LAAI 2 - Scala Concepts Reference

Originally intended for module 2 of “Language and Algorithms for Artificial Intelligence” at UNIBO.

Last update: November 19, 2021
D.M.

! QUITE PRELIMINARY VERSION !

Contents

(*) Starred paragraphs are not strictly part of the program.

[1][**REPL & COMPILATION**] "Read-Evaluate-Print Loop" is avail in Scala, as usual for functional languages. The "evaluate" part is demanded to the interpreter. Alternatively, Scala can be compiled (see *scalac* and *sbt* commands.)

[2][**EXPRESSIONS EVALUATION**] Non-primitive expr.s are evaluated by evaluating the operands (from the **leftmost**) of the **leftmost** operator. This happens recursively until a primitive expression results.
A name is evaluated by replacing it with the RHS of its definition.
A function application is evaluated by replacing the call with the RHS of the definition of the f. itself, in a CBV way (see §3). For this reason, evaluation can be infinite: `def loop:Int = loop+1`

[3][**F. APPLICATIONS EVALUATION STRATEGIES**] 1) **CBV** (Call-By-Value): the formal parameters appearing in the body are replaced by the (evaluated) actual parameters;
2) **CBN** (Call-By-Name): the formal parameters appearing in the body are replaced by the correspondent expression in the function application.
CBV only evaluates each expression once; CBN has the advantage that each expression in the application is only evaluated if it's used. Both strategies, if terminate, result in the same value only when there are no side-effects involved.
In Scala, CBV is used for the parameters unless the type declaration is defined with an arrow (note the leading space): `def f(x:Int, y: =>Int) ...` (x uses CBV, y CBN).

[4][**VALUE DEFINITIONS**] Creates a value (immutable variable):

```
val x = loop
```

Creates a variable (mutable):

```
var x = loop
```

Creates an "alias" (0-ary method):

```
def x = loop
```

The latter is the only one that delays the evaluation of loop. Another way for delaying is to use a *lazy val* (var cannot be lazy), that is a val whose value is computed the first it is required:

```
def test = println("called")
lazy val a = test
a
a
// outputs "called"
def b = test
b
b
// outputs "called\ncalled"
```

[5][**SCOPING, BLOCKS**] Scoping blocks are defined using curly braces. Inside-block definitions shadow external definitions.
Being themselves expressions, blocks assume the value of their last expression.

[6][**HOF AND LAMBDAS**] Higher-Order Functions are functions that accept as a parameter or return another function.
A function type is indicated as $arg\_type \Rightarrow return\_type$; for ex: `Int => Int` .
It is possible to define **anonymous/lambda functions** with the same "arrowed" syntax: `x => x*x*x` .
See §7 for examples of lambda ff; see also §27.

```
scala> val nf = (hof:Int=>Int, x:Int) => (hof(x)+2)
// output:
// val nf: (Int => Int, Int) => Int = $Lambda$1080/0x00000008405f9840@4c86da0c
```

[7][**EXAMPLES OF LAMBDA FUNCTIONS**] Two examples of use of lambda functions:

```
val ints = List(1, 2, 3)

ints.map((i: Int) => i * 2)
ints.map((i) => i * 2) // type can be inferred
ints.map(i => i * 2) // only one arg: parenthesis are not needed
ints.map(_ * 2) // the param. appears only once, thus can be anonimized

ints.foreach((i: Int) => println(i)) // same:
ints.foreach(println(_))
ints.foreach(println)
ints foreach println
```

[8][**CURRYING**] To instatiate at different times the parameters of a function, we can rewrite the original f. as a f. that returns a partial application of the original f. with the first(s) argument(s) fixed. This procedure is known as currying. Scala provides a specific notation to facilitate this:

```
def general(f: (Int,Int)=>Int) = (a:Int,b:Int) => f(a,b)
val sum = general((a,b)=>a+b)

// This is equivalent to:
def general(f: (Int,Int)=>Int)(a:Int, b:Int) = f(a,b)
val sum:(Int,Int)=>Int = general((a,b)=>a+b)

print(sum(5,4))
```

[9][**EXAMPLE: FIXED-POINT COMPUTATION**] (see slides for mathematical background)

```
import Math.abs

def fixPoint(maxIter: Int)(f: Double=>Double) = {
    def iter(x:Double, n:Int):Double = {
        if (n==0) throw new Exception("maxIter")
        val v = f(x)
        if (abs(v-x)<1e-10) v else iter(v,n-1)
    }
    iter(1.1f, maxIter)
}

lazy val line = fixPoint(100)((x)=>1+x/2.0)
lazy val sqrt2 = fixPoint(10)((x)=>(2/x+x)/2)

println(f"$sqrt2%.5f")

// usage:
// $ scala fixp.scala
// 1,41421
```

[10][**CLASSES: BASICS**] "Classes in Scala are blueprints for creating objects. They can contain methods, values, variables, types, objects, traits, and classes which are collectively called members." Simplest declaration and usage:
`class User; val user1 = new User`
A class definition may have parameters. "If a formal parameter declaration x:T is preceded by a val or var keyword, an accessor (getter) definition for this parameter is implicitly added to the class. If the introducing keyword is var, a setter for that parameter is also implicitly added to the class."
`class Point(x: Int, y: Int) // x,y not accessible`
`class Point(val x: Int, val y: Int)`
`val point = new Point(1, 2)`
Parameters for which a value is specified are OPTIONAL. When an object whose class has optional parameters is created, the order of the parameters can be rearranged.
`class Point(var x: Int = 0, var y: Int = 0) .`
`val point = new Point`
`val point = new Point(4,3)`
`val point = new Point(y=3, x=4)`
Classes declaration may contain a body. Every contained member (fields, methods...) is public by default.
As shown by the ex. below, unary operators can be defined for the objects of the class and used in an infix fashion; for example `new Rational(1) max r` is the same as `(new Rational(1)).max(r)` . **this** identifies the current object.
Note the presence of spaces after the name of variables that end with _, needed to avoid ambiguity for the interpreter. The *override* keyword will be commented in §11.

```
class Rational(private val x_ :Int, private val y_ :Int = 1) {
    val x = x_ /gcd(x_,y_)
    val y = y_ /gcd(x_,y_)
    private def gcd(a:Int, b:Int):Int = if (b==0) a else gcd(b, a%b)
    def unary_- = new Rational(-x,y)
    def + (r:Rational) = new Rational(x*r.y + y*r.x, y*r.y)
    override def toString = s"($x, $y)"
    def max(r:Rational) = if (x*r.y > y*r.x) this else r
}

val r = new Rational(4,2)
println(r)
println(-r)
println(r + new Rational(15,45))
println(new Rational(1) max r)

// outputs:
// (2, 1)
// (-2, 1)
// (7, 3)
// (2, 1)
```

