# Functional Data Structures

*Lecture 020 of Advanced Programming*

*Feb. 7, 2018*

**Andrzej Wasowski & Zhoulai Fu**

**IT University of Copenhagen**

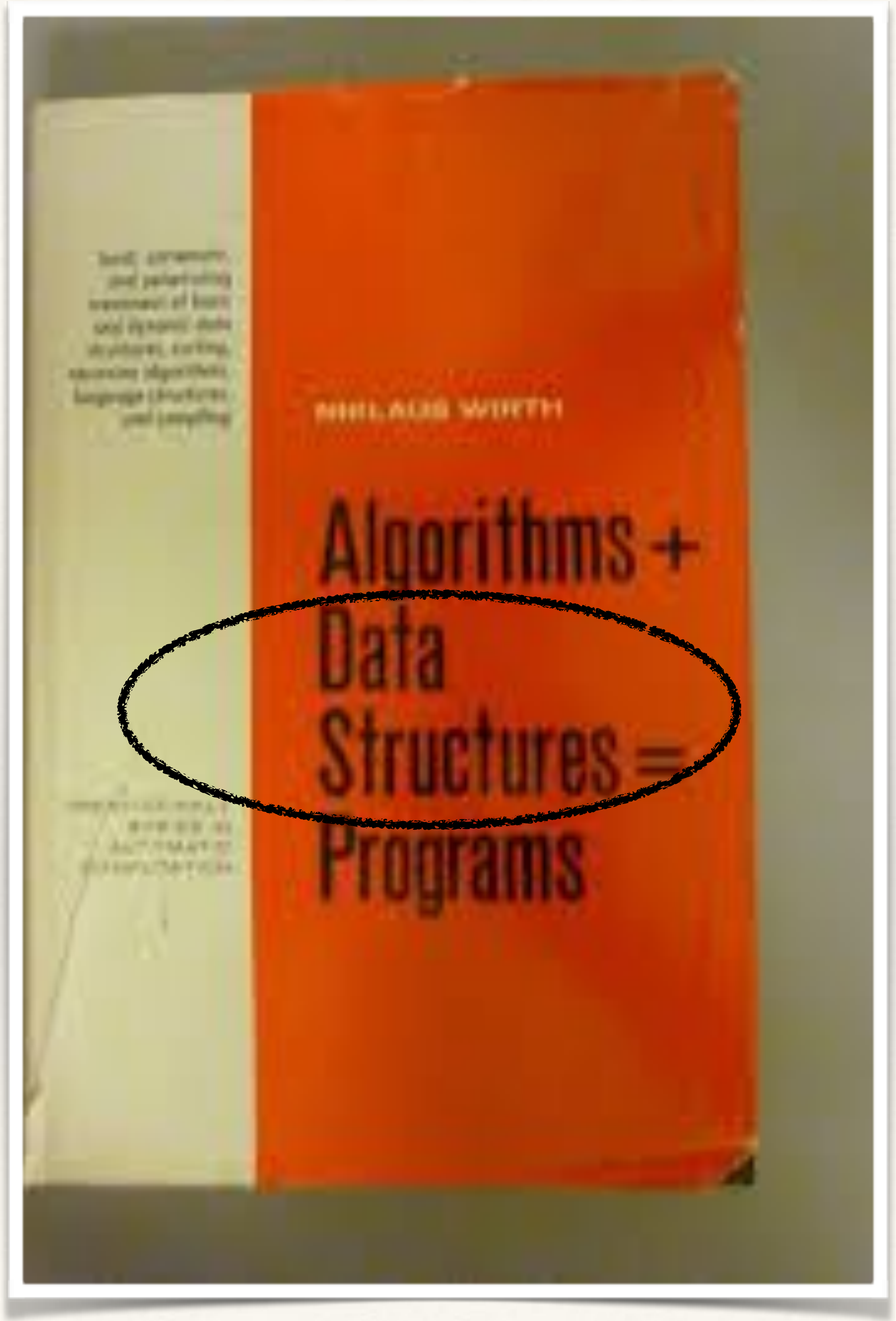
## Know About Your Teacher

- First name: Zhoulai. Just call me “July”
- Before teaching this class:
    - Graduate of Ecole Polytechnique, France
    - Ph.D in *Static Program Analysis* from INRIA, France
    - Postdoc at University of California, Davis, USA
    - Assistant Professor of Computer Science,  ITU (since Sep. 2017)
- My objective for teaching this class:
    - You achieve a good understanding of Scala (for your exams, CV, skill set)
    - You enjoy it!

