Statically-typed Class-based languages (Scala)

Department of
Distributed and
Dependable
Systems

D3S

Tomas Bures

bures@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Scala

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

- Example: E01

Semicolon 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.

Static vs. dynamic typing

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

- Example: E02

- Decompiled – DynamicTypingMain

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

Department of
Distributed and
Dependable
Systems

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.

Department of
Distributed and
Dependable
Systems

D3S

Operator precedences

- Operator precedence determined by the first character

  - Only if the operator ends with "=", the last character is used

(all other special characters)

$* \, / \, \%$

$+ \, -$

$:$

$= \, !$

$< \, >$

$\&$

$\char`^$

$|$

(all letters)

(all assignment operators)

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

  - must be in current scope or source or target type scope

  - scalac -Xprint:typer mocha.scala

    - program after implicits added and fully-qualified types substituted

- Example: E07 + H1

Rich wrappers

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

Type Hierarchy
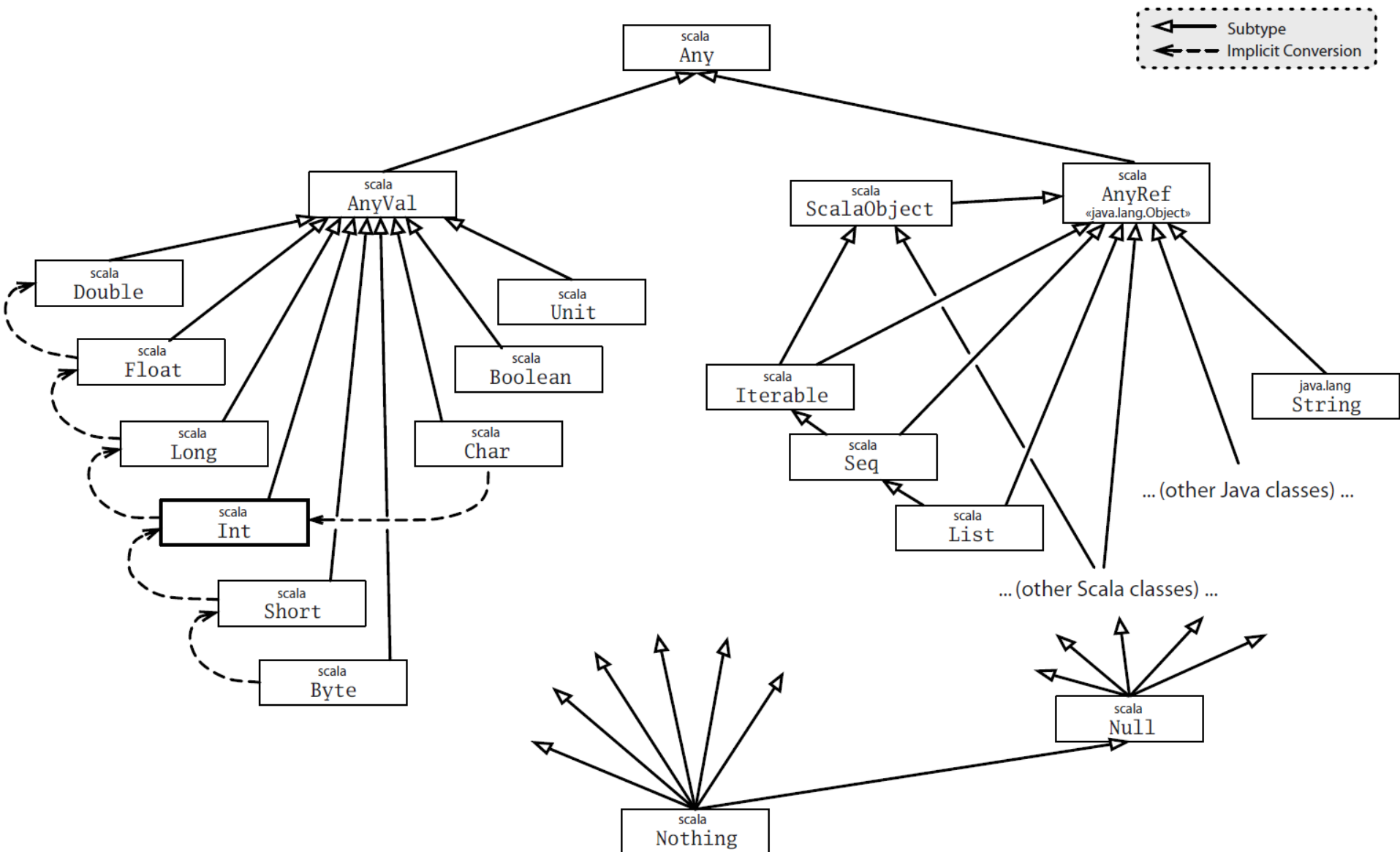
Null and Nothing types

- **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

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


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

Nothing in Use II

```scala
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 new NoSuchElementException()
}
```

Equality

- Overloading of operator "==" is used to implement equality

- Reference equality is tested by functions eq and ne

```
val s1 = "Hello"
val s2 = "World"
println("1: " + (s1 + s2 == s1 + s2)) // true
println("2: " + (s1 + s2 eq s1 + s2)) // false
```

