hello world!")
   }
}
```

```
>scalac HelloWorld.scala

>scala HelloWorld
Hello world!

>scala -cp . HelloWorld
Hello world!

>java -cp scala-library.jar;. HelloWorld
Hello world!
```

Getting Started

- **Scala Compiler generates Java class files**

```
>scalac Complex.scala
>javap Complex
Compiled from "Complex.scala"
public class Complex {
    public static Complex DoubleToComplex(double);
    public static Complex apply(double, double);
    public double re();
    public double im();
    public Complex $plus(Complex);
    public Complex $minus(Complex);
    public Complex unary_$minus();
    public Complex $times(Complex);
    public Complex $div(Complex);
    public java.lang.String toString();
    public Complex(double, double);
}
```

Getting Started

- **Scala Code may use any Java class**

```
import javax.swing._
import java.awt._

object SampleGUI {
   def main(args: Array[String]) {
      val f = new JFrame("Title")
      f.setLayout(new FlowLayout())
      f.add(new JLabel("text"))
      f.add(new JButton("OK"))
      f.pack
      f.setVisible(true)
   }
}
```

Introduction to Scala

- Overview
- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

Scala Types

- **Basic Types**
  - Byte, Short, Int, Long, Float, Double
    - have methods
    - Operators are method calls
    - Assignment compatibility: Byte -> Short -> Int -> Long -> Float -> Double
  - Char
    - Assignment compatibility: Char -> Int
  - Boolean
  - String
    - More methods as in Java:
    - String interpolation
    - Multiline Strings
  - Unit

```
5.toFloat        5.0.hashCode
```

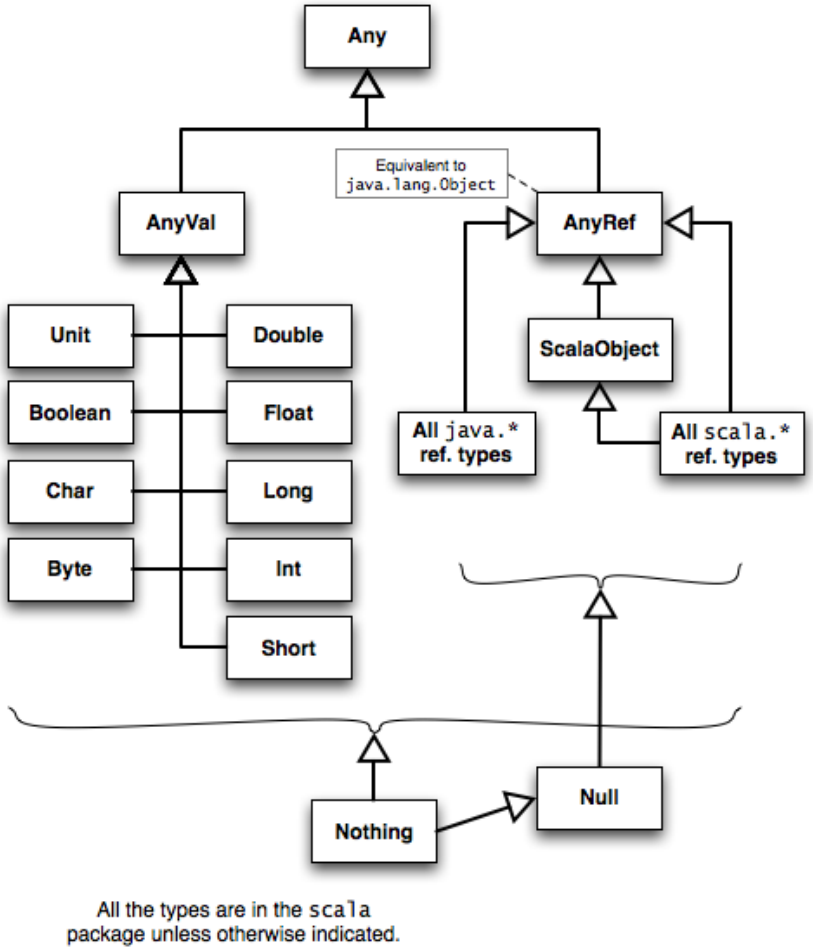
```
10 ./(3)         5.unary_-
```