# Today's topic: Functional Data Structure

Part I. Concepts about Functional Data Structures

Part II  Important Details



## Part 1: Concepts

- *Data structure is about how your data are constructed and operated*
- Question 1: How you *construct* functional data structures?
- Question 2: How you *operate* on functional data structure?

# Algebraic Data Types (ADT)

An ADT is just a data type defined by one or more data constructors, each of which may contain zero or more arguments. We say that the data type is the *sum* or *union* of its data constructors, and each data constructor is the *product* of its arguments, hence the name *algebraic* data type.

❖ [Chiusano *et al*.] Page 44

## Example: ListInt as a functional data structure

- Nil is a ListInt
- Cons(head, tail) is a ListInt if head is an Int and tail is an ListInt

# Quiz 1: ListDouble

- Nil is a ListDouble
- Cons(head, tail) is a ListDouble if head is a Double and tail is a ListDouble

DEMO

# Quiz 2: ListString

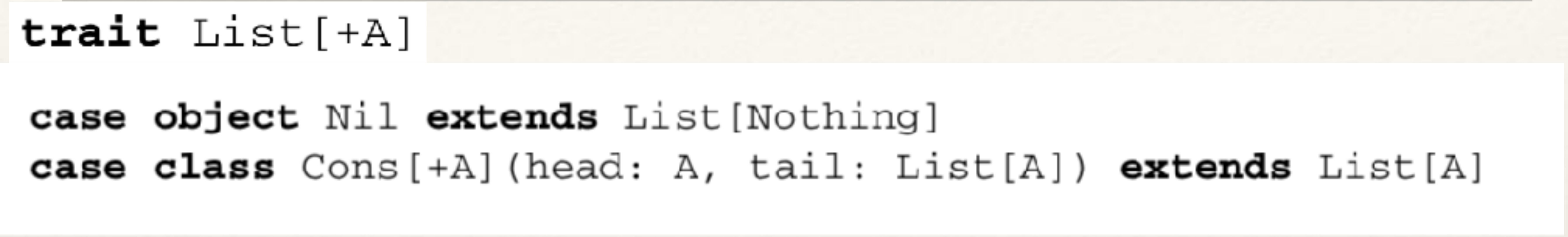
- Nil is a ListString
- Cons(head, tail) is a ListString if head is a String and tail is an ListString

DEMO

## Suppose your next quiz is to construct ListChar

- Too simple, actually boring!
- Time to talk about *Generic ADT*
- A generic ADT comes with a parameter
- e.g. List[A]
  - A list of Ints has type List[Int]
  - A list of Strings has type List[String]
  - List[MyClass]