[11][**CLASS HIERARCHIES (ABSTRACT, EXTENDS)**] Abstract classes are declared with **abstract** as a first keyword and are allowed to include methods without the body; see the ex. below.
To implement the abstract methods, a class that extends the abstract one must be declared, for ex.
`class Empty extends IntSet {/*implementation*/}` or
`class NonEmpty(elem:Int, link:IntSet) extends IntSet {/*implementation*/}` ,
in which the body of the abstract methods is present.
In the extending class is also possible to override an already-defined method by using the **override** keyword (see "toString" method in the ex. of §10).

```
abstract class IntSet {
    def incl(x:Int):IntSet
    def contains(x:Int):Boolean
}
```

[12][**SINGLETON OBJECTS**] "An object is a class that has exactly one instance. It is created lazily when it is referenced, like a lazy val." Object are declared as classes using **object** instead of **class**. Objects cannot have formal parameters in their definiton.

```
object obj { def pr(x:String) = print(s"INFO: $x") }
```

Being a singleton, the use of the *new* keyword is not required:

```
obj.pr("pippo") // outputs: INFO: pippo
```

[13][***COMPANION OBJECTS**] An object declared inside a file of a class that has the same name as the class is said to be a CO.
A CO and its class can access each other's private members.
COs are commonly used to implement the possibility of creating an object without the **new** keyword. As a matter of facts, Scala translates `Person("Fred Flinstone")` as `Person.apply("Fred Flinstone")` , so that the purposely-crafted method of the companion object of Person can act as a factory method, creating the instances of Person:

```
class Person {
    var name = ""
}

object Person {
    def apply(name: String): Person = {
        var p = new Person
        p.name = name
        p
    }
}
```

[14][**ABOUT TYPES**] Objects and classes define types. Thanks to type inference, a value assumes the type of the object assigned to it, unless the type is explicitly specified (this specification is called **tag**). A value may contain a subtype of the type specified by its tag. To get the class (tag) of an object or value v, use: **v.getClass**.
To get the actual type of an object or value at runtime, import *scala.reflect.ClassTag* then use:

```
def f[T](v: T)(implicit ev: ClassTag[T]) = ev
```

More about this in [3].
An explicit casting between object types is possible using **asInstanceOf[Class]**; see code below.

```
scala> class Animal; class Dog extends Animal
// ...
scala> val fido = new Dog
// val fido: Dog = Dog@6070775e
scala> val fido:Animal = new Dog
// val fido: Animal = Dog@693f2213
scala> (new Dog).asInstanceOf[Animal]
// val res1: Animal = Dog@693f2c89
scala> (new Animal).asInstanceOf[Dog]
// java.lang.ClassCastException: [...]
```

[15][**BINDING AND OVERLOAD RESOLUTION**] Scala adopts **dynamic/late binding** to bind the method's name to its implementation. Since a variable of type Ob could contain an object of a subtype of Ob, say S, the implementation of the method called on Ob may be either the original implementation (Ob's) or the one from S (that, being a subtype, can override Ob's implementation.) Dynamic binding decides which implementation to use at runtime, when the content of the variable is known (instance of Ob or S).
Therefore we can say that Scala has late binding for the method receiver (the object on which we call the method). On the contrary, dynamic binding is associated with **static overload resolution**: the type of the objects as specified by the tag (at compile time) is used to select the method among the overloaded ones. The example below illustrate both concepts.

```
class A {
    def greet(name:String) = println(s"Hello $name")
}
class B extends A {
    override def greet(name:String) = println(s"You stupid $name")
    def boh(o:A) = println("A")
    def boh(o:B) = println("B")

}
class C extends B {
    override def boh(o:A) = println("Ac")
    override def boh(o:B) = println("Bc")
}

val a:A = new B
a.greet("ciccio")
// You stupid ciccio

val b:B = new C
a.asInstanceOf[B].boh(a)
b.boh(a)
// A
// Ac
```

[16][**TRAITS**] Traits are used to share interfaces, implementations and fields between classes. They can be seen as "rich interfaces". Classes and objects can extend multiple traits (while only one class), but traits cannot be instantiated and therefore have no parameters. This differentiates them from abstract classes.
Technically, Traits are **mixins**, classes that contains methods for use by other classes without having to be the parent class of those other classes.
As for classes, traits define types (cfr §14). Classes and traits that inherit from traits use the the keyword **extends** before the first trait, then the list of the remaining traits, each one precede by the keyword **with**; see ex. below [for the meaning of T, see 20] and §17. The order in which traits are specified is used for binding resolution in case of conflicting overrides: the latter is preferred to the formers; see §18
A specific aspect of traits is the **dynamic binding of "this" and "super"**: since those keywords cannot be statically resolved when parsing the trait code, they are is dynamically bound to the mixed-in class; see §17 for an example. Particularly, to bind super the **linearization** process is needed (§18).

```
trait TotOrder[T] {
    def compare(r:T):Double
    def > (r:T) = (this compare r)>0
    [...]
}
// Extends one class and one trait:
class C1(x:Int, y:Int) extends Rational(x,y) with TotOrder[OrdRat] { [...] }

// Only one trait:
class C2(x:Int, y:Int) extends TotOrder[OrdRat] { [...] }
```

[17][**TRAITS EXAMPLE**] Example that illustrates the use of traits, including the binding of "this"

```
trait Order[T] {
    def compare(E:T):Int
    def size:Int
    def >(E:T):Boolean  = (this compare E)>0
}

trait NiceRepr { def repr = s"{${this.toString}}" }

abstract class IntSet { def size:Int }

object Empty extends IntSet with Order[IntSet] {
    override def toString = ""
    def size = 0
    def compare(E:IntSet) = if (E.size==0) 0 else -1
}

class NonEmpty(i:Int, s:IntSet) extends IntSet with Order[IntSet] with NiceRepr {
    override def toString = i +"," + s.toString
    def size = 1+s.size
    def compare(E:IntSet) = this.size - E.size
}

object SomethingElse extends NiceRepr

val s1 = new NonEmpty(3, Empty)
val s2 = new NonEmpty(2, s1)

println(s1>s2, s2>s1, s1>Empty)
// outputs: (false,true,true)

// two examples to illustrate that "this" is dynamically binded in NiceRepr definition:
println(s2.repr)
println(SomethingElse.repr)
/* outputs:
* {2,3,}
* {Main$$anon$1$SomethingElse$@37eeec90}
*/
```

[18] **[*LINEARIZATION PROCEDURE]** Class/Traits hierarchy form a DAG that is linearized as showed in fig.1.

