Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Tools and Advanced Scala Features for Library Designers

Attività Propedeutica alla Prova Finale

Supervisor:
Prof. MIRKO VIROLI

Presented by:
ROBERTO CASADEI

Examiner:
Prof. ANTONIO NATALI

Winter, 2015

Contents

1 A Quick Tour of Scala **9**
1.1 Introduction to Scala 10
1.2 A Quick Tour of the Basics 11
1.2.1 Scala Class Hierarchy 11
1.2.2 Key data structures 12
1.3 Basic Functional Programming in Scala 15
1.3.1 Expressions 15
1.3.2 Immutability 15
1.3.3 On Functions 16
1.3.4 Laziness 19
1.3.5 Non-strict functions and call-by-name parameters 20
1.3.6 For comprehension 20
1.3.7 Pattern matching 21
1.3.8 Case classes and Algebraic Data Types (ADTs) 22
1.3.9 Functional Error Handling: Option and Either 23
1.4 Basic Object-Oriented Programming in Scala 24
1.4.1 Packages, package objects, imports 25
1.4.2 Classes 26
1.4.3 Fields and Visibility 28
1.4.4 Objects 29
1.4.5 Object equality 30
1.5 Traits 30
1.5.1 Trait as interfaces 30

1.5.2 Trait as mixins . . . 31

2 Advanced Features in Scala **33**

2.1 More on OOP in Scala . . . 33

2.1.1 Class construction and linearization . . . 34

2.1.2 Traits: member overriding and `super` resolution . . . 35

2.1.3 Trait instantiation, refinement, early definitions . . . 36

2.2 Advanced types . . . 38

2.2.1 Some preparatory definitions . . . 38

2.2.2 Parameterized types . . . 39

2.2.3 Infix types . . . 39

2.2.4 Structural types . . . 39

2.2.5 Compound types . . . 39

2.2.6 Existential types . . . 40

2.2.7 Self-types . . . 41

2.2.8 Abstract type members . . . 41

2.3 Generic programming in Scala . . . 42

2.3.1 Type parameters . . . 44

2.3.2 Type bounds (bounded quantification) . . . 44

2.3.3 Type variance . . . 46

2.3.4 Abstract types vs. type parameters . . . 47

2.4 Implicits . . . 50

2.4.1 Implicit scope . . . 51

2.4.2 Implicit classes . . . 54

2.4.3 More on implicits . . . 54

3 Advanced Programming Techniques in Scala **59**

3.1 ”Pimp my library” pattern . . . 59

3.2 Type classes . . . 61

3.3 Component modelling and implementation . . . 63

3.4 Cake Pattern . . . 67

3.5 Family Polymorphism . . . 72

3.5.1 Case study: an interpreter and simulator for aggregate programming 76
3.6 Internal DSL Development 78
3.6.1 On syntactic sugar 78
3.6.2 On associativity and precedence 80
3.6.3 Examples 81

4 The Scala Toolchain 85
4.1 Software project automation with sbt 85
4.2 Testing in Scala with ScalaTest and ScalaCheck 88
4.2.1 Testing Styles 90
4.2.2 Property-based Testing 94
4.3 Performance Testing with ScalaMeter 98
4.3.1 Microbenchmarking 100
4.3.2 Performance regression testing 101

5 Summary and further directions 103

References 104

Introduction

The design of a software library is not substantially different from the design of an application. In both cases there are functional and non-functional requirements – explicitly or implicitly stated – that should be satisfied, and in both cases there is a resulting software product that is released to its intended end-users.

However, library designers usually require knowledge and skills that are often unnecessary to application designers. Intuitively, this is essentially because, while for application designers is sufficient to *understand how tools can be used* (i.e., without necessarily understand the general implications or the underlying mechanisms), library designers have to enter into the merits of *how things actually work*.

A strong link exists between these two roles in software development. Application designers are in direct contact with the forces originating from the problem domain. These forces may reveal inadequacies inherent in the conceptual or practical tools adopted – in other words, an **abstraction gap**. This means that there is a gap between the problem and the platform on which our solutions are implemented; this means it is time to fix, extend, improve, or rethink the tools that are used to tackle problems – it is where library designers come in.

An abstraction gap may manifest itself through recurring problems or issues. Once a solution is found for a problem, it would be desirable to **reuse** it across all instances of the problem. This may require *abstracting* from the peculiarities of the contexts in which the problem shows itself. Thus, one key activity of library designers is one of **generalization**.

Another symptom of abstraction gap is complication or difficulty in use. An important property of software libraries is *usability*. Application designers cannot effectively solve application-specific problems if their tools are little usable. As a consequence, we may say that **simplification** is another meta-goal for library designers.

In summary, the general goal of library designers is to develop libraries of components that are, in addition of being functionally correct, usable and reusable. This requires an in-depth knowledge of the problem domain and a significant expertise in the reference platform.

This very work presents a set of features in the Scala programming language that collectively represent a impressive toolbox for library designers. Scala comes with a powerful static type system, a good integration of object-oriented and functional programming abstractions, and sophisticated features that make it an interesting source for exploration. First, we will introduce these elements independently. Then, we will see how these advanced features of Scala can be combined to support the development of high-quality software libraries. Finally, we will also have a quick look at some concrete tools that may be important during the development of Scala-based libraries or projects.

In the end, this preparatory activity has consisted in the study and application of many theorical and practical notions in the Scala universe. It has more emphasis on breadth than depth, even though many presented features are quite sophisticated. Moreover, this is essentially a work of synthesis; many of the insights here expressed derive from the reading of multiple books and articles – refer to the bibliography at the end of the document for a list of the key contributing sources.

Chapter 1

A Quick Tour of Scala

This chapter serves as a warm-up. It will introduce some key basic features of Scala whose understanding is essential to understand the following chapters. We assume some working knowledge of OOP and FP, as well as familiarity with a Java-like programming language.

