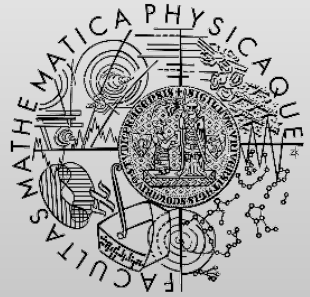
Statically-typed Class-based languages – Scala

http://d3s.mff.cuni.cz



FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

Tomas Bures

bures@d3s.mff.cuni.cz

Scala

- Statically-typed language
- Compiles to bytecode
- Modern concepts

- Example: E01

Syntax inference

- A line ending is treated as a semicolon unless one of the following conditions is true:
  - The line in question ends in a word that would not be legal as the end of a statement, such as a period or an infix operator.
  - The next line begins with a word that cannot start a statement.
  - The line ends while inside parentheses (...) or brackets [...], because these cannot contain multiple statements anyway.

- Blocks are based on indentation
  - Possible to use curly braces (version 2 syntax)

Static vs. dynamic typing

- Target function is determined
  - at compile time – static typing
  - at runtime – dynamic typing

- Example: E02

Classes vs. objects

- Scala does not have static method
- Instead it features a singleton object
  - Defines a class and a singleton instance

- Example: E03

- Decompiled – AppLogger, Logger

Type inference

- Types can be omitted – they are inferred automatically
  - At compile time

- Example: E04

Type Hierarchy

- Everything is an object
  - primitive data types behind the scene (boxing/unboxing)
- Compiler optimizes the use of primitive types
  - a primitive type is used if possible

Companion object

- A class and object may have the same name
  - Must be defined in the same source
- Then the class and object may access each others private fields
- Example: E05

Constructors

- One primary constructor
  - class parameters
  - can invoke superclass constructor
- Auxiliary constructors
  - must invoke the primary constructor (as the first one)
  - must not invoke superclass constructor

Operators

- Scala allows almost arbitrary method names (including operators)
- A method may be called without a dot
- Prefix operators have special names

- Example: E06

Flexibility in Identifiers and Operators

- Alphanumeric identifier
  - starts with letter or underscore
- Operator identifier
  - an operator character belongs to the Unicode set of mathematical symbols(Sm) or other symbols(So), or to the 7-bit ASCII characters that are not letters, digits
  - any sequence of them
- Mixed identifier
  - e.g. unary_- to denote a prefix operator
- Literal identifier
  - with backticks (e.g. `class`) to avoid clashes with reserved words, etc.

Operator precedences

- Operator precedence determined by the first character
  - Only if the operator ends with “=“, the last character is used

(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

Extensions

- Similar to C#, Scala makes it possible to declare an extension of an existing type
- The extensions have to be brought to scope
  - Typically imported
- Example: E07

Context parameters (aka givens)

- Scala allows naming instances (called “givens”) that define canonical values of certain types
  - used to synthetize arguments for context parameters
- Givens have to be brought to scope to be applicable
  - Special import notation

- Example: E08

Implicit conversions

- Scala allows specifying functions that are applied automatically to make the code correct
  - conversion to the type of the argument or to the type of the receiver
  - the conversion is brough in as a “given” – same rules apply for making it visible as for other givens

- Example: E09 + H1

Rich wrappers

- Implicit conversions used to implement so called Rich wrappers
- Standard library contains rich types for the basic ones
  - E.g. RichInt – defines methods to, until, …

Namespaces

- Scala allows groups classes to packages (similar to Java and C#)
- Similar to C#, it allows defining multiple classes and even packages in the same file

- Example: E10

First-class functions

- Functions are first-class citizens
- May be passed as parameters
- Anonymous functions, …
- Anonymous functions are instances of classes
  - Function1, Function2, …

- Example: E11

Tail recursion

- The compiler can do simple tail recursion
  - If the return value of a function is a recursive call to the function itself
- Example: E12

For-comprehension

- Generalized for-loops
  - generators, definitions, filters
- Translated to operations over collections
  - map, flatMap, withFilter, foreach
- Example: E13