# Construct a Generic List



```scala
trait List[+A]

case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

- +A mean type covariance: If X is a subtype of Y, List[X] will be a subtype of List[Y]
- e.g., Put “+” if you want List[Dog] be a subtype of List[Animal]
- Here, Nil extends List[Nothing], which is a subtype of List[Int], List[String]

- … concerns about variance aren't very important for the present discussion and are more of an artifact of how Scala encodes data constructors via subtyping, so don't worry if this is not completely clear right now. It's certainly possible to write code without using variance annotations at all.

- [Chiusano *et al*.] Page 32

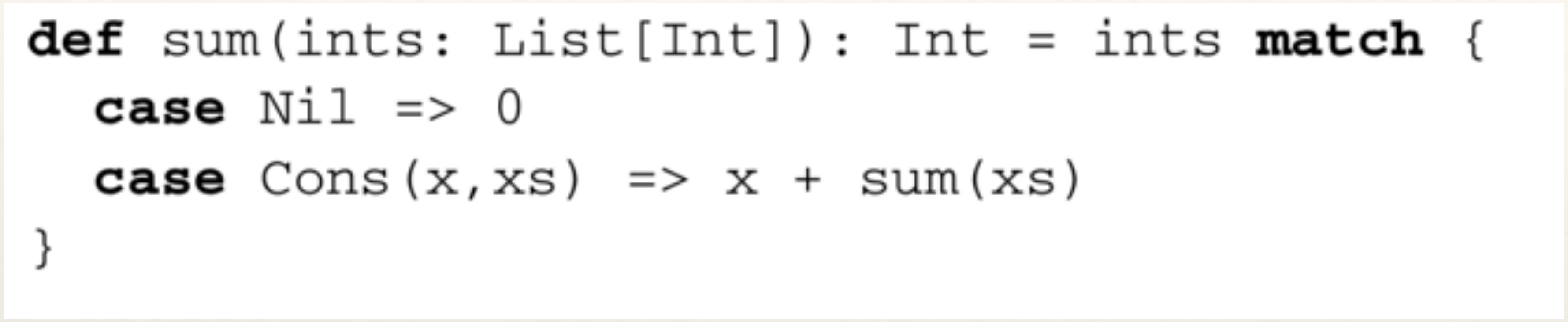
## Quiz 3 (if time permits): TreeInt

- TreeInt:
  - Nil is a TreeInt
  - A Leaf of integer is a TreeInt
  - A Node of two TreeInt is a TreeInt

## Your takeaway 1:

Algebraic Data Type constructs functional data structures

## Q2: How you operate on functional data structures?

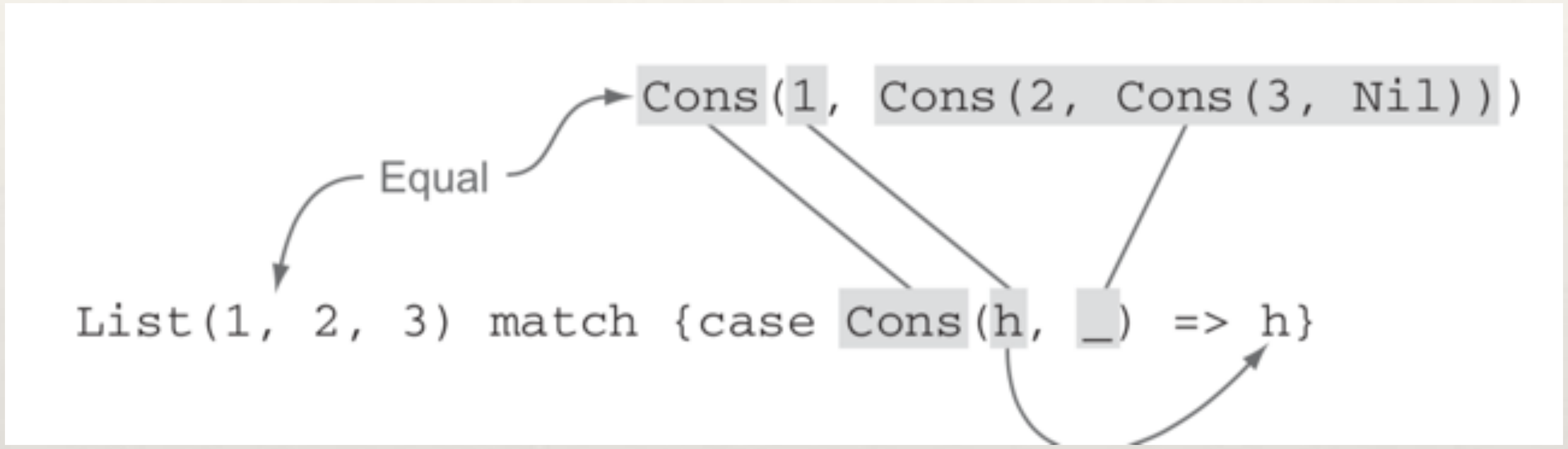