Let C be a class with template C_1 with ... with C_n { *stats* }`. The *linearization* of C, $\mathcal{L}(C)$ is defined as follows:

$$\mathcal{L}(C) = C, \mathcal{L}(C_n) \vec{+} \ldots \vec{+} \mathcal{L}(C_1)$$

Here $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left operand:

$$\begin{array}{rcll} a, A \vec{+} B & = & a, (A \vec{+} B) & \textbf{if } a \notin B \\ & = & A \vec{+} B & \textbf{if } a \in B \end{array}$$

Figure 1: formal definition of the linearization procedure.

For example, the following code is linearized as shown in fig.2

```
trait A
trait B extends A
trait C extends A
trait D extends B with C
```

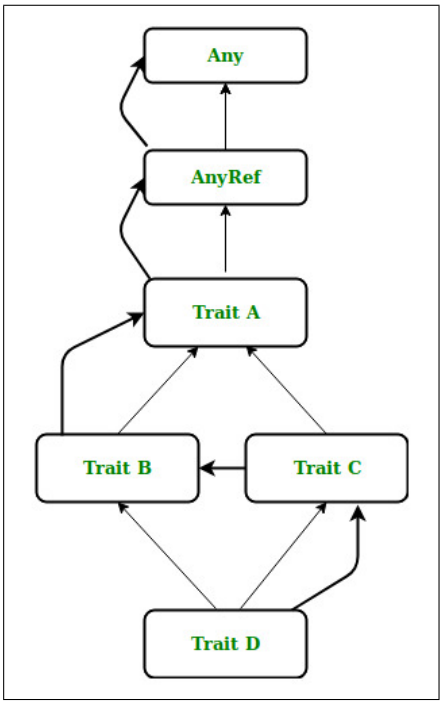


Figure 2: Bold arrows represent the linearization of the DAG.

[19][**BUILT-IN TYPES HIERARCHY**] The hierarchy in appendix A is built-in in Scala; the fundamental templates are (see the code below for the included methods):

Any base type of all types

AnyRef base type of all reference (non-primitive) types; alias of **java.lang.Object**

AnyVal Base type of all primitive types

Nothing Subtype of every other type; It assumes no value (trait) and It's used as a type for *throw* expressions to signal abnormal termination.

Null Subtype of every reference type; It assumes the value **null**.

```
abstract class Any {
    def equals(that: Any): Boolean
    def hashCode(): Int
    def toString(): String
    final def getClass(): Class[_] = sys.error("getClass")
    final def ==(that: Any): Boolean = this equals that
    final def != (that: Any): Boolean = !(this == that)
    final def ## : Int = sys.error("##")
    final def isInstanceOf[T0]: Boolean = sys.error("isInstanceOf")
    final def asInstanceOf[T0]: T0 = sys.error("asInstanceOf")
}


abstract class AnyVal extends Any {
    def getClass(): Class[_ <: AnyVal] = null
}


trait AnyRef extends Any {
    def equals(that: Any): Boolean = this eq that
    def hashCode: Int = sys.error("hashCode")
    def toString: String = sys.error("toString")
    def synchronized[T](body: => T): T = sys.error("synchronized")
    final def eq(that: AnyRef): Boolean = sys.error("eq")
    final def ne(that: AnyRef): Boolean = !(this eq that)
    final def ==(that: Any): Boolean =
        if (this eq null) that.asInstanceOf[AnyRef] eq null else this equals that
    protected def clone(): AnyRef
    protected def finalize(): Unit
    final def notify(): Unit
    final def notifyAll(): Unit
    final def wait (): Unit
    final def wait (timeout: Long, nanos: Int): Unit
    final def wait (timeout: Long): Unit
}


sealed trait Nothing


sealed trait Null


// Singleton is used by the compiler as a supertype for singleton types.
// This includes literal types, as they are also singleton types.
final trait Singleton extends Any
```

---

[20][**GENERIC CLASSES**] Generic (or polymorphic) Classes allow one or more type in the class definition to be parametric, in order to define "generic" methods and objects; for example, we can declare a class that contains a value of a type T and then let the interpreter infer the type T from the instantiation of an object of the class :

```
scala> class C[T](val v:T)
// class C
scala> new C(3)
// val res5: C[Int] = C@72eed547
scala> (new C[Double](3)).v
// val res6: Double = 3.0
```

[21][**POLYMORPHISM**] Subtyping and generic types are two mechanism to implement polimorphism, ie. the use of the same symbol to refer to different functions or objects. The interplay among these forms is explained in the next par.s.

[22][**TYPE BOUNDS**] Scala allows the programmer to express upper and lower bounds in the hierarchy for the object type that can instantiate a generic template.
For example, assume *IntList* has two different subclasses, Empty and NonEmpty. We would like to correctly tag the definition of an identity function *id* so that when id is applied to an object of type Empty the returned type remains Empty; the same for NonEmpty. Instead of:

```
def id(l: IntList ): IntList
```

we can impose the same type for l and the returned value, plus an upper bound:

```
def id[T <: IntList](l:T):T
```

This has to be interpreted as "T must be subsumed by IntList". Similarly we can impose a lower bound using `>:` .