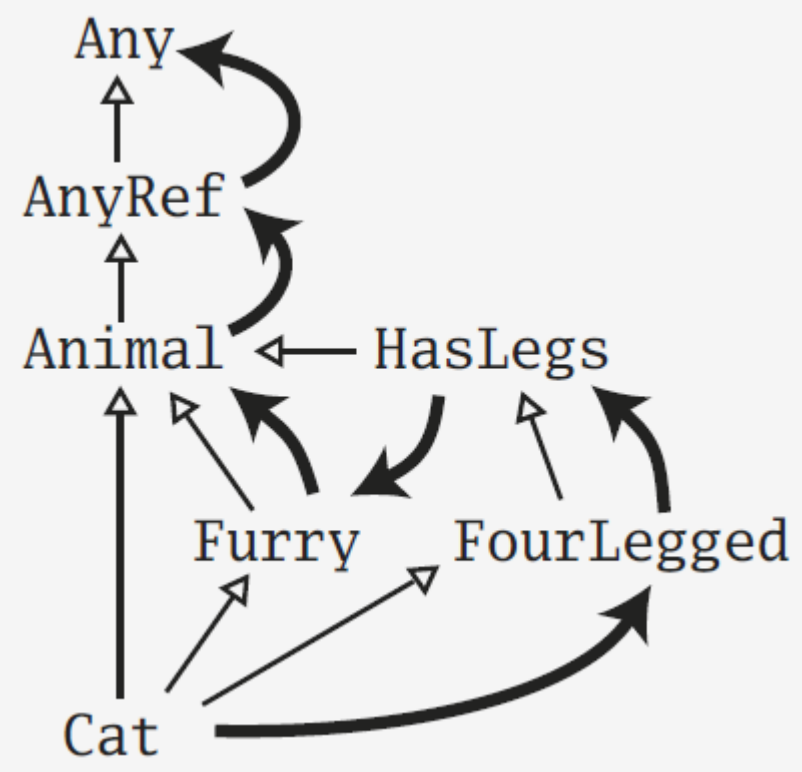
Traits

- Scala does not have interfaces
  - It has something stronger – mixins (called traits)
- A trait is like an interface, but allows for defining methods and variables

- Example: E14

Linearization

- As opposed to multiple inheritance, traits do not suffer from the diamond problem
- This is because the semantics of super is determined only when the final type is defined
- Example: E15



Composing Traits

- Composition of traits can be used to address the same problem as Dependency injection addresses
  - “Cake pattern”

- Example: E16

Abstract types

- What about if we want methods in a subclass to specialize method parameters?