Outline

1. Key characteristics of Scala
2. Scala type hierarchy
3. Key data structures: tuples, sequences, maps, sets
4. Basics of Functional Programming in Scala: expressions, immutability, first-class functions, lambdas, partial functions, closures, laziness, pattern matching, extractors, case classes, Algebraic Data Types, for comprehension, functional error handling
5. Basics of Object-Oriented Programming in Scala: packages, classes, constructors, fields, objects, object equality
6. Traits: traits as interfaces, traits as mixins

1.1 Introduction to Scala

Scala is a general-purpose programming language with the following main characteristics [1]

- runs on the JVM and **integrates with Java** and the Java ecosystem in a seamless way
- provides a **smooth integration of the object-oriented and functional paradigms**
- is a **pure OOPL**, i.e., every value is an object and every operation is a method call
- is designed to be a **"scalable" language**, i.e., it keeps things simple while accomodating growing complexity
- has a powerful, expressive static **type system** with type inference

Even though Scala compiles to bytecode, it also comes with a *Read-Print-Eval Loop (REPL)* that supports interpreter-style experiments.

```
$ scala

scala> 1 + 1
res0: Int = 2

scala> val s = "Hello, Scala"
s: String = Hello, Scala

scala> s = s + "!!!"
<console>:11: error: reassignment to val

scala> println(s)
Hello, Scala
```

[1]For more: What is Scala? `http://www.scala-lang.org/what-is-scala.html`

1.2 A Quick Tour of the Basics

1.2.1 Scala Class Hierarchy

All values are objects, i.e., instances of a class. Figure 1.1 shows the Scala class hierarchy.

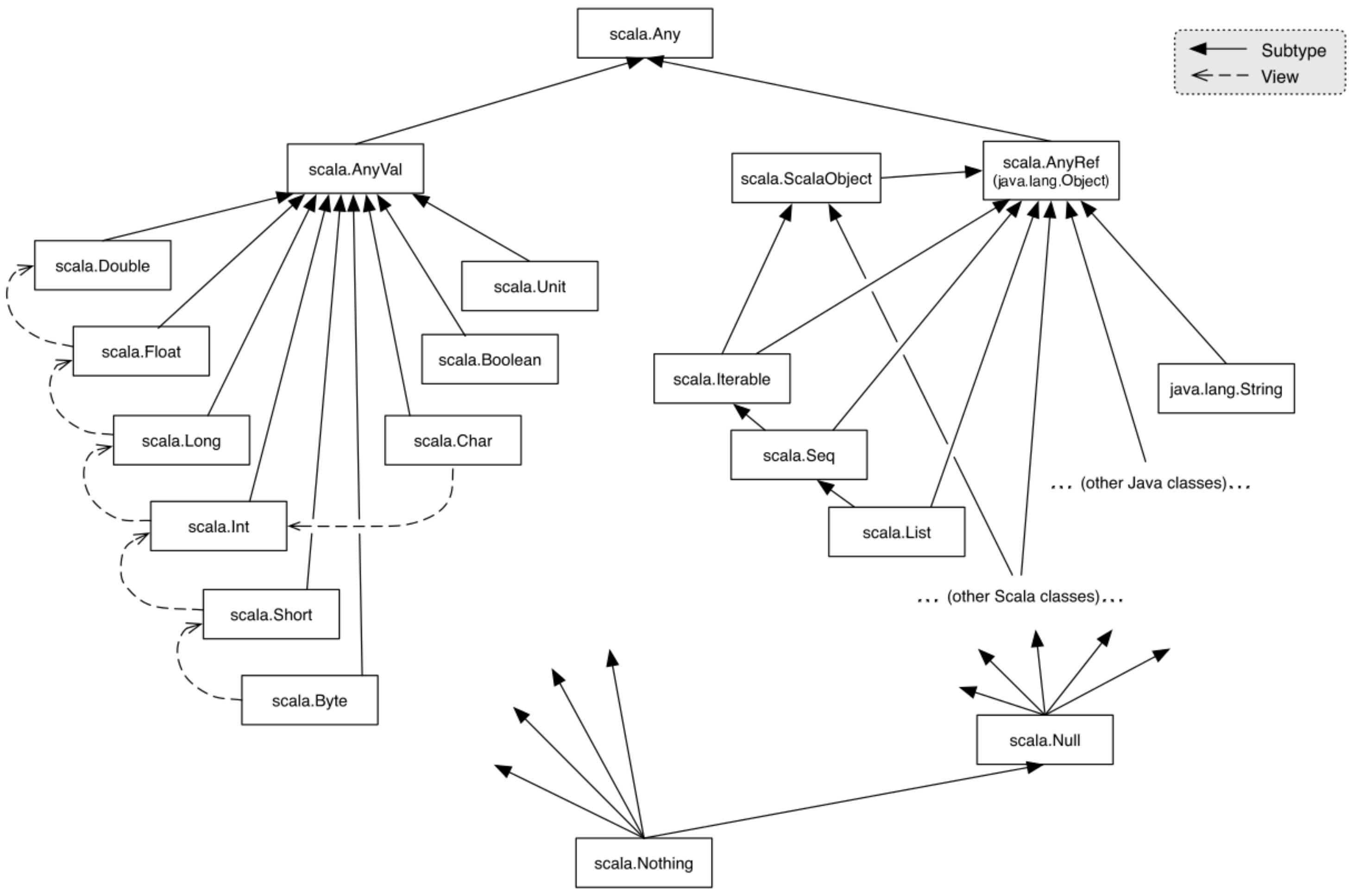

Figure 1.1: Scala Class Hierarchy

We see that a first difference is between *value classes* (which are predefined and correspond to primitive data types in Java) and *reference classes*.

Note that, in addition to the top types (the root `Any` plus `AnyVal` for value classes and `AnyRef` for reference classes), we also have *bottom types*:

`Null` is the bottom type of all reference types, and `Nothing` is the bottom type of all types. There are no values of type `Nothing`, however this type is useful for example when it comes to generic type variance: `List[+T]` is the list type, it is covariant in `T` (see Section 2.3.3), and `Nil` (the empty list) is defined as being of type `List[Nothing]`; as a consequence, object `Nil` is the empty list for any list type.

1.2.2 Key data structures

The following taxonomy provides an overview of the key data structures in Scala

- **Tuples**: immutable, fixed-length, ordered collection of etherogeneous values
- Collections
  - **Sequences**: ordered collections
  - **Maps**: collections of key-value pairs
  - **Sets**: collections with no duplicates elements

It is worth noting that Scala's collections split into three dichotomies:

- Immutable and mutable collections
- Eager and lazy collections
- Sequential and parallel collections

This means that, for example, maps have immutable and mutable implementations, and so on.

We can zoom the `scala.collection` package (see Figure 1.2) to have an overview of the main abstract classes (traits).