```scala
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)
}
```

- Pattern matching works a bit like a fancy switch statement that may descend into the structure of the expression it examines and extract subexpressions of that structure.

- [Chiusano *et al*.] Page 32

## Quiz 4: Value of List (1,2,3) match {case Cons(h,_) => h}


Cons(1, Cons(2, Cons(3, Nil)))
Equal
List(1, 2, 3) match {case Cons(h, _) => h}

## Quiz 5: ExprInt

- Construct an ADT to represent an integer expression like *“(2+3)*4”* (Hint: An integer expression is either a constant of integer, a Sum of two integers, a Product of two integers, etc)
- Evaluate it

DEMO

## Your takeaway 2:

Operate on your functional data structures via *pattern matching*

## Part II. :Important Details (many slides borrowed from Andrzej's)

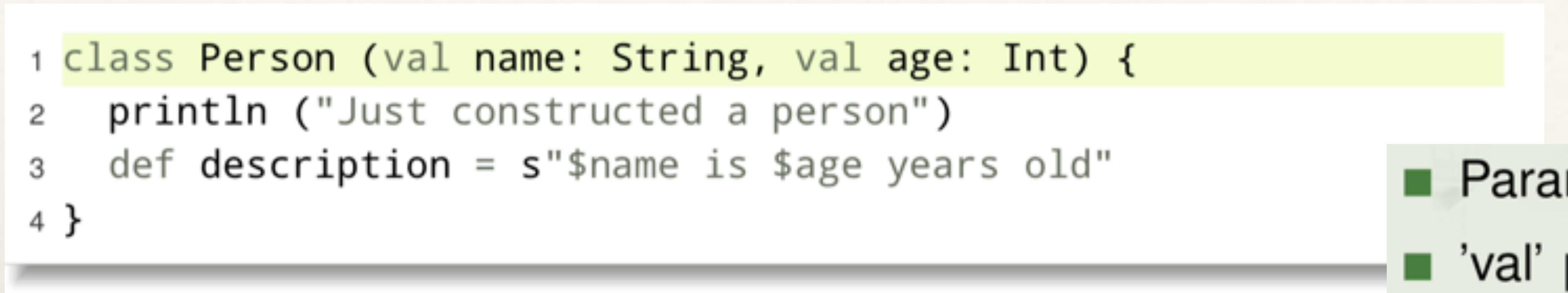
- Variadic functions
- Primary constructor
- Companion objects
- Trait
- Folding

## Syntax sugar 1: Variadic functions

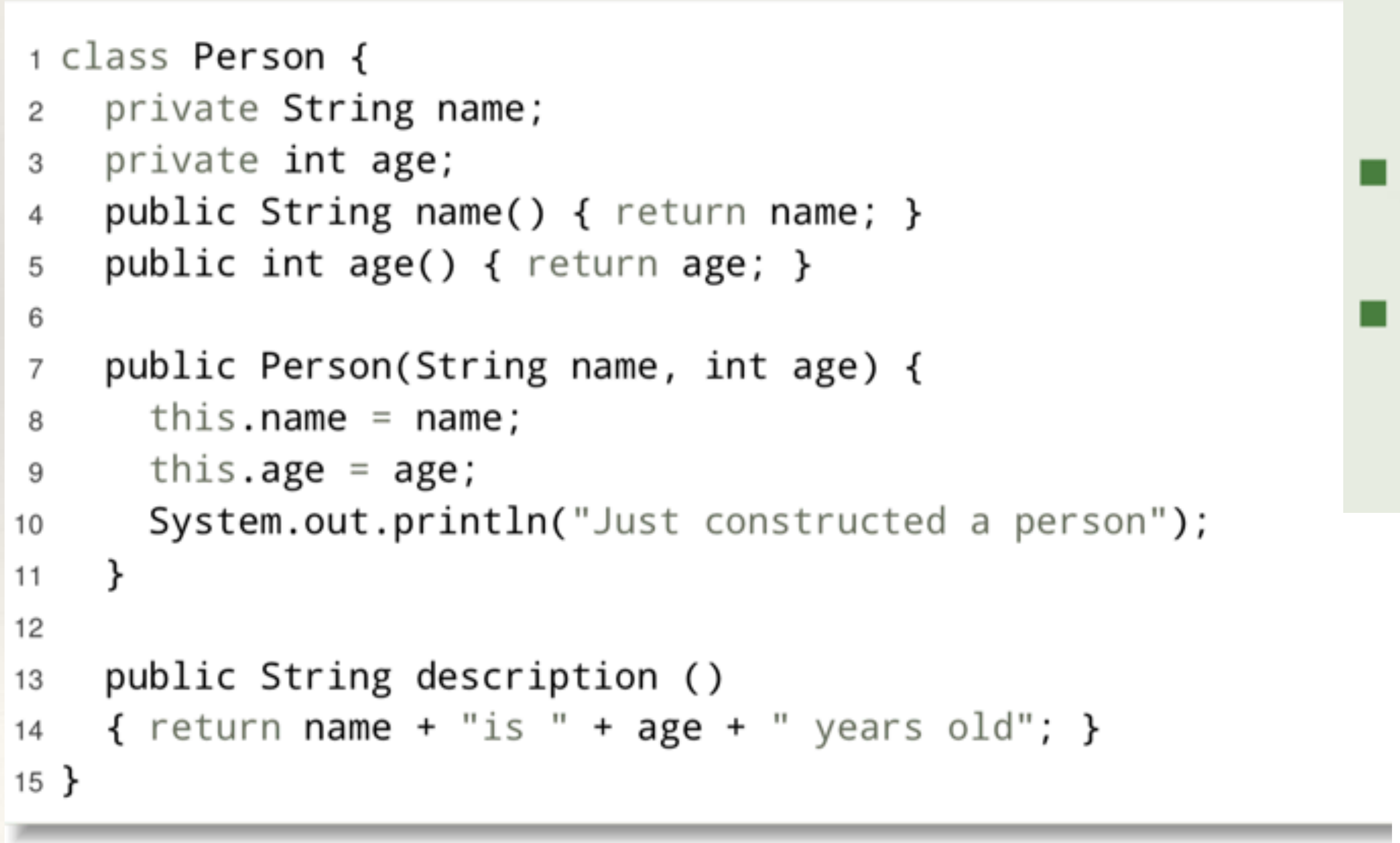
- Sometimes, it is convenient to implement a function that can take a variable number of arguments
  - Use Int* to indicate a variable number of ints
  - Use (1 to 10) :_* to convert the list to 10 ints

DEMO

# Syntax Sugar 2: The Primary Constructor