[23][**VARIANCE**] A parametrized type C[T] for which `A<:B` implies `C[A]<:C[B]` is said **covariant**.
This is the same as interpreting the subtyping relation as suggested by the **Liskov substitution principle**: a property provable for objects of type B is also provable for objects of a subtype of B ($\forall A | A <: B$).
For ex, we would like to be able to use a List[Int] in every place in which List[Double] is allowed. This is reasonable if the List structure is **immutable**, otherwise type checking can be broken. [TO EXTEND]
A parametrized type C[T] for which `A<:B` implies `C[B]<:C[A]` is said **contravariant**.
A parametrized type that is not variant is said to be **nonvariant**.
In Scala it is possible to declare variant classes:

```
class C[+A] { ... } // C is covariant
class C[-A] { ... } // C is contravariant
class C[A]  { ... } // C is nonvariant
```

[24][**CASE CLASSES**] CCs have all the functionalities of a standard class, plus some addiotional ones useful for functional programming:
constructor parameters are public val fields by default, so **accessor methods** are generated for each parameter;
an **apply** method is created in the companion object of the class, so the new keyword is unneeded;
an **unapply** method is generated, which is useful for using CCs in **pattern matching** (see §25, an example is in §26);
copy, equals, hashCode, toString are auto-generated too, see [4] for details.

[25][**PATTERN MATCHING - SYNTAX**] Pattern matching can be used to decompose object according to their structure. The general syntax is: `e match {case p1 => e1 ... case pn => en}` , where e is the expression to be decomposed (**selector**), pi a possible pattern and ei the expression to be evaluated if pi matches with e. Each case is also called an **alternative**.
A pattern is built from constants, constructors, variables and type tests. Pattern matching tests whether a given selector (value or sequence of values etc.) has the shape defined by a pattern, and, if it does, binds the variables in the pattern to the corresponding components of the selector. The same variable name may not be bound more than once in a pattern.
An expression e may match one of the following patterns:

variable pattern - x A simple variable that matches every expr. (only once, then is binded and matches that value); a special case is the **wildcard pattern**: irrelevant variables can be replaced by _, that is treated as a fresh variable at each occurrence.

typed pattern - x:T consists of a pattern variable x and a type pattern T; the type of x is T.

pattern binder - x@p It only matches the expressions of the type implied by the pattern p (p implies a type if it only matches expressions of that type).

literal pattern - L only matches if e==L.

interpolated string pattern It has the form of an *interpolated string* (see §28); for ex. id"text0 pat1 text1 ... patn textn" is interpreted as if It was: StringContext("text0",...,"textn").id(pat1,...,patn).

stable identifier pattern Represented by a a stable identifier (a previously-declared variable or field , such as x, X or o.x); note that there is a partial syntactic overlap with variable patterns, so that lowercase variables must be backquoted in order to be considere s.i.p. instead of variable patterns (uppercase are considere s.i.p. by default). [see code below]

constructor pattern a pattern in the form: $c(p_1, \ldots, p_n)$ where $n \geq 0$ and c is a stable identifier denoting a *case class* (see §24); the pattern matches all objects created from constructor invocations $c(v_1, \ldots, v_n)$ where each element pattern p_i matches the corresponding value v_i.

tuple pattern - $(p_1, \ldots, p_n)$ matches a tuple of patterns.

extractor pattern similar to constructor pattern for specific kind of objects; see [1]

infix operation pattern "p op q" is a shorthand for the constructor or extractor pattern op(p,q)

pattern alternative - $p_1, \ldots, p_2$ Specifies alternatives that may not bind variables other than wildcards. The alternative pattern matches a value e if at least one its alternatives matches e.

XML patterns see: [2]

```
// stable identifier pattern example
class C { c =>
    val x = 42
    val y = 27
    val Z = 8
    def f(x: Int) = x match {
        case c.x => 1  // matches 42
        case `y` => 2  // matches 27
        case Z   => 3  // matches 8
        case x   => 4  // matches any value (variable pattern)
    }
}
```

[26][**PATTERN MATCHING - EVALUATION**] The first pattern that matches the selector causes the correspondent expressions block to be evaluated, with the variables in the block properly replaced by the correspondent subexpressions in the pattern.

```
trait Expr {
    def eval(indent:Int):Int = {
        def iprint[T](el:T) = println("  "*indent+el)
        this match {
            case N(n) => iprint(this); n
            case Sum(l,r)  => iprint(this); l.eval(indent+1) + r.eval(indent+1)
            case Prod(l,r) => iprint(this); l.eval(indent+1) * r.eval(indent+1)
        }
    }
    def eval:Int = this.eval(0)
}

case class N(n:Int) extends Expr
case class Sum(e:Expr, f:Expr) extends Expr
case class Prod(e:Expr, f:Expr) extends Expr

println(new Prod(new Sum(N(3),N(-5)),N(7)).eval)
```

The output is:

```
Prod(Sum(N(3),N(-5)),N(7))
  Sum(N(3),N(-5))
    N(3)
    N(-5)
  N(7)
-14
```

[27][**ANONIMOUS FUNCTIONS WITH CASE**] To create a lambda function, pattern matching can be used:

```
scala> val v :(Int=>Any) = {case i if i%2==0 => (i,"even"); case i => (i,"odd")}
[...]
scala> 1 to 5 map v
val res2: IndexedSeq[Any] = Vector((1,odd), (2,even), (3,odd), (4,even), (5,odd))
```

[28][**STRING INTERPOLATION**] [rewrite and EXTEND] ...

```
val name = "James"
s"Hello, $name"  // Hello, James
s"1 + 1 = ${1 + 1}"
s"New offers starting at $$14.99"  // escaping
"""{"name":"James"}"""
f"$name%s is $height%2.2f meters tall"
raw"a\nb"  // similar to s interpol., no escaping

// Custom si. see:
// https://docs.scala-lang.org/overviews/core/string-interpolation.html#advanced-usage
```

[29][**LISTS**] Lists are Sequences available in the standard library. They are:
1) immutable;
2) nestable;
3) homogeneous (all the elements must be of the same type T, where the list type is List[T]).
List are built using the constructor **List** or the **construction operator ::**:
`List(1,2,3) == 1::2::3::Nil // true`
`List() == Nil // true`
Note that the cons operator is right-associative and takes an element on the left and the rest of the list on the right.
Some basic methods are: **head, tail, isEmpty, length, ++, :+, reverse**.

```
scala> List(1,2,3) :+ 4
// val res1: List[Int] = List(1, 2, 3, 4)

scala> List(1,2,3) ++ List(4)
// val res2: List[Int] = List(1, 2, 3, 4)
```

