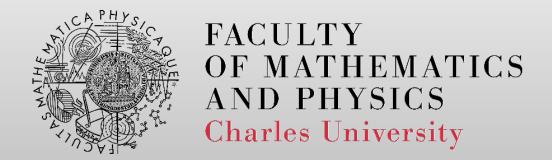
Statically-typed Class-based languages – Scala

http://d3s.mff.cuni.cz





Tomas Bures

bures@d3s.mff.cuni.cz

Scala

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



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)



Comparison: Code Blocks

- 1D syntax
 - Relies on explicitly delineated blocks
 - Curly braces (Java, C, C#, ...)
 - Begin/end (Pascal)
- 2D syntax
 - Relies on indentation (Scala 3, Python, YAML)

- Block does not have to corresponding to variable scope
 - Python, "var" in JavaScript use function scope



Static vs. dynamic typing

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



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



Comparison: Types of methods

- Instance methods i.e. qualified by this
 - Virtual vs. non-virtual
 - E.g. instance methods in Java, any method in Scala
 - Python:
 def foo(self, ...)
- Class methods i.e. qualified by class
 - Python:
 @classmethod
 def foo(cls, ...)
- Static methods i.e. without qualification
 - E.g. static methods Java
 - Python:
 @staticmethod
 def foo(...)



Type inference

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



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

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



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)
```

Comparison: Operator overloading

- Operators typically use syntax different to normal functions, thus they cannot be completely freely defined
- Typically, there is some limited support for their overloading:
 - No overloading e.g. Java
 - Limited overloading
 - Using a dedicated keyword e.g. C# or C++
 public static Complex operator +(Complex c1, Complex c2) ... C#
 Complex operator + (Complex const &obj) ... C+
 - Using a dedicated name e.g. Python def __add__(self, other)
 - Instance method (C++, Python) vs. static method (C#)
 - Ability to create new operators with some restriction on their names
 - Operators treated almost as regular functions e.g. Scala, Smalltalk
 - Requires flexible syntax that can infer the "dot" between the receiver and message aSum := aPoint + aSmallInt ... sends message "+" with parameter aSmallInt to object aPoint



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

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



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 brought in as a "given" same rules apply for making it visible as for other givens

- This is similar to C# implicit operator
 - public static implicit operator byte(Digit d) => d.digit;

• 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, ...



Comparison: Extending functionality of existing classes

- A typical problem: 3 + aRational
- Possible solutions:
 - Extension methods e.g. C#, Scala

```
public static int add(this int lhs, Rational rhs) ... C# (but as of 2022 no "extension operators")
```

- extension (lhs: Int)
 def + (rhs: Rational) = Rational(lhs) + rhs ... Scala
- Implicit conversion e.g. Scala
 - given Conversion[Int, Rational] = new Rational(_)
- Right-hand side operators e.g. Python
 - def __radd__(self, other)
 - Note tha Python has similar functions also for in-place updates (x += y) e.g. Department and department of the properties of the

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



First-class functions

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



Tail recursion

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

For-comprehension

- Generalized for-loops
 - generators, definitions, filters

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



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

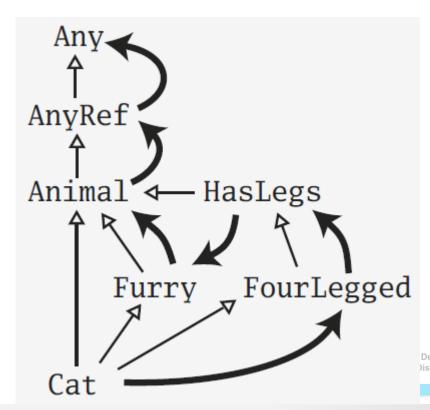


Linearization

 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



Comparison: Composing behavior

- Multiple inheritance
 - Without linearization e.g. C++
 - Difficulties connected with diamond inheritance problem
 - With linearization e.g. Python
 - Similar possibilities as with Scala, but certain problems are not there because Python is a dynamically typed language with limited abstractions for declaring classes
- Default methods in interfaces
 - Java, C#
 - No support for constructors, linearization (i.e., stackable behavior), no possibility of adding fields (i.e. state)
- Traits e.g. Scala, Groovy
 - Support for constructors, fields, and stackable behavior



Note on Terminology: C++ "Traits"

- The term "trait" in C++ has a different meaning
- Connected with templates, it means common characteristics of types
- E.g. std::numeric_limits