Here are some examples of use

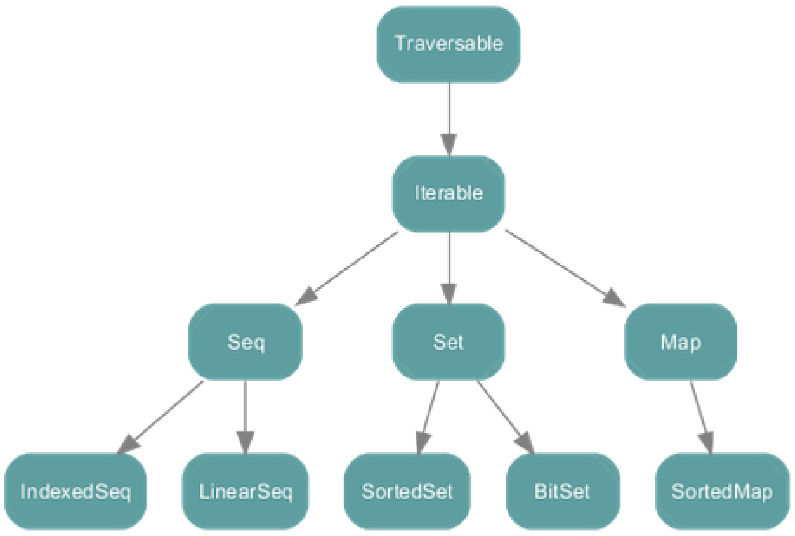

Figure 1.2: Scala collections

```
/////////////// TUPLES

val t1 = ("Scala", 2.10, false)
// t1: (String, Double, Boolean)
t1._1 + t1._2
// String = Scala2.0
val t2 = Tuple2[String, Int]("xxx", 0)
// t2: (String, Int) = (xxx,0)

/////////////// SETS

val s1 = Set(1,1,1,2,3)
// s1: Set[Int] = Set(1, 2, 3)
s1(3)
// Boolean = true (i.e., 3 is in the set)
s1 & Set(2,7,5)
// Set[Int] = Set(2)

/////////////// MAPS

val m1 = scala.collection.mutable.Map[String, Int]()
// m1: scala.collection.mutable.Map[String,Int]
m1 += "a" -> 88 += "b" -> 99
// m1.type = Map(b -> 99, a -> 88)
m1("b")
```

```scala
// Int = 99 (map access by key)
m1.keys
// Iterable[String] = Set(b, a)
m1.toSeq
// Seq[(String, Int)] = ArrayBuffer((b, 99), (a, 88))

////////////// SEQUENCES (LISTS)

val l1 = (1 to 5).toList
// l1: List[Int] = List(1, 2, 3, 4, 5)
l1.length
// Int = 5
l1(2)
// Int = 3 (i.e., 0-indexed access)
```

Notes:

- `List`, `Set`, `Map` are generic classes and accept type parameters
- The type inference allows you to not specify the type in many circumstances
- Tuples have a convenient syntactic sugar
- 2-tuple elements have a syntactic sugar `a -> b` that comes useful with maps, as key-value pairs can be simply represented as `Tuple2[K,V]` objects
- It seems that is possibly to use objects to functions. Actually, under the hood, `o(...)` is translated to the method call `o.apply(...)`
- There are no operators and `m1 +=...` is actually a method call `m1.+=(...)`. A similar thing happens in the case of `1 to 5` which is translated to `1.to(5)`, producing a `Range` object
- In Scala, parameterless methods can be called as $o.m$ instead of $o.m()$. For example, `m1.keys` is a method (and not an access to a field). Moreover, it is possible to enforce this style of invokation; for example, the expression `m1.keys()` is in fact an error.

1.3 Basic Functional Programming in Scala

In the Functional Programming paradigm, programs consist of pure functions (i.e., without side effects) that operate on values and purely functional, immutable data structures. Computation is not expressed through statements (as in the imperative paradigm), but as an evaluation of expressions.

1.3.1 Expressions

Many constructs that in Java are statements, in Scala are expressions.

```scala
val x = if(true) "then" else 99.9
// x: Any = then
val y = try { 1 } catch { case _ => 88.8 } finally { "xxx" }
// y: Double = 1.0
val z = { 1; true; "foo" }
```

Notes:

- Multiple expressions on the same line must be separated by semicolon ;
- For the `if-else` construct, the return type is the lowest common supertype of the types of the last expressions in the `if` and `else` branches
- For the `try-catch-finally` construct, the return type is the lowest common supertype of the types of the last expressions in the `try` and `catch` blocks (`finally` is used only for side effects)
- The return value for a block is the result of the evaluation of its last expression

1.3.2 Immutability

A common advice in Scala is: *prefer immutability.*

In Scala, you can have immutable or mutable *references* through `val` or `var`, respectively. Then, a reference may point to an object which may be immutable or mutable. That is, there are four combinations:

1. Immutable reference to immutable object
2. Mutable reference to immutable object
3. Immutable reference to mutable object
4. Mutable reference to mutable object

```
import scala.collection._

val a = Set(1,2,3)
var b = Set(1,2,3)
val c = mutable.Set(1,2,3)
var d = mutable.Set(1,2,3)
```

It should be noted that Scala immutable collections are persistent (i.e., when changed, their previous version is preserved). One may think that creating a new version of a collection for every "update" operation is inefficient. Actually, this is usually not the case thanks to structural sharing.

1.3.3 On Functions

First-Class Functions

In FP, functions are first-class entities that can be stored, passed to, and returned from other functions.

In Scala, a *function value* is an object of type `Function<N>`. In the Scala standard library, there are predefined function traits for functions of arity from 0 to 22: `Function0[+R]`, `Function1[-T1,+R]`, `Function2[-T1,-T2,+R]` and so on (`-T` indicates that type parameter `T` is contravariant, while `+R` indicates that type parameter `R` is covariant). These traits define an abstract method `apply(v1: T1, v2: T2, ...): R` that is invoked on the function call. In fact, the function call is a syntactic sugar for the method call on the function object.

Function types

Scala provides some syntactic sugar for *function types*. That is, a function type can be expressed as `(T1,..,Tn)=>R`, which is a shorthand for the function trait `FunctionN[T1,...,TN,R]`

Anonymous functions (lambdas)

Functions can be created on-the-fly, and Scala provides concise shortcuts for that:

```
val add1 = (x: Int, y: Int) => x + y
val add2 = add1(_, _)
val add3: (Int, Int) => Int = (x, y) => x + y
val add4: (Int, Int) => Int = _ + _
val add5: Int => Int => Int = x => y => x + y // Curried
```

It is also possible to turn a method to a function:

```
object x { // Singleton object
  def m(s: String, n: Int) = (1 to n).map(_ => s).foldLeft("")(_+_)
} // Note that return type doesn't need to be specified thanks to
   type inference

val f = x.m _    // f: (String, Int) => String
f("a", 10)       // String = aaaaaaaaaa
```

Higher-Order Functions (HOFs)

Higher-order functions are functions that accept functions as arguments or return functions as result.