```
"Scala".drop(2)          => ala
"Scala".reverse          => alacS
```

```
"""this is a string
Which contains a newline"""
```

Scala Types

- **Type Hierarchy**
  - Any = Scala base type
    - == a==b ⇔ a.equals(b)
    - !=
    - isInstanceOf [T]
    - asInstanceOf [T]
  - AnyRef = root of all reference types = Java base type Object
    - equals
    - eq (reference comparison)
    - hashCode
    - wait / notify
    - getClass



Scala Types

- **Unit**
  - Corresponds to void in Java, is used as result type for "procedures"
  - Unit type has only one value: ()
- **Null**
  - Subtype of all reference types
  - Its only instance is null
- **Nothing**
  - Bottom type, is a subtype of all types
  - Type Nothing has *no* instances!
    - Example: What is the type of an empty list which can be assigned to a list of any type?

```
scala> val empty = List()
empty: List[Nothing] = List()
scala> val ls : List[String] = empty
ls: List[String] = List()
scala> val li : List[Int] = empty
li: List[Int] = List()
```

Scala Types

- **Lists (`scala.collection.immutable.List`)**
  - Lists are concrete classes, no interfaces
  - A list is implemented as linked list and may contain an arbitrary number of elements
  - Lists are immutable (mutable variants exist)

```
scala> val list = List("Hello", "World", "!")
list: List[java.lang.String] = List(Hello, World, !)
scala> list.head
res1: java.lang.String = Hello
scala> list.tail
res2: List[java.lang.String] = List(World, !)
scala> list(2)
res3: java.lang.String = !
scala>  1 :: 2 :: 3 :: Nil
res4: List[Int] = List(1, 2, 3)
```

Scala Types

- **Tuples (`scala.Tuple*`)**
  - Contains a fixed number of elements
  - Element types may be different (not homogeneous as in lists)
  - Access is 1-based

```
scala> val t = Tuple2("Romeo", 22)
t: (String, Int) = (Romeo,22)
scala> t._2
res0: Int = 22
scala> "Romeo" -> 22
res1: (String, Int) = (Romeo,22)
scala> (1,2,3,4)
res2: (Int, Int, Int, Int) = (1,2,3,4)
```

Scala Types

- **Maps (`scala.collection.immutable.Map`)**
  - Map contains pairs
  - Immutable (mutable variants exist)

```
scala> var m = Map("Romeo" -> 22, ("Julia", 21))
m: scala.collection.immutable.Map[String,Int] =
Map(Romeo -> 22, Julia -> 21)
scala> m("Romeo")
res1: Int = 22
scala> m = m.updated("Romeo", 25)
m: scala.collection.immutable.Map[String,Int] =
Map(Romeo -> 25, Julia -> 21)
scala> m += ("Meier" -> 33)
scala> m
res2: scala.collection.immutable.Map[String,Int] =
Map(Romeo -> 25), Julia -> 21, Meier -> 33)
```

Scala Types

- **Variable declaration**
  - val value, cannot be changed (corresponds to Java's final)
  - var variable, can be reassigned
- **Type declaration is optional upon definition**

```
scala> val age = 3
age: Int = 3
scala> age = 4
<console>:12: error: reassignment to val
       age = 4
           ^
scala> var age = 34
age: Int = 34
scala> age = 35
age: Int = 35
```

Introduction to Scala

- Overview
- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

Control Expressions

- **If-expression**

  `"if" "(" BooleanExpr ")" Expr ["else" Expr]`

  - If-expression returns a result (and has a type!)
  - No ternary "cond ? expr1 : expr2" operator in Scala
  - Expr may be a block expressions
  - Type of if expression is "greatest common base type" (may be Any)

```
scala> val (a,b) = (1,2)
a: Int = 1
b: Int = 2
scala> val max = if(a > b) a else {b}
max: Int = 2
scala> if(true) { println(1) } else 1
1
res0: AnyVal = ()
```

Control Expressions

- **While-loop**

  `"while" "(" BooleanExpr ")" Expr`

  – is an expression of type Unit
- **Do-loop**

  `"do" Expr while "(" BooleanExpr ")"`

  – is an expression of type Unit

```
scala> var i = 0
scala> val res = while (i < 10) i += 1
scala> println(res)
()
scala> res.isInstanceOf[Unit]
res1: Boolean = true
scala> i
res2: Int = 10
```

Control Expressions

- **For-comprehension**

  `"for" "(" Generators")" ["yield"] Expr`

  - without yield: is an expression of type Unit
  - with yield: is an expression of the type of the first Generator (approx.)