```scala
class Person (val name: String, val age: Int) {
  println ("Just constructed a person")
  def description = s"$name is $age years old"
}
```



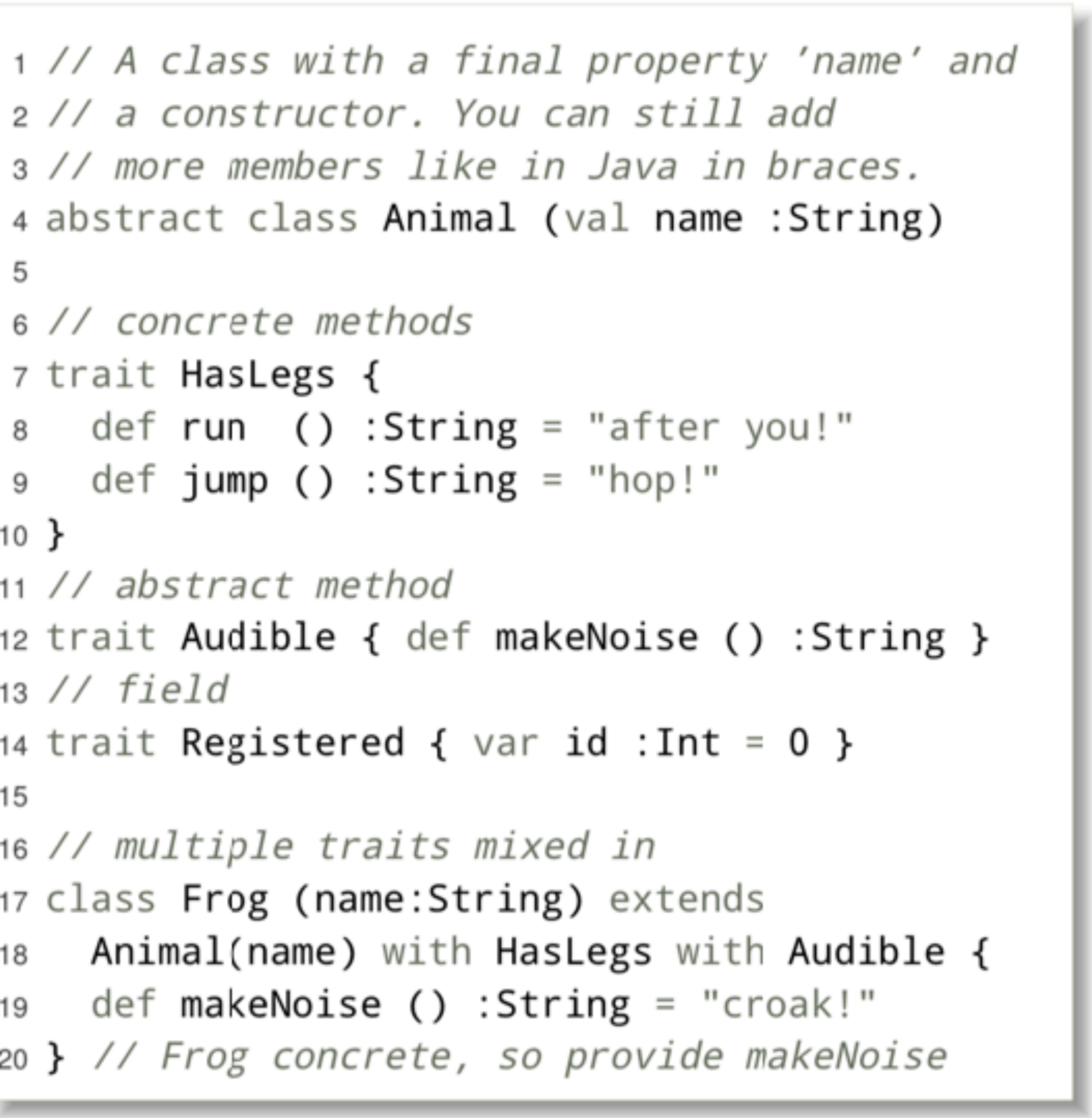
```java
class Person {
  private String name;
  private int age;
  public String name() { return name; }
  public int age() { return age; }