Tail-recursive implementation of insertion sort:

```
type T = Int

def place(e:T, l1:List[T], l2:List[T]) :List[T] = l2 match {
    case Nil => (e::l1).reverse
    case h::t  => if (e>h) place(e,h::l1,t) else l1.reverse++(e::l2)
}


def isort: List[T]=>List[T] = {
    case h::t => place(h, Nil, isort(t))
    case _ => Nil
}

println(isort(List(4,5,1,9,7,-1,-34,67)))
// outputs: List(-34, -1, 1, 4, 5, 7, 9, 67)
```

[30][**TUPLES**] Tuples are available in Scala up to 22 member elements. They are:
1) immutable;
2) nestable;
3) heterogeneous.
A n-tuple is implemented in "scala.Tuple**n**" and its i-th element is accessed with **._i**.
Instances of Tuple2 are also called **pairs**.

```
scala> val t = new Tuple3[Int,Double,String](1,2,"ciao")
// val t: (Int, Double, String) = (1,2.0,ciao)

// Note that 2 is specified as a double here, int wouldn't work:
scala> val (x, y:Double, s) = (1,2.0,"ciao")
// val x: Int = 1
// val y: Double = 2.0
// val s: String = ciao

scala> (x,y,s) == t
// val res63: Boolean = true
```

[31][**SOME HOF FOR LISTS**] .

map	**final def** map[B](f: (A) => B): List[B]
filter	**def** filter (p: (A) => Boolean): List[A]
filterNot	**def** filterNot (p: (A) => Boolean): List[A]
partition	**def** partition (p: (A) => Boolean): (List[A], List[A])
takeWhile	**final def** takeWhile(p: (A) => Boolean): List[A]

dropWhile	`def dropWhile(p: (A) => Boolean): List[A]`
span	`final def span(p: (A) => Boolean): (List[A], List[A])`
reduceLeft	`def reduceLeft[B >: A](op: (B, A) => B): B`
foldLeft	`def foldLeft[B](z: B)(op: (B, A) => B): B`

[32][**VECTORS**] Vs are linear structure similar to lists with a balanced access to each member, indipendently from the position.
They provide random access and updates in O(log n) time, as well as very fast append/prepend/tail/init (amortized O(1), worst case O(log n)). Vectors are implemented by radix-balanced finger trees of width 32.
Vs are:
1) immutable;
2) nestable;
3) homogeneous.
Creation: `Vector(1,2,3)`
They support the same operations as lists (except for ::, that is replaced by +: and :+, with the COLon going to the COLlection side) plus some additional operations exploiting indexing, such as:

```
def indexOf[B >: A](elem: B, from: Int): Int
def updated[B >: A](index: Int, elem: B): Vector[B]
```

[33][**RANGES**] The Range class represents integer value ranges, with non-zero step value step. It's a special case of an indexed sequence. A range can be declared using `(start) to (end) by (step)`

```
scala> 1 to 3  // [start;end]
// val res5: scala.collection.immutable.Range.Inclusive = Range [start;end) 1 to 3

scala> 1 until 3  //  [start;end)
// val res6: scala.collection.immutable.Range = Range 1 until 3

scala> (res6.start to res6.end by res6.step+1).length
val res9: Int = 2
```

[34][**SOME HOF FOR SEQS**] Here, p represents a predicate, s and t a sequence, f a function [scala.collection.LinearSeqOps]:

s exists p	`def exists(p: (A) => Boolean): Boolean`
s forall p	`def forall(p: (A) => Boolean): Boolean`
s zip t	`def zip[B](that: IterableOnce[B]): CC[(A, B)]`
s.unzip	`def unzip[A1, A2](implicit asPair: (A) => (A1, A2)): (CC[A1], CC[A2])`
s flatMap f	`def flatMap[B](f: (A) => IterableOnce[B]): IterableOnce[B]`
s.sum	`def sum[B >: A](implicit num: math.Numeric[B]): B`
s.product	`def product[B >: A](implicit num: math.Numeric[B]): B`
s.max, s.min	`defmax[B >: A](implicit ord: math.Ordering[B]): A`

[35][**FOR EXPRESSIONS**] A typical pattern used to generate a collection iterating on another collection is
(range) flatMap (collection) filter (f)
For eaxample to generate the set of pairs of integers between 1 and 3 having a sum that is prime:

```
(1 to 3) flatMap (i⇒(1 to i) map (j⇒(i,j))) filter {case (i,j)⇒isPrime(i+j)}
// or
(1 to 3) flatMap (i⇒(1 to i) map (j⇒(i,j))) filter (t ⇒ isPrime(t._1 + t._2))
```

This pattern is easily mapped into a **for comprehension**:

```
for { enumerators } yield expression
// or
for { enumerators } statement
```

Where an enumerator can be a **generator** (v <− collection) or a **filter** (**if** (condition)). In the case with *yield*, a new collection is built using "expression" as element, instantiated according to what is specified by the enumerators. For ex., the previous example becomes:

```
for {i <− 1 to 3; j <−1 to i; if isPrime(i+j)} yield (i,j)
// or
for (i <− 1 to 3; j <−1 to i; if isPrime(i+j)) yield (i,j)
```

In the case without the *yield*, the the for expression is used like a standard for (for the side effects of the statement) and returns ():Unit.
Given a collection of objects, for comprehension allows to express structured queries on the collection. For ex, assume we have a List of obects Book **case class** Book(title:String, authors:List[String]) , than we can write:
for{b<−books; a<−b.authors; **if** a startsWith ''Bird''} **yield** b. title

[36][**SETS**] A collection that is:
1) immutable,
2) heterogeneous (nestable),
3) UNORDERED,
4) WITHOUT DUPLICATES.

```
scala> val s = Set(1,2,"ciao")
// val s: scala.collection.immutable.Set[Any] = Set(1, 2, ciao)

scala> val s = Set(1,2,3)
// val s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s ++ (1::5::Nil)
// val res36: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 5)

scala> s + 4
// val res38: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
```