Basic Types + Symbol Literals

- Types are pretty much the same as in Java

- Symbol literals
  - Similar to constant strings, but represented as instances of class Symbol
  - Possible to compare them by reference

- Example: E08

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: E10

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: E11
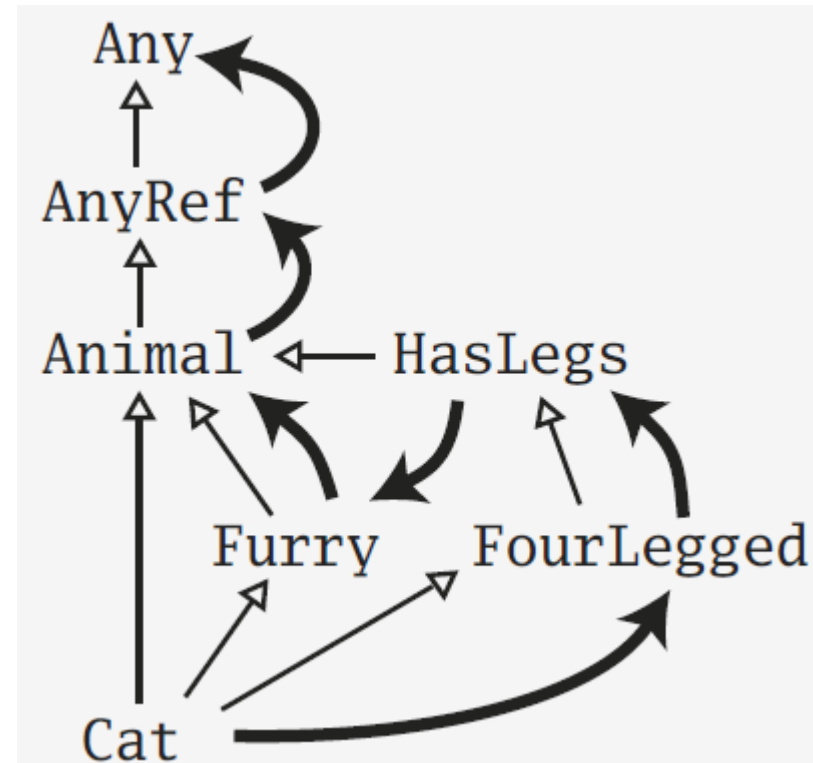
Composing Traits

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

- Example: E12 + test

Type parameterization

- Each class and method may be parameterized by a type

- Lower and upper bounds

- Example: E14

Instance private data

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

- Example (covariance): E15

- Example (contravariance): E16

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: E17

Abstract types

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

- Example: E18 + H2

Structural subtyping

- It is possible to specify only properties of a type

- Example: E19

First-class functions

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

- Example: E20

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: E21

For-comprehension

- Generalized for-loops
  - generators, definitions, filters

- Translated to operations over collections
  - map, flatMap, withFilter, foreach

- Example: E22

New control structures

- Currying – function that returns function

- By-name parameters
  - omitting empty parameter list in an anonymous function

- Example: E23

Behavior Driven Development

- Unit test as a specification

- Human readable style

- Still executable

- Example: E24

Case-classes, pattern matching

- Scala allows for simple pattern matching (similar to Prolog terms)

- Case-classes

  - factory method (no new necessary)

  - all parameters are vals

- Example: E25

Case sequences & Partial functions

- Functions may be defined as case sequences
  - It's like a function with more entry points

- Since the case sequence does not have to cover all cases, it yields a partial function
  - Partial function may be queried if a given value is in its domain
  - or it throws a runtime exception if called with an unsupported input argument

- Example: E26 + H3

Extractors

- Pattern matching relies on method unapply
  - apply – takes arguments, returns an object
  - unapply – takes an object, returns a tuple or sequence of arguments

- Allows creating custom matchers

- Example: E27, E28

String interpolation

- String interpolation implemented by rewriting code at compile time
- Standard interpolators
  - s – interpolator
    ```
    val name = "reader"
    println(s"Hello, $name!")
    ```
  - raw – interpolator
    ```
    println(raw"No\\\\escape!") // prints: No\\\\escape!
    ```
  - f – interpolator (printf-like formatter)
    ```
    f"${math.Pi}%.5f"
    ```

- Custom interpolators can be defined
- Example: E09

Regular Expressions

- Regular expressions are normal library class
- They can be used as extractors
  - Even more interesting when combined with string interpolation

- Example: E29

Scala – Java interoperability

- trait T
  - interface T – method declarations
  - class T$class – method implementations
- class C extends T
  - instance methods of C
  - delegate methods to methods of T$class
- object C
  - static methods in C
    - delegate to methods of C$.MODULE
  - class C$
    - instance methods of C
    - static field C$.MODULE of type C (the singleton instance)

- Example: E13

Delayed init

- If a class extends the DelayedInit trait
  - the compiler turns the class initializer to a function
  - and calls delayedInit function on the class instance giving it the initializer function

- Example: E30
  - Use of DelayedInit to implement the App trait

XML

- Scala has native support for parsing XML
- XML can be included where expression is expected
- It gets transformed to a runtime structure

- Example: E31