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
    System.out.println("Just constructed a person");
  }

  public String description ()
  { return name + "is " + age + " years old"; }
}
```

- Parameters become fields
- 'val' parameters become values, 'var' become variables
- If no parameter list, primary constructor takes none
- Constructor initializes fields and executes top-level statements of the class

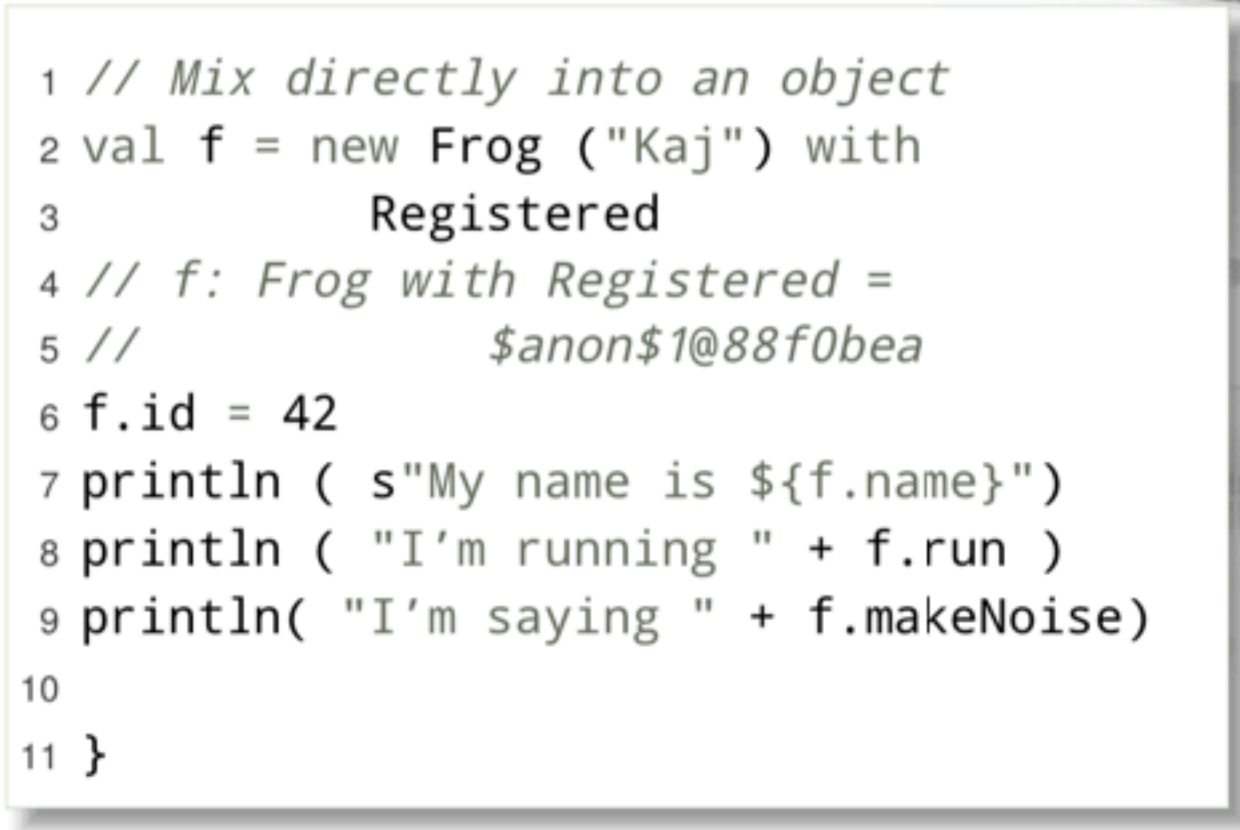
DEMO

[Horstmann 2012, Chpt. 5.7] explains the primary constructors in Scala

# Traits