[37][**REPEATED PARAMETERS**] Using a * postfix operator It is possible to have vararg-like methods. Caveat:
- repeated parameters are always the last method parameter;
- repeated parameters should be of the same type;
- a method can have, at most, a single repeated parameter.
Using the so-called type ascription **_*** it is possible to "decompose" a Seq in order to pass it to a method that requires a list of parameters.

```
scala> def m(l:Int*) = l map (_*10)
// def m(l: Int*): Seq[Int]

scala> m(1,2,3)
// val res49: Seq[Int] = ArraySeq(10, 20, 30)

scala> val v = List(1,2,3,4)
// val v: List[Int] = List(1, 2, 3, 4)

scala> m(v:_*)
// val res50: Seq[Int] = List(10, 20, 30, 40)

// The argument is treated like a Seq inside the function:
scala> def m(l:Int*) = l.getClass
// def m(l: Int*): Class[_ <: Seq[Int]]
```

[38][**MAPS**] Maps are (iterable) collections that associate keys to values; the type is Map[Key,Value]
Maps values can be accessed using **get** operator, that returns a value (of type **Option[value_type]**) that can be either Some(x) or None. This mechanism simplify the handling of undefined values:

```
scala> val cap = Map("it"->"roma", "fr"->"paris")
// val cap: scala.collection.immutable.Map[String,String] = Map(it -> roma, fr -> paris)

scala> cap get "it" match {
| case Some(x) => println(x)
| case None => println("sorry")
| }
// roma
```

It is possible to iterate over them as It would iterate over a collection of pairs:

```
scala> cap map {case (s,c) => s}
// val res39: scala.collection.immutable.Iterable[String] = List(it, fr)
```

Maps also extend the function type, so that they can be used as partial functions. It's also possible to extend them using the operator **withDefaultValue** to transform them into total function:

```
scala> cap("it")
// val res40: String = roma

scala> (cap withDefaultValue "unk")("unk")
// val res42: String = unk
```

Maps can be concatenated (as all Seq) using ++; the RH operand has the priority in case of duplicate keys:

```
scala> (Map(1->100, 2->"pippo") ++ Map(2->200, 3->300)).values
// val res59: Iterable[Any] = Iterable(100, 200, 300)
```

[39][**GROUP_BY**] groupBy partitions a collection depending on the value returned by a function f applied to each element; It returns a Map where:
- the keys are all the possible codomain values of f (associated to at least one element of the domain);
- the values are, for each key k, all the elements of the collection that evaluate to k.

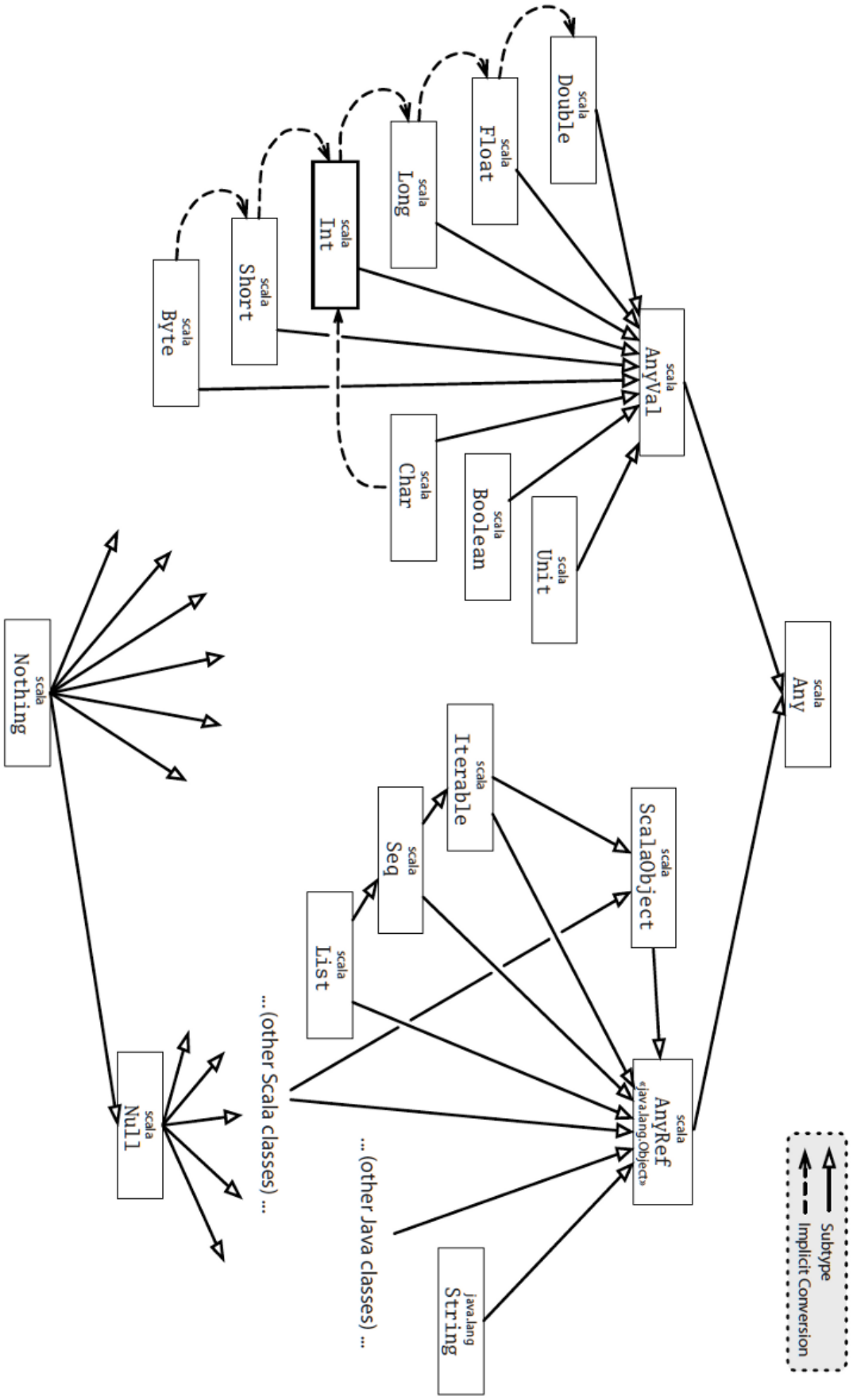
```
scala> (1 to 10) groupBy (v => if (v%2==0) "even" else "odd")
// [...] HashMap(odd -> Vector(1, 3, 5, 7, 9), even -> Vector(2, 4, 6, 8, 10))
```