A typical HOF is `filter`, which filters a list based on some predicate on its elements.

```
def filter[T](lst: List[T])(p: T => Boolean): List[T] = lst match {
    case h :: t if p(h) => h :: filter(t)(p)
    case h :: t => filter(t)(p)
    case l => l
}

filter((1 to 10).toList)(_%2==0) // List[Int] = List(2,4,6,8,10)
```

Notes:

- `filter` is a *generic* function
- The function is defined with two parameter groups to help the type inferencer (otherwise the lambda would need to specify the type of its argument)
- The implementation is not ideal because it will overflow the stack for big lists
- The implementation makes use of pattern matching with guards

Closures

Scala supports functions as closures.

```
def getClosure: () => Int = {
  var i = 77
  () => { i+=1; i }
}
val c = getClosure
c() // => 78
c() // => 79
```

Currying

Currying is the process of transforming a n-ary function into a chain of 1-ary functions.

```
val f = (x:Int, y:Int, z:Int) => x + y + z
val g = f.curried  // g: Int => (Int => (Int => Int)) = <function1>
g(3)(2)(1)         // => 6

def m(x: Int)(y: Int)(z: Int) = x + y + z
val h = m _        // h: Int => (Int => (Int => Int)) = <function1>
```

Partial function application

A partially applied function is a function for which some parameters have already been bound to their arguments.

```
def repeat(s: String, n: Int, reverse: Boolean) = {
   val res = (1 to n).map(_ => s).foldLeft("")(_+_)
   if(reverse) res.reverse else res
}

val repeatAndReverse = repeat(_:String, _:Int, true)
val twiceAndReverse = repeatAndReverse(_:String, 2)
twiceAndReverse("ab") // => baba
```

Partial functions

Partially applied functions should not be confused with *partial functions*, i.e., functions that are defined only for a subset of their (natural) domain. Partial functions have type `PartialFunction[-A,+B]`.

```
val buildRange: PartialFunction[Int, Range] = {
    case n if n>0 => 1 to n
}
buildRange.isDefinedAt(-1)  // => false
buildRange(4)               // => Range(1,2,3,4)
buildRange(-2)              // scala.MatchError
```

1.3.4 Laziness

Lazy values/variables are initialized at their first use:

```
lazy val x = println("initialized!")   // x: Unit = <lazy>
x           // initialized!
x
x
```

1.3.5 Non-strict functions and call-by-name parameters

Strict functions always evaluate their arguments. On the contrary, non-strict functions may choose not to evaluate one or more arguments. One can define non-strict functions by using functions as arguments. Scala provides a nicer syntax through emphcall-by-name parameters.

```
def myIf[A](c: Boolean)(th: => A)(el: => A) = if(c) th else el

myIf(true){ println("a") } { println("b") }  // a
```

1.3.6 For comprehension

For-comprehension is a convenient syntax for writing a sequence of nested calls to `map`, `flatMap`, and `filter`.

The `for` consists of a head with multiple generators (possibly guarded) and variable assignments, and a body. The body may start with `yield`, in which case a value is produced each iteration, resulting in a collection.

```
for(x <- 1 to 5) print(x)     // 12345

for(
  x <- 1 to 5;
  y <- 1 to 2 if x%2==0;
  a = "a"
) yield (x,y)
// IndexedSeq[(Int,Int,String)] = Vector((2,1,a),(2,2,a),(4,1,a)
    ,(4,2,a))
```

1.3.7 Pattern matching

Pattern matching is a feature that allows to match objects against patterns, bind variables, and select appropriate actions depending from the matching cases.

```
obj match {
    case x: Int if x<=0   => 0
    case x: Int if x>0    => x
    case s: String        => Integer.parseInt(s)
    case _                => 0
}

lst match {
    case x :: y :: Nil  => x + " " + y
    case 0 :: tail      => "0 ..."
}

arr match {
    case Array(x, y)  => x + " " + y
    case whole @ Array(0, rest @ _*) => "0 " + rest
}
// The Array companion object is an extractor => Array.unapplySeq(
   arr)

tpl match {
   case (x, y) => x + " " + y
}
```

It's important to note that the patterns should exhaustively cover all inputs. In the event that no case matches, a `MatchError` exception is raised at runtime.

Extractors

Scala provides a mechanism for defining custom extractors for pattern matching.

For example, let's define an extractor to extract the starting uppercase character in a string, if any:

```
object StartsByUppercase {
  def unapply(s: String): Option[Char] =
    if(s.isEmpty || !s.head.isUpper) None else Some(s.head)
}

"Hello" match { case StartsByUppercase(c) => c } // => H
"hello" match {
  case StartsByUppercase(c) => c;
  case _ => "default"
} // => default
```

1.3.8 Case classes and Algebraic Data Types (ADTs)

Case classes are regular classes which publicly expose their constructor parameters. In addition, they come with a few facilitations:

- A companion object is automatically defined
- `apply` is automatically provided in the companion object (so that you don't need to use `new` to create instances)
- `unapply` is automatically provided in the companion object (so that you can use pattern matching decomposition)
- `equals`, `toString`, `hashCode`, `copy` are automatically provided (the default implementation of `equal` implements structural equality)

Case classes can be used to implement *algebraic data types*:

```
abstract class Tree[T]
case class Leaf[T](value: T) extends Tree[T]
case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]

val tree = Node(Node(Leaf(1), Leaf(2)), Leaf(3))
// tree: Node[Int] = Node(Node(Leaf(1),Leaf(2)),Leaf(3))

def depth[T](t: Tree[T]): Int = t match {
  case Node(l,r) => 1 + math.max(maxDepth(l), maxDepth(r))
  case Leaf(_) => 0
}
```

```scala
depth(tree) // => 2
```

1.3.9 Functional Error Handling: Option and Either

It can be argued that raising an exception is a side-effect. An alternate approach, purely functional, is to represent failures as ordinary values and then defining functions to encode error handling patterns on these values.

In scala, for the purpose you are given two types:

1. `Option[T]`: indicating the presence of a value (via `Some(v)`) or its absence
2. `Either[A,B]`: representing two alternate values, `Left(l)` or `Right(r)` (by convention, the former is used for failure, and the latter for success)

Now, we concentrate on `Option`. The idiomatic way of using options is to call on them functions such as `map, flatMap, filter, foreach`. This allows, for example, to split a computation into multiple stages, and abort the overall computation as soon as a failure is encountered; moreover, you can specify defaults to apply in case of failure.