```
scala> for(i <- 1 to 4) { print(" " + i) }
 1 2 3 4
scala> for(i <- List(1,2)) { println(i) }
1
2
scala> val q = for(i <- 1 to 10 if i%2==0) yield (i*i)
q: scala.collection.immutable.IndexedSeq[Int]
= Vector(4, 16, 36, 64, 100)
scala> for(i <-1 to 8 by 2; j <- 1 until i) print(i,j)
(3,1)(3,2)(5,1)(5,2)(5,3)(5,4)(7,1)(7,2)(7,3)(7,4)(7,5)(7,6)
```

Introduction to Scala

- Overview
- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

Classes

- **Object model of Scala is similar to Java's one**
  - Abstract and final classes
  - Single inheritance of classes
  - Classes may be nested
- **Members**
  - Values (var / val)
  - Methods (def)
  - Types (type)
    - ⇒ In an abstract class, all these members may be abstract (also types)!
    - ⇒ Default visibility is public
- **Every class has a primary constructor which is always called**
  - Auxiliary constructors may be defined

Classes

```
class CreditCard(val number: Int, var limit: Int) {
   def this(number: Int) = this(number, 1000) // aux constrctor
   println("new card created")  // executed in primary constructor
   private var sum = 0
   def buy(amount: Int) {
     if(sum + amount > limit) throw new RuntimeException
     sum += amount
   }
   def remainder = limit-sum   // method which does not take
}                              // parameters
```

```
scala> val c = new CreditCard(1234)
new card created
c: CreditCard = CreditCard@284ac3
scala> c.limit = 2000
scala> c.buy(1700)
scala> c.remainder
res8: Int = 300
```

Methods

- `"def" id "(" ParameterList ")" [":" Type] = Expr`
  - Expr may be a single expression or a block expression
  - Result of method call is
    - Value of expression or
    - Value of argument of a return statement
  - If block contains a return statement, then result type must be specified
  - If method is defined recursively, then result type must be specified

```
def add(x: Int, y: Int): Int = {return x+y}
def add(x: Int, y: Int): Int = return x+y
def add(x: Int, y: Int) = x+y
def add(x: Int, y: Int) = {val z = x; z+y }

def max(x: Int, y: Int) = if(x>y) x else y
def fact(x: Int):Int = if(x==0) 1 else x * fact(x-1)
```

Methods

- Multiple results may be returned with the help of tuples

```
def quorem(m: Int, n: Int) : (Int, Int) = (m / n, m % n)
def quorem(m: Int, n: Int) = (m / n, m % n)
```

- Parameters may have default values

```
def add(m: Int, n: Int = 1) = m + n
```

- Parameter may be called by name; named args must follow positional args

```
scala> quorem(n = 2, m = 4)
res1: (Int, Int) = (2,0)
```

- Parameter lists may be separated (curried)

```
def add(m: Int)(n: Int) = m+n
```

```
scala> add(2){5}
res2: Int = 7
```

Inheritance

```
abstract class Base(param: String) {
   def doSomething: String            // without body abstract
   val value = param.toInt
   override def toString()= { "^" + super.toString()  }
}

class Derived extends Base("0") {
   def doSomething = "working" // override may be added
   override val value = 20
}
```

```
scala> val x = new Derived()
x: Derived = ^Derived@615dc4
scala> x.value
res47: Int = 20
scala> x.doSomething
res48: String = working
```

Inheritance

```
class Vehicle(val id: Int, val year: Int) {
   override def toString() = s"ID: $id Year: $year"
}

class Car(override val id: Int, y: Int, var fuelLevel: Int)
                                              extends Vehicle(0, y) {
   override def toString() =
      s"${super.toString()} FuelLevel: $fuelLevel";
}
```

```
scala> new Vehicle(1, 1998)
res0: Vehicle = ID: 1 Year: 1998

scala> new Car(2, 2010, 100)
res1: Car = ID: 2 Year: 2010 FuelLevel: 100
```

Inheritance

```
abstract class X {
   val size: Int
   println("X() " + size)
}

class Y extends X {
   val size = 5
}
```

```
scala> new Y()
X() 0
res0: Y = Y@7b641e81
scala> res0.size
res1: Int = 5
```

Inheritance