[40][**STREAMS AND LAZY LISTS**] Streams are like lists in which the tail is only evaluated when needed. This allow to avoid inefficient computations, as we could do with a traditional *for*, while keeping the functional approach.
Stream is deprecated in favor of LazyList.
A **LazyList** does the same of a stream but with "full lazyness": not only the tail, but even the head is only computed when needed. This implies that the head must be computed to discover if the list is empty. "Elements are **memoized**; that is, the value of each element is computed at most once. Elements are computed in-order and are never skipped. In other words, accessing the tail causes the head to be computed first. A LazyList may be **infinite**. For example, LazyList.from(0) contains all of the natural numbers 0, 1, 2, and so on. For infinite sequences, some methods (such as count, sum, max or min) will not terminate."
LLs can be constructed with the **#::** operator, alias of LazyList.cons, or with **.to(LazyList)**, and concatenated with **#:::**.
LLs can be implemented using the **lazy** keyword (§4) or leveraging the CBN approach (§3).

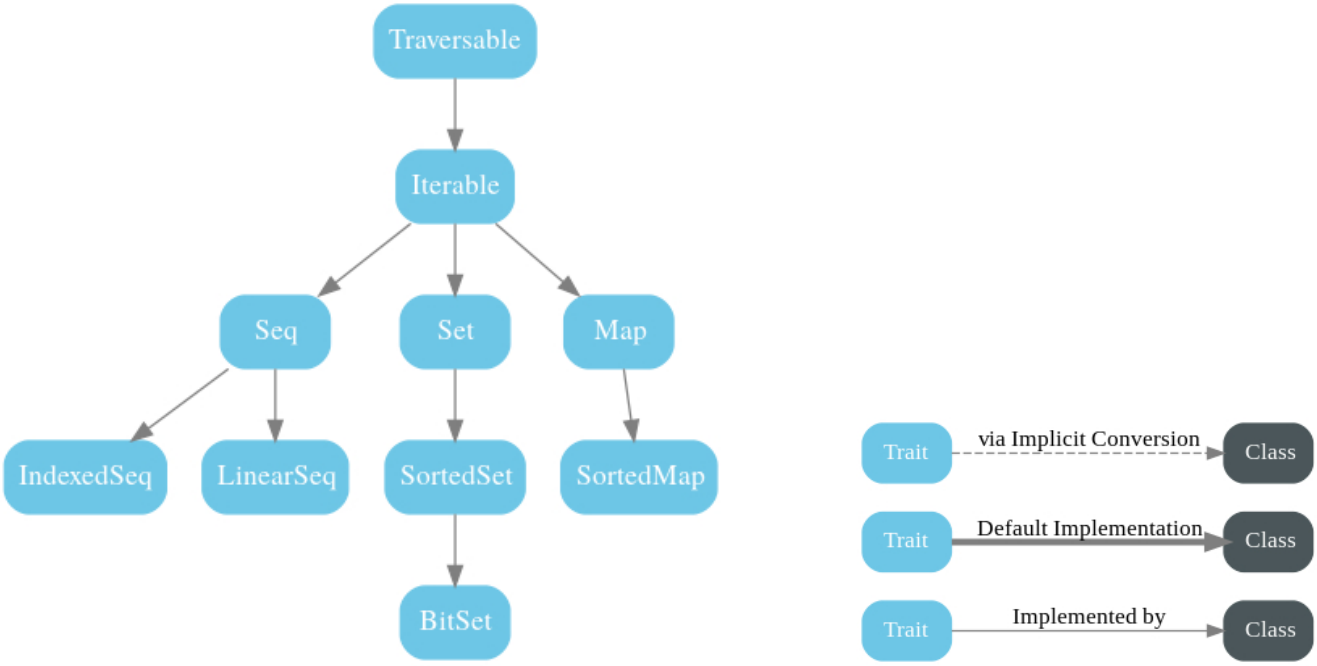
```
scala> val ll = LazyList(1,2,3) #::: LazyList.empty
val ll: scala.collection.immutable.LazyList[Int] = LazyList(<not computed>)

scala> (0 #:: ll).toList
val res109: List[Int] = List(0, 1, 2, 3)
```