Here is an example:

```scala
case class User(name: String, age: Int)

def validate[T](pred: T => Boolean): T => Option[T] =
  v => if(pred(v)) Some(v) else None

val validateName = validate[String](name => {
  val l = name.length; l>6 && l<40
})
val validateAge = validate[Int](age => age>=0 && age <=120)

val name = getName()
val age = getAge()

val user = for(_name <- validateName(name);
    _age <- validateAge(age)
) yield User(_name, _age)

val acct = user.flatMap(u => createAccountForUser(u)).
    getOrElse(throw new Exception("Cannot create the account"))
```

1.4 Basic Object-Oriented Programming in Scala

Scala is a class-based OOPL where any value is an object.

Let's see an abstract example that summarises many features in one place:

```
package __root.__org
package parentpackage // org.parentpackage

package otherpkg1 { class SomeBaseClass }
package otherpkg2 { trait SomeTrait }

package mypackage { // org.parentpackage.mypackage

import otherpkg1.{SomeBaseClass => MyBase} // Selective + rename
import otherpkg2._  // Import all

class A(val x: Int, private var y: Char, arg: Double)
  extends MyBase with Serializable with SomeTrait {
  private[mypackage] var _z = arg
  def z = _z
  def z_=(newZ: Double) { _z = newZ }

  private[this] var foo: Boolean = _

  @scala.beans.BeanProperty var bar: Int = _

  def this() {
    this(0, 'a', 0.0)
    this.bar = 2
    this.setBar(1)
  }

  def update(flip: Boolean, foo: Boolean) = this.foo = foo ^ flip
  override val toString = s"A($x,$y,$z,$foo)"

  class Inner { val myFoo = A.this.foo }
  var inner: A#Inner = new Inner
}

object A {
  def apply(c: Char) = new A(0, c, 0)

  def sortOfStaticMethod() = { true }
}

package object mypackage {
  val obj = new A()          // Use auxiliary constructor
  obj._z                     // Field '_z' is package private
  obj.z = 7.7                // Actually a method call: obj.z=(7.7)
  print(obj.toString)        // toString was overridden as a val!
  obj(true) = false          // Rewritten as: obj.update(true,false)
  val inner = new obj.Inner // Nested classes are object-specific
  val obj2 = A('z')          // Rewritten as: A.apply('z')
  A.sortOfStaticMethod       // Calls method on companion object
  obj2.inner = obj.inner     // Ok thanks to type projection
}

}
```

1.4.1 Packages, package objects, imports

Packages can contain definitions for classes, objects, and traits. Moreover, a package can have an associated **package object** with the same name where you can put function, value, and variable definitions.

Here are some practical details on packages:

- In Scala, packages can be used in a more flexible way than in Java. For example, in Scala you don't need to specify the package at the top of the file, and you could contribute to more than one packages in the same file.

- Scala supports flexible forms of import (all, selective, aliased). Moreover, you can use `import` anywhere, even inside functions.

- Package paths are relative, not absolute. That is, `package a; package b` is equivalent to `package a.b`. However, you can define absolute package paths by explicitly referring to the special, top-level package `_root_`
- On the REPL, you can play with packages using `:paste -raw`

1.4.2 Classes

Here are some facts about classes:

- A class can inherit by one class at most (**single-class inheritance** scheme), but can implement multiple traits (interfaces)
- Each class has a **primary constructor** which consists of the class body itself; this means that the class body can contain also statements, not only definitions
- The primary constructor can specify parameters. These parameters can be normal parameters as in function calls, or they can be marked by `val` or `var` (possibly with visibility modifiers) to make them become fields
- A class can have **auxiliary constructors** which are named `this` and must start with a call to a previously defined auxiliary constructor or the primary constructor
- Scala supports **fields** with automatically generated getters and setters
- Each class can have a **companion object** with the same name as the class
- There are **no static methods** in Scala, but they can be implemented as methods on the class' companion object.

On inheritance and overriding

The `extend` and `final` keywords work like in Java. Moreover, classes can be declared as `sealed`, meaning that they can only be inherited within the same source file in which they are declared.

It is worth mentioning some details:

- In Scala, you have the ability to override fields (not only methods)
- It is also possible to ”downgrade” a method to a field. The following rules apply:
  - A `def` can only override another `def`
  - A `val` can only override another `val` *or a parameterless* ***`def`***
  - A `var` can only override an abstract `var` or a pair of getter/setter `def`s; but note:

```
trait A { def x: Int }
class B extends A { var x = 8 }
// In this case, B overrides "def x:Int" and **adds** "def x_=
    (v:Int):Unit"

trait A { def x: Int }
class B extends A { override var x = 8 } // ERROR
// Error: method x_= overrides nothing
//       class B extends A { override var x = 8 }
```

- When you override a non-abstract method/field, you must use the `override` keyword

Nested classes

Classes can have *nested classes*, but it is essential to note that a inner class is tied to the object, not to the class. However, you can express a type such as ”a `Inner` of any `A`” via *type projection* `A#Inner` (see Section 2.2.1).

1.4.3 Fields and Visibility

Fields are introduced with `val` and `var` declarations within the class body or as constructor parameters. The logic of method generation for fields is coherent with the concept of im/mutability; that is, a `val` field is only given a getter, while a `var` field is given a getter and a setter.

Access modifiers are more sophisticated as in Java. In particular, you can restrict visibility to a package, class, or object using the syntax `private[X]` or `protected[X]`. For example, you have the expressibility to state:

- `public`: public access (it is the default – note it is different from Java's package-private default)
- `protected`: inheritance access – means that any subclass can access the member (also from other objects of any subclass)
- `private`: class-private access – means that the member can be accessed only from the same class (also from other objects of the same class, no subclass)
- `protected[package]`: package-private and inheritance access – means the member is accessible everywhere in the package and from any subclass (possibly in another packages)
- `private[package]`: package-private (without inheritance access)
- `protected[this]`, `private[this]`: object-protected/private access

```
// private (must fail when accessed in subclass)
class X(private val x: Int)
class Y extends X(0) { this.x } // ERROR
class Z extends X(0) { def otherx(other: X) = other.x } // ERROR

// private vs. private[this]
class X(private val x: Int) { def otherx(other: X) = other.x } // OK
class X(private[this] val x: Int) { def otherx(other: X) = other.x } //
    ERROR

// protected (can access from subclass)
```

```
class X(protected val x: Int);
class Y extends X(0) { this.x } // OK

// protected (on another object)
class X(protected val x: Int)
class Y extends X(0) { def otherx(other: X) = other.x } // ERROR (subtle)
class Z extends X(0) { def otherx(other: Z) = other.x } // OK

// protected[this] (must fail when accessed on another object)
class X(protected[this] val x: Int)
class Y extends X(0) { def otherx(other: Y) = other.x } // ERROR

package a {
  class X(private[a] val x: Int)
  class Y(protected[a] val y: Int)

  package a.b { }; package object b {
    def f = new X(7).x // OK (private[package] includes subpackages)
  }
}
package object a {
  def f = new X(7).x // OK
}

package c {
  class Z extends a.Y(0) { this.y } // OK
}
package object c {
  //def f = new a.X(7).x // ERROR
  //def g = new a.Y(7).x // ERROR
}
```

1.4.4 Objects

We have already seen that a class can have, in the same source file, a *companion object* with the same name. For what concerns visibility, a class and its companion object can access each other's private entities.