- Example: E17 + H2

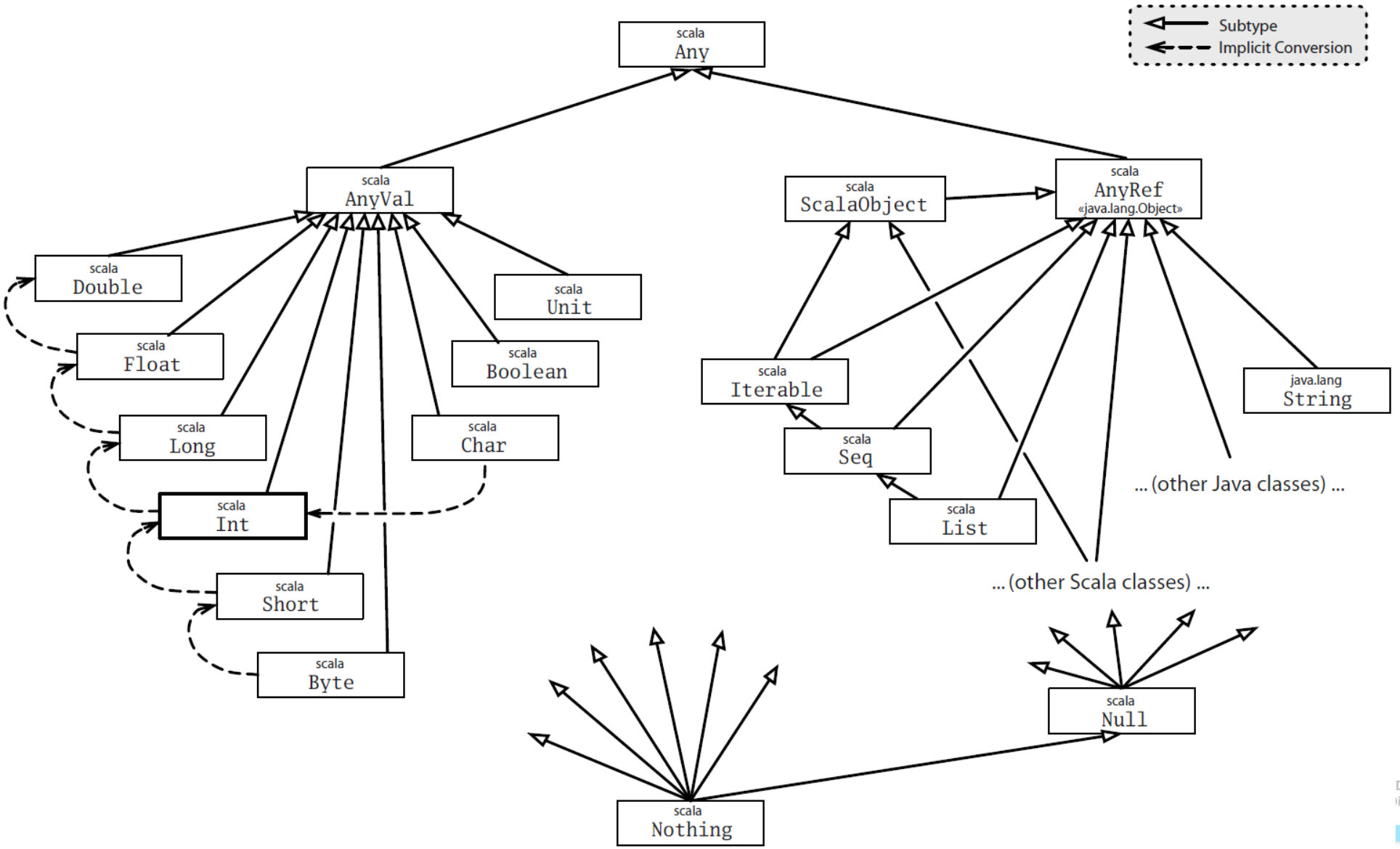
Embedding and exporting

- Alternative to composing functionalities by traits
- Instead of extending a class, an object is embedded and its methods are exported on the interface
  - Invocation of those methods act on the embedded object
- This is similar to type embedding in Go
  - But since Go lacks inheritance or mixins, the embedding (with all its limits) is the only way of composing functinality
- Example E18

Type parameterization

- Each class and method may be parameterized by a type
- Lower and upper bounds
- Example: E19

Type Hierarchy



Null and Nothing types

- **null** is singleton instance of Null
  - can be assigned to any AnyRef

- Nothing is a subtype of everything
  - Can be assigned to anything, but does not have any instance

```
def doesNotReturn(): Nothing =
  throw new Exception
```

Nothing in Use I

```
def fail(msg: String): Nothing =
      println(msg)
      sys.exit(1)


val y = if x != null then x else fail("$&#@!")
```

Nothing in Use II

```
abstract class Option[+A]:
  def isEmpty
  def get: A


case class Some[+A](x: A) extends Option[A]:
  def isEmpty = false
  def get = x


case object None extends Option[Nothing]:
  def isEmpty = true
  def get = throw NoSuchElementException()
```

Type variance

- Assuming: A <: B (A subtype of B) and X[T]
- What is the relation of **X[A]** and **X[B]** ?
  - X[A] **<:** X[B] … X is covariant in its type parameter T
  - X[A] **>:** X[B] … X is contravariant in its type parameter T
  - No relation … X is invariant w.r.t. to the type parameter T

- Example: E20

Instance private data

- The mutable state in a class typically prevents the covariance/contravariance
- Why?

- Example: E21

Collections

- Scala offers immutable and mutable variants of collections (immutable ones preferred)
  - List vs. ListBuffer
  - HashSet vs. mutable.HashSet
  - HashMap vs. mutable.HashMap

- To achieve performance, immutable types are internally often backed by mutable structures

Path dependent types

- Nested traits/classes are specific to the instance of the outer class
- This makes types different based on the instance they are tied with
  - Though this is a runtime property, it can be with some effort checked statically
- Example: E22

Structural subtyping

- It is possible to specify only properties of a type
- Example: E23