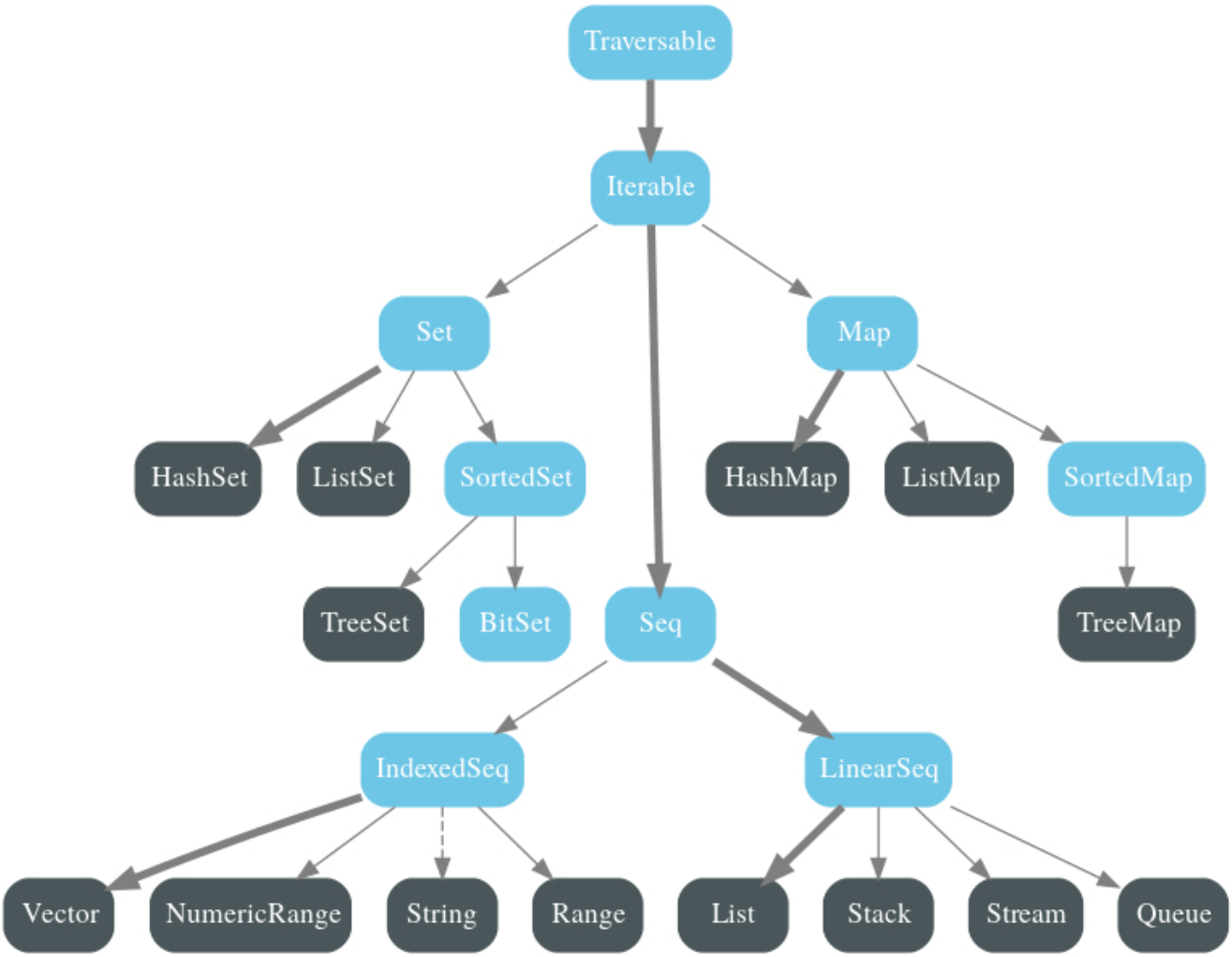
A BUILT-IN HIERARCHY



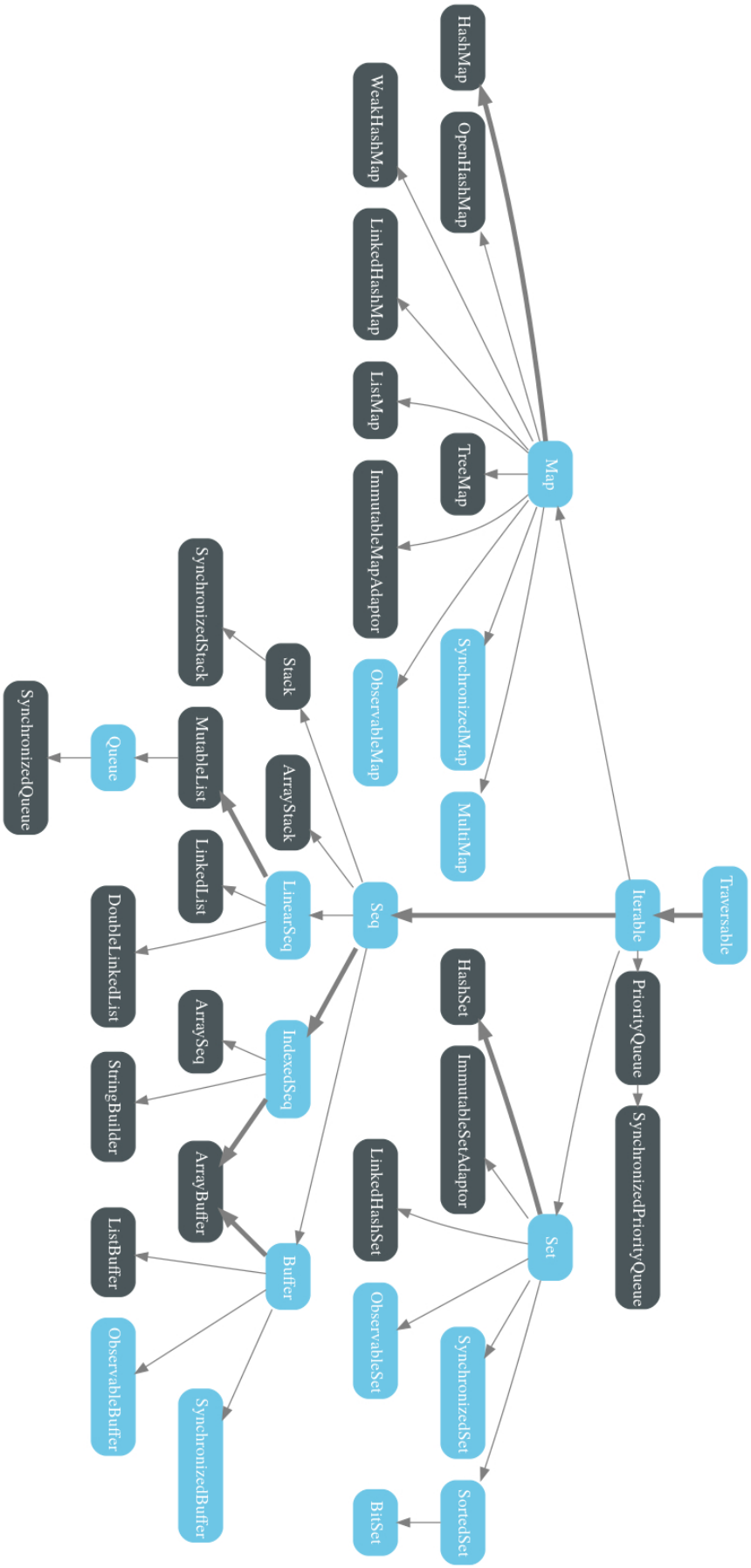
B SCALA.COLLECTION HIERARCHY



C SCALA.COLLECTION.IMMUTABLE HIERARCHY



D SCALA.COLLECTION.MUTABLE HIERARCHY



E OPERATORS PRECEDENCE

Category	Operator	Associativity
Postfix	() []	Left to right
Unary	! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Figure 3: Operators with the highest precedence appear at the top.

F EXTERNAL LINKS

References

[1] https://scala-lang.org/files/archive/spec/2.13/08-pattern-matching.html#extractor-patterns

[2] https://scala-lang.org/files/archive/spec/2.13/10-xml-expressions-and-patterns.html#xml-patterns

[3] https://docs.scala-lang.org/overviews/reflection/typetags-manifests.html

[4] https://docs.scala-lang.org/overviews/scala-book/case-classes.html