```
abstract class X {
   def size: Int;
   println("X() " + size)
}

class Y extends X {
   def size = 5
}
```

```
scala> new Y()
X() 5
res2: Y = Y@3603dd0a
```

Singleton Objects

```
class Color(val r: Int, val g: Int, val b: Int)
```

```
object ColorFactory {
   private val cols = Map(
      "red" -> new Color(255,0,0),
      "blue" -> new Color(0,0,255),
      "green" -> new Color(0,255,0))

   def getColor(color: String) =
      if(cols contains color) cols(color) else null
}
```

```
scala> ColorFactory.getColor("red")
res0: Color = Color@1e3d24a
scala> ColorFactory.getColor("pink")
res1: Color = null
```

Singleton Objects

- **Singleton Properties**
  - Singleton can be accessed by its name
  - Name of singleton represents the single instance
  - Singleton can be passed to functions with parameter ColorFactory.type

```
def createColors(f: ColorFactory.type){
   val c = f.getColor("red")
   println(c.r, c.g, c.b)
}
```

```
scala> createColors(ColorFactory)
(255,0,0)
```

Companion Objects

```
class Color private (val r: Int, val g: Int, val b: Int)

object Color {
   private val cols = Map(
      "red" -> new Color(255,0,0),
      "blue" -> new Color(0,0,255),
      "green" -> new Color(0,255,0))

   def getColor(color: String) =
      if(cols contains color) cols(color) else null
}
```

- Classes and companion objects have no boundaries, they can access the private fields and methods of each other
- In the above example the constructor of Color is private, i.e. new instances can only be accessed using method getColor

Companion Objects

```
class Color private (val r: Int, val g: Int, val b: Int)

object Color {
   private val cols = Map(
      "red" -> new Color(255,0,0),
      "blue" -> new Color(0,0,255),
      "green" -> new Color(0,255,0))

   def apply(color: String) =
      if(cols contains color) cols(color) else null
}
```

```
scala> Color("red")
res0: Color = Color@1623820
scala> Color("pink")
res1: Color = null
```

Introduction to Scala

- Overview
- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

Function Values

- **Functions are first-class citizens**

```
scala> val add = (m: Int, n: Int) => m+n
add: (Int, Int) => Int = $$Lambda$1012/2143233788@4962b41e

scala> add(2,3)
res0: Int = 5
```

```
scala> people.filter((p:Person) => p.age >= 18)
scala> people.filter(p => p.age >= 18)
scala> people.filter(_.age >= 18)

scala> object X {def vollJaehrig(p: Person) = p.age >= 18 }
defined object X
scala> people.filter(X.vollJaehrig)
res1: List[Person] = List(Person@1124cfc, Person@7c4246)
```

Function Values

- **Functions are instances of class FunctionX (X=0..22)**
  - Classes FunctionX contain methods which can be applied on function instances

```
scala> val add = (m: Int, n: Int) => m+n
add: (Int, Int) => Int = $$Lambda$1022/196340990@5ec88f9
scala> add.apply(1, 3)   // same as add(1,3)
res1: Int = 4
scala> val addc = add.curried
addc: Int => (Int => Int) = scala.Function2$$Lambda@1086
scala> val inc = addc(1)
inc: Int => Int = scala.Function2$$Lambda$1083/2010@11abd6c
scala> inc(5)
res2: Int = 6
scala> val addt = add.tupled
addt: ((Int, Int)) => Int = scala.Function2$$Lambda@219
scala> addt( 1 -> 2)
res3: Int = 3
```

Function Values

- **Curried Definitions**
  - Functions can also be defined in curried form

```
scala> val add = (x: Int) => (y : Int) => x+y
add: Int => (Int => Int) = $$Lambda$1105/828629051@6aa18912
scala> add(1)
res1: Int => Int = $$Lambda$1106/2034790200@7b364f47
scala> add(1)(2)
res2: Int = 3
```

  - Fortunately Scala supports type inference

```
scala> val add : Int => Int => Int =
                                  (x: Int ) => (y : Int) => x+y
add: Int => (Int => Int) =  $$Lambda$1122/737434492@2559
```

```
scala> val add : (Int, Int) => Int = _+_
add: (Int, Int) => Int = $$Lambda$1125/193362025@7b6b8cea
```

Function Values

- **Curried Functions**
  - Methods can be converted to a partially applied functions with "_"