```scala
// A class with a final property 'name' and
// a constructor. You can still add
// more members like in Java in braces.
abstract class Animal (val name :String)

// concrete methods
trait HasLegs {
  def run  () :String = "after you!"
  def jump () :String = "hop!"
}
// abstract method
trait Audible { def makeNoise () :String }
// field
trait Registered { var id :Int = 0 }

// multiple traits mixed in
class Frog (name:String) extends
  Animal(name) with HasLegs with Audible {
  def makeNoise () :String = "croak!"
} // Frog concrete, so provide makeNoise
```



```scala
// Mix directly into an object
val f = new Frog ("Kaj") with
          Registered
// f: Frog with Registered =
//              $anon$1@88f0bea
f.id = 42
println ( s"My name is ${f.name}")
println ( "I'm running " + f.run )
println( "I'm saying " + f.makeNoise)

}
```

Trait can specify ADT or classes of various levels of abstraction, and can "mixin" with objects

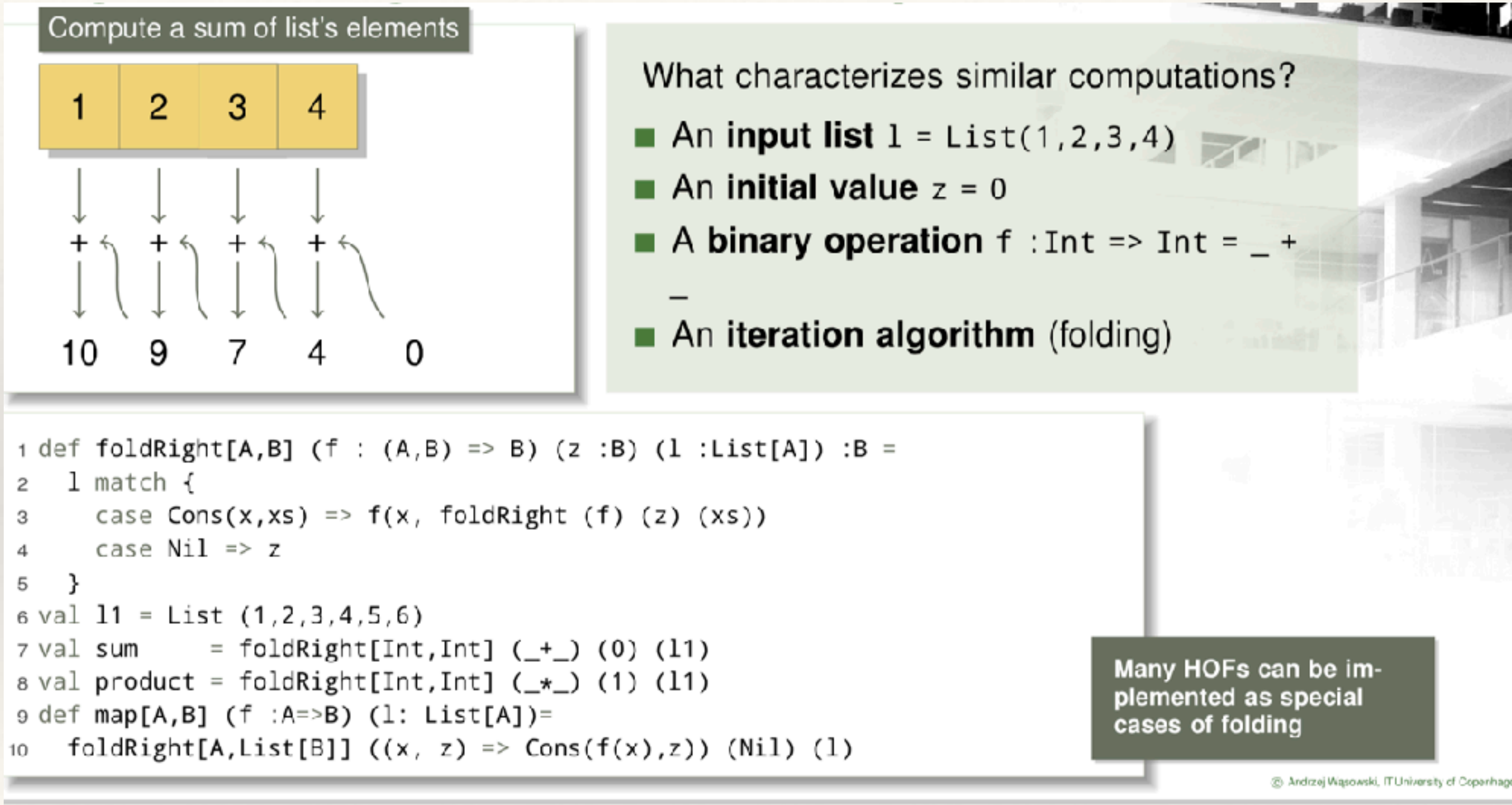
Horstmann 2012, chpt. 10], [Odersky et al. 2014, chpt. 12] have more info than [Chiusano, Bjarnason 2014]

# Companion Objects

- We'll often declare a companion object in addition to our data type and its data constructors. This is just an object with the same name as the data type (in this case List) where we put various convenience functions for creating or working with values of the data type.