More generally, Scala allows to define **singleton objects** with the `object` keyword. An object declaration is similar to a class declaration, with some differences. An object can extend a class and traits, can have fields, methods, inner classes. However, objects cannot have type parameters, nor constructors.

```
object x { }
```

```
val x2: x.type = x   // x.type is a 'singleton type'
x eq x2              // => true (same instance)
```

Note how this feature makes the implementation of the Singleton pattern absolutely trivial.

1.4.5 Object equality

The top-level class `Any` defines two methods `equals` and `hashCode` that can be overridden to implement custom equality. It also introduces two final methods, `==` and `##`, which build on `equals` and `hashCode`, respectively, and also check for null. Then, `AnyRef`, the top-level class for reference classes, defines an `eq` method for checking if two references point to the same object instance.

As a rule of thumb, any implementation of `equals` should be an equivalence relation (i.e., reflexive, symmetric, transitive), and `equals` and `hashCode` should be implemented in a consistent way. Moreover, a situation where you should pay attention is when you mix custom equality and inheritance.

1.5 Traits

Traits are similar to abstract classes, in that they can include both abstract and concrete methods/fields. As a difference, traits can only have a parameterless primary constructor. Another difference is that, while traits can be used everywhere abstract classes can be used, only traits can be used as mixins.

1.5.1 Trait as interfaces

By defining abstract fields and methods, traits work as **interfaces**. That is, all concrete classes (or objects) implementing the trait are required implement the trait's abstract entities.

```
trait Logger {
  def log(msg: String): Unit
}

class ConsoleLogger extends Logger {
  def log(msg: String) = println(msg)
}
```

1.5.2 Trait as mixins

When a trait defines concrete fields and methods, it works as a **mixin**, meaning that its functionality can be mixed into other classes, object, or traits

```
trait Comparable[T] {
  def compareTo(other: T): Int

  def >(other: T) = compareTo(other) > 0
  def <(other: T) = compareTo(other) < 0
  def ===(other: T) = compareTo(other) == 0
}

class Box[T <: Comparable[T]](val value: T) extends Comparable[Box[
    T]] {
  def compareTo(b2: Box[T]): Int = value.compareTo(b2.value)
}

class NumWrapper(val x: Int) extends Comparable[NumWrapper] {
  def compareTo(other: NumWrapper): Int =
    if(x==other.x) 0 else if(x > other.x) 1 else -1
}

val box1 = new Box(new NumWrapper(1))
val box2 = new Box(new NumWrapper(5))
box1 === box1  // true
box1 === box2  // false
box1 < box2    // true
box1 > box2    // false
```

This example makes use of some generic programming features that will be described in later chapters. For now, what should be noted is that the classes `NumWrapper` and `Box` acquire the concrete methods `<,>,===` of trait `Comparable`. The above example also shows that traits can effectively work as interfaces and mixins at the same time.

The interesting thing of using traits as mixins is that they *compose* (i.e., you can provide multiple mixins at the same time) and they are *stackable* (see the `Logger` example in Section 2.1.2). For the ”compose” part:

```
trait X
trait Y
trait Z

class C extends X with Y with Z
```

This is all about traits for now. We'll say more about them in the following chapters. We have also completed this quick tour of the basics of Scala. It is not meant to be exhaustive, and it is not the subject of this preparatory activity for the thesis either – it is provided just to make this report a little more self-contained.

Chapter 2

Advanced Features in Scala

Outline

1. Class construction, class linearization, member overriding, `super` resolution in traits, early definitions

2. Basics of type system: path-dependent types, type projection, singleton types

3. Advanced types: structural types, compound types, existential types, self types, abstract types

4. Generic programming: type parameters, type bounds, F-bounded polymorphism, type variance

5. Implicits: implicit parameters, implicit conversions/views, implicit resolution

2.1 More on OOP in Scala

In this section, we'll explore more detailed stuff about classes, traits, objects.

2.1.1 Class construction and linearization

A class may inherit from a base class and mix-in multiple traits at the same time. Now, we are interested in the construction order and in the ordered hierarchy of superclasses, because these notions will allow us to better specify a couple interesting aspects of traits.

Well, class construction happens in the following order:

1. Superclass' constructor
2. Traits' constructor, from left to right, with parents constructed first (and not constructed a second time)
3. Class constructor

Let's see an example:

```scala
class A { print("A") }
trait R { print("R") }
trait S extends R { print("S") }
trait T { print("T") }
trait U extends T with R { print("U") }
class B extends A with U with S { print("B") }

new B  // A T R U S B
```

The process is intuitive: elements specified on the left come first and thus are constructed first, and each element needs to construct its parents first.

We note that the construction order also defines the *linearization of a class*, that is, the process that determines the linear hierarchy of parents of a class. For the example above, we have

$$\begin{aligned} lin(B) &= B \succ lin(S) \succ lin(U) \succ lin(A) \\ &= B \succ (S \succ \cancel{R}) \succ (U \succ R \succ T) \succ A \\ &= B \succ S \succ U \succ R \succ T \succ A \end{aligned}$$

Note two things: first, the first R is elided because the second occurrence wins (R must be constructed at a higher point in the hiararchy); second, the construction order is the reverse of the linearization order.

This is intuitive too: when defining a class, the elements specified next (on the right) are down the hierarchy, thus they need to be constructed after the elements at the top of the hierarchy, and they also take the precedence in case of conflict (i.e., they override previous definitions).

2.1.2 Traits: member overriding and `super` resolution

Now we are ready for two interesting points about traits:

1. If multiple traits override the same member, the trait that wins is the one constructed last, i.e., the trait that is ”closer” to the defining class/object in the class linearization

2. In a trait, the method calls on `super` are resolved depending on the order in which traits are added. That is, the dispatched method implementation is the one of the next trait in the class linearization order.