```
class X {
   def add(x: Int)(y: Int) = x+y
}
```

```
scala> val x = new X()
x: X = X@196a21e
scala> x.add(5)(7)
res0: Int = 12
scala> x.add(5)_
res1: Int => Int = $$Lambda$1135/1230346437@6e4ac3f5
scala> res1(2)
res2: Int = 7
scala> x.add _
res3: Int => (Int => Int) = $$Lambda$1153/855914030@78b293a
```

Function Values

- **Functions are instances of class FunctionX (X=0..22)**
  - A function can also be defined as a subclass of class FunctionX

```
scala> object add extends Function2[Int, Int, Int] {
  def apply(x: Int, y: Int) = x+y
}
defined object add
scala> add(3, 4)
res1: Int = 7
scala> add.tupled((5,6))
res2: Int = 11
scala> add.tupled(5,6)
res3: Int = 11
scala> val add1 = add.curried(1)
add1: Int => Int = scala.Function2$$Lambda$1086
scala> add1(7)
res4: Int = 8
```

Introduction to Scala

- Getting Started
- Scala Types
- Control Expressions
- Classes
- Functions
- Pattern Matching

Pattern Matching

- **Match expression**
  - `Expr "match" "{" CaseClauses "}"`

```
def patternMatching(i : Int) = {
   i match {
      case 0 => "Null"
      case 1 => "One"
      case _ => "?"
   }
}
```

  - No fall-through
  - case _ is default pattern
  - Match throws an error if no pattern matches

Proposal for Java [April 2017]
http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html

Pattern Matching

- **Types and Guards**
  - Patterns may define types and guards
  - Patterns may contain variables which are bound
    - Pattern variables start with a lower-case letter

```
def patternMatching(any : Any) =
   any match {
      case i : Int => "Int: " + i
      case "s"     => "String: s"
      case s : String => "String: " + s
      case d : Double if d > 0 => "Positive Double: " + d
      case any => any.toString
   }
```

Pattern Matching

- **Matching Lists**

```
def length(list: List[Any]) : Int = {
   list match {
      case List()  => 0
      case x :: xs => 1 + length(xs)
   }
}
```

- **Matching Tuples**

```
def process(input: Any) = {
   input match {
      case (a,b)  => printf("Processing (%d,%d)...\n", a, b)
      case "done" => println("done")
      case _      => null
   }
}
```

Pattern Matching with Case Classes

- **Case Classes have additional features**
  - The new keyword is not mandatory to create instances of these classes
  - Getter functions are automatically defined for the constructor parameters
  - Default definitions for methods equals and hashCode are provided
  - Default definition for method toString is provided
  - Instances of these classes can be decomposed through pattern matching

```
abstract class Tree
case class Sum(x: Tree, y: Tree) extends Tree
case class Prod(x: Tree, y: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Pattern Matching with Case Classes

- **Simple Evaluator**

```
def eval(t: Tree, env: Map[String,Int]) : Int = t match {
   case Sum(x, y)  => eval(x, env) + eval(y, env)
   case Prod(x, y) => eval(x, env) * eval(y, env)
   case Var(n)     => env(n)
   case Const(v)   => v
}
```

- **Simple Test (2x+7y)**

```
scala> val t = Sum(Sum(Var("x"),Var("x")),
                   Prod(Const(7),Var("y")))
t: Sum = Sum(Sum(Var(x),Var(x)),Prod(Const(7),Var(y)))

scala> eval(t, Map("x"->8, "y"->3))
res0: Int = 37
```

"If I were to pick a language to use today other than Java, it would be Scala."

-- James Gosling, creator of Java

Source: http://jonasboner.com/2009/01/30/slides-pragmatic-real-world-scala.html

I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky [...] back in 2003 I'd probably have never created Groovy.

-- James Strachan, creator of Groovy

Source: http://macstrac.blogspot.ch/2009/04/scala-as-long-term-replacement-for.html

Teach Yourself!

- http://docs.scala-lang.org/
- https://docs.scala-lang.org/tour/tour-of-scala.html
- http://twitter.github.io/scala_school/
- http://www.scala-lang.org/api/current/
- https://docs.scala-lang.org/cheatsheets/index.html
- https://scala-lang.org/files/archive/spec/2.13/

Stackoverflow Developer Survey

- https://insights.stackoverflow.com/survey/2019#top-paying-technologies