- If, for instance, we wanted a function def fill[A](n: Int, a: A): List[A] that created a List with n copies of the element a, the List companion object would be a good place for it. Companion objects are more of a convention in Scala.4 We could have called this module Foo if we wanted, but calling it List makes it clear that the module contains functions relevant to working with lists.
    - [Chiusano *et al.*] Page 33

DEMO

# Folding: Loops in Functional Programming



What characterizes similar computations?

- An **input list** l = List(1,2,3,4)
- An **initial value** z = 0
- A **binary operation** f :Int => Int = _ + _
- An **iteration algorithm** (folding)

```
def foldRight[A,B] (f : (A,B) => B) (z :B) (l :List[A]) :B =
  l match {
    case Cons(x,xs) => f(x, foldRight (f) (z) (xs))
    case Nil => z
  }
val l1 = List (1,2,3,4,5,6)
val sum     = foldRight[Int,Int] (_+_) (0) (l1)
val product = foldRight[Int,Int] (_*_) (1) (l1)
def map[A,B] (f :A=>B) (l: List[A])=
  foldRight[A,List[B]] ((x, z) => Cons(f(x),z)) (Nil) (l)
```

Many HOFs can be implemented as special cases of folding

- Making a good use of Folding needs much practice.

## Conclusions

- Construct functional data structures via algebraic data types
- Operate on them with pattern matching
- Generic Types
- Traits
- Type variance
- Companion objects
- Variadic functions
- Primary constructors