Let’s exemplify these two aspects:

```
trait Logger {
  def log(msg: String)
}

trait ShortLogger extends Logger {
  val maxLength: Int
  val ellipsis = "..."

  abstract override def log(msg: String) {
    super.log(msg.take(maxLength)+ellipsis)
  }
}

trait WrapLogger extends Logger {
  val wrapStr = "|"
  abstract override def log(msg: String) {
    super.log(wrapStr + msg + wrapStr)
  }
}

trait ConsoleLogger extends Logger {
  override def log(msg: String) = println(msg)
}

val obj = new {
  val maxLength = 6
  override val ellipsis = ",,,"
} with ConsoleLogger with WrapLogger with ShortLogger

obj.log("linearization") // |linear,,,|
```

Some points should be underlined:

- First, note how the different behaviors are composed at instantiation time
- In `ShortLogger` and `WrapLogger`, the `log` method is decorated with `abstract override`, because you are – at the same time – overriding the method and calling *some* `super` implementation
- Note how the mixin order is relevant to the final result: if you switch `WrapLogger` and `ShortLogger`, you'll have the last occurrence of the wrapping string stripped
- Note how the chain of `log` calls flow from right to left in the trait list (or, equivalently, from bottom to top in the hierarchy as given by linearization)
- The `new {...}` part is an *early-definition* which is needed to concretize the `maxLength` abstract field before the `ShortLogger` can be constructed

2.1.3 Trait instantiation, refinement, early definitions

Scala provides a mechanism to instantiate traits and abstract classes once they have no abstract members.

```
trait A
trait B

new A with B {
  println("This is a refinement")
  def foo: String = "bar"
}
```

The block following `A` is a *refinement* (empty, in this case). A refinement is a mechanism to provide a delta (i.e., overrides or additional members) to a type. This actually defines a structural type (which will be covered next).

```
trait A {
  val x: Int
  require(x > 0)
}

val a = new A {
  val x = 10
} // java.lang.IllegalArgumentException: requirement failed
```

The problem here is that `A` is constructed *before* being refined. To solve this issue, we have to provide an early definition:

```
trait A {
  val x: Int
  require(x > 0)
}

val a = new { val x = 10 } with A
```

This works because, according to the rules for method overriding[1], an abstract member cannot override a concrete member.

2.2 Advanced types

2.2.1 Some preparatory definitions

- A (static) **type** is a set of information hold by the compiler about program entities
- A **binding** is a name used to refer to an entity
- Types can be located at certain **paths**
- According to the Scala Language Specification [2], a path is one of the following:
  1. The empty path ϵ
  2. `C.this` – where C refers to a class
  3. `p.x` – where p is a path and x is a stable member of p; a stable member is a package or a members introduced by (non-volatile) object or value definitions
  4. `C.super.x` – where x is a stable member of the superclass of the class referenced by C
- A **singleton type** has form `p.type` where path p points to a value conforming to `AnyRef`
- A **type projection** `T#x` refers to the type member x of type T
- A **type designator** refers to a named value type. A type designator can be
  - *Qualified*: has form `p.t` where p is a path and x is a named type (it is equivalent to `p.type#t`)
  - *Unqualified and bound to a package or class or object C*: `t` is a shorthand for `C.this.type#t`
  - *Unqualified and NOT bound to a package/class/object*: `t` is a shorthand for ϵ`.type#t`

2.2.2 Parameterized types

A *parameterized types* consists of a type designator `T` and n *type parameters* U_i:

```
T[U1,U2,...,Un]
```

2.2.3 Infix types

Any type which accepts two type parameters can be used as an *infix type*. Consider the following example with the standard map `Map[K,V]`:

```scala
val m: String Map Int = Map("a" -> 1, "b" -> 2)
// m: Map[String,Int] = Map(a -> 1, b -> 2)
```

2.2.4 Structural types

A *structural type* can be defined through refinement. In the refinement, you can add declarations or type definitions. You can also use refinement alone, without any explicit refined type; in that case, `{...}` is equivalent to `AnyRef{...}`.

```scala
def f(x: { def foo: String }) = x.foo
// f: (x: AnyRef{def foo: String})String

f(new { def foo = "bar" })
// String = bar
```

Structural types essentially enables a form of *duck typing*, where you can specify requirements on objects in terms of an exhaustive specification of supported methods and fields. However, this feature is implemented via reflection, so it comes with a cost.

2.2.5 Compound types

A compound type is one of the form

```
T1 with T2 ... with TN { R }
```

where the Ts are types and R is an optional refinement. A compound type C without any refinement is equivalent to `C {}`. Compare the following:

```
trait A; trait B
new A // Error: trait A is abstract; cannot be instantiated
new A { }    // Ok
new A with B // Ok
```

A more sophisticated example, which also makes use of existential types and type constraints, follows:

```
trait A; trait B; trait C extends B

val x: (A { def foo: Unit }) with
  ((X forSome {type X<:B}) { def bar: Unit }) =
  new A with C { def foo { }; def bar { } }
```

Compound types are also known as *intersection type* because a value, in order to belong to the compound type, must belong to all the individual types.

2.2.6 Existential types

An existential type has the form

```
T forSome { Q }
```

where Q is a sequence of type declarations (which may be constrained).

The underscore _ has many uses in Scala. One of them consists in providing a syntactic sugar for existential types:

```
val m1: Map[_,_<:List[_]] =
  Map(1 -> List('a','b'), "k" -> List(true,true))

val m2: Map[A,B] forSome {
   type A;
   type B <: List[C] forSome { type C }
} = m1
```

Note that such a use of existential types is related to the notion of *variance* (see Section 2.3.3). The connection between generic type variance and existential types has been pointed out in [3].

2.2.7 Self-types

Self types are used to constrain a trait or class to be used necessarily within a compound type that includes a type conforming to the self type. In other words, a trait or class can be used only when mixed in (together) with the self type.

```
trait A {
  def foo { }
}

trait B { self: A =>
  def bar = foo
}

trait C { self: A with B { def gulp } => }

new B { } // Error: illegal inheritance; self-type B
          //   does not conform to B's selftype B with A
new B with A // OK
new A with B // OK
new C with A with B { def gulp { } } // Error

trait Gulp { def gulp { } }
new C with A with B with Gulp // OK
```

2.2.8 Abstract type members

Just like it is possible to declare abstract fields and methods, a class or trait can declare *abstract types*.

```
trait Box {
  type TValue
  def peek: TValue
}

class StringBox(s: String) extends Box {
  override type TValue = String
  val peek = s
}

val sbox = new StringBox("xxx")
sbox.peek // String = xxx
```

Note that the previous example can be rewritten by using a type parameter.

2.3 Generic programming in Scala

Generic programming is all about defining generic entities and algorithms. In common sense, the term *generic* means ”belonging to a large group of objects” (source: `etymonline.com`). Thus, we may say that an entity or algorithm is generic when it or its properties can be found in many other entities or algorithms; the other way round works as well, i.e., an entity or algorithm is generic if many other entities or algorithms ”can be generated” from it.

Genericity involves some form of abstraction as a fully-specified entity is not generic by definition. The way to achieve genericity is by delaying the specification of certain details until a later time. When it comes to programming languages, three main forms of abstraction are given by:

1. types
2. parameters
3. abstract members

A type abstracts from the specific values belonging to that type. A parameter allows to abstract from specific parts of an entity or algorithm. An

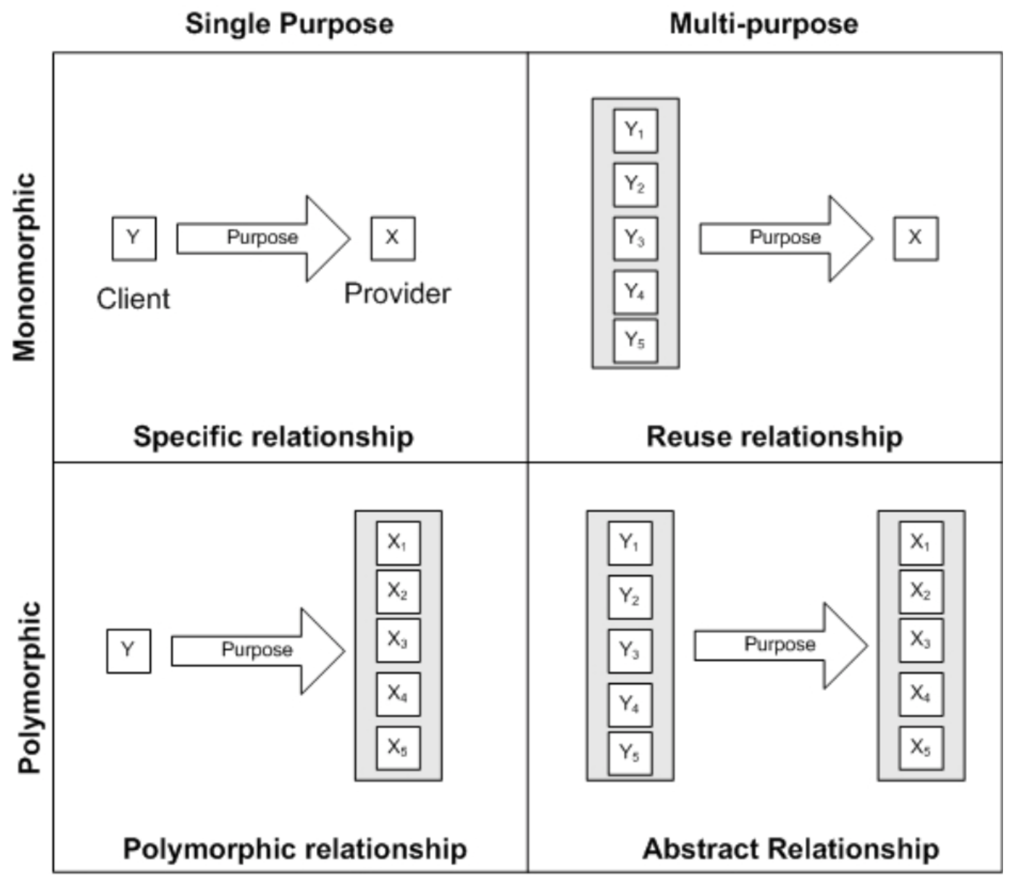

Figure 2.1: From specific to abstract relationships. Picture taken from http://www.erights.org/talks/categories/categories.html

abstract member formalizes the promise that the member will be concretized in future.

Scala's abstract type members (see Section 2.2.8) and type parameters combine the abstraction provided by types with the abstraction provided by parameters and abstract members, respectively.

Next, we'll cover some basic and advanced aspects of generic programming in Scala. We'll be quick as many of these features are mainstream.

2.3.1 Type parameters

Classes, traits, methods, and type constructors can accept type parameters.

```scala
// Generic method
def headOption[T](lst: List[T]): Option[T] =
  lst match { case h :: _ => Some(h); case _ => None }
headOption(List[Int]())      // None
headOption((3 to 7).toList) // Some(3)

// Generic trait
trait TBox[+T] {
  def value: T
}

// Generic class
class Box[+T](val value: T) extends TBox[T]

// Type constructor
type Cellar[T] = Map[String, TBox[T]]

val secretbox = new Box("xxx")
val cellar: Cellar[Any] = Map("secretbox" -> secretbox)
cellar("secretbox").value // Any = "xxx"
```

Note that, thanks to type inference, often you will not need to specify the type parameter.

2.3.2 Type bounds (bounded quantification)

Scala allows you to specify constraints to type variables:

- *Upper bound*: `T<:UB`: `T` must be a subtype of `UB`
- *Lower bound*: `T>:LB`: `T` must be a supertype of `LB`

Other bounds exist but we'll see them when talking about implicits.

Let's see an example of upper and lower bounds:

```scala
trait A
trait B extends A
trait C extends B

class Pair[T1 >: B, T2 <: B](val _1: T1, val _2: T2)

new Pair(new A{}, new C{}) // Pair[A,C]
new Pair(new B{}, new B{}) // Pair[B,B]
new Pair(new C{}, new C{}) // Pair[B,C] !!!!
new Pair[C,B](new C{}, new C{}) // Error: do not conform with
    constraint
```

Note that in Scala all types have a maximum upper-bound (`Any`) and a minimum lower-bound (`Nothing`).

F-bounded polymorphism and self-recursive types

Scala supports *F-bounded quantification* [4], i.e., a type parameter can be used in its own type constraint. A common application of F-bounded quantification is in *self-recursive types*, e.g., `T[U <: T[U]]`.

A self-recursive type is a type that is defined in terms of itself, for example:

```
case class Point(x: Double, y: Double) {
  // Positive recursion
  def move(x: Double, y: Double): Point

  // Negative recursion
  def isEqual(pt: Point): Boolean
}
```

With respect to positive recursion, we may want methods like `move` to return – in case of subtyping – an object of the same type as the receiver (and not always the type of the base class). This can be achieved with F-bounded types:

```
trait Movable[T <: Movable[T]] {
  def move(x: Double, y: Double): T
}

case class Point(val x: Double,
                 val y: Double) extends Movable[Point] {
```

```
  def move(x: Double, y: Double) = Point(