```
template<class T>
T findMax(const T const * data, const size_t const numItems) {
   T largest = std::numeric_limits<T>::min();
   for(unsigned int i=0; i<numItems; ++i)
      if (data[i] > largest) largest = data[i];
   return largest;
}
```

C++ Custom Trait Definition

```
template< typename T >
struct is_void{
 static const bool value = false;
template<>
struct is void< void >{
 static const bool value = true;
```

Composing Traits

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



Abstract types

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

Comparison: Type Parameters vs. Type Variables

- Type can be "passed" to a class
 - via a type parameter connected with a constructor
 - via a type assignment from a descendant

```
abstract class Animal:
type SuitableFood <: Food</pre>
```

class Cow extends Animal:
 override type SuitableFood = Grass

• • •

val c: Animal = new Cow()

```
abstract class Animal[SuitableFood]
```

class Cow extends Animal[Grass]

• • •

val c: Animal[_] = new Cow()



This Type

- Represent the type of the descendant in the parent class
- Can be regarded as an implicit abstract type
- Useful for defining chainable methods



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
 - Also known as "Composition over inheritance"
- 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 functionality



Example: Embedding in Go

```
// Usage
// Types
type Base struct {
                                                co := Container{Base: Base{b: 10}, c: "foo"}
 b int
                                                // OR
type Container struct {
                                                co := Container{}
 Base c string
                                                co.b = 1
                                                co.c = "string"
func (base Base) Describe() string {
                                                // Calling a method declared on the Base
 return fmt.Sprintf("base %d", base.b)
                                                fmt.Println(cc.Describe())
```

New control structures

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



String interpolation

- String interpolation implemented by rewriting code at compile time
- Standard interpolators
 - s interpolator

 val name = "reader"

 println(s"Hello, \$name!")
 raw = interpolator
 - raw interpolator
 println(raw"No\\\escape!") // prints: No\\\escape!
 - f interpolator (printf-like formatter)
 f"\${math.Pi}%.5f"
- Custom interpolators can be defined
- Example: E21



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



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



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: E24, E25

Regular Expressions

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



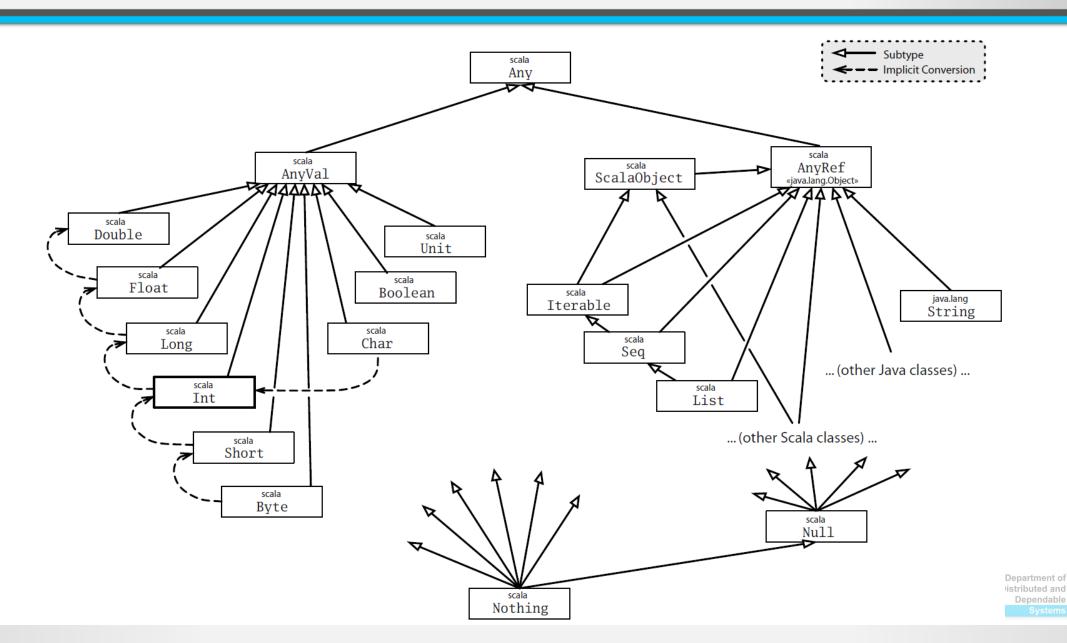
Type parameterization

• Each class and method may be parameterized by a type

Lower and upper bounds



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</p>
 - 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



Instance private data

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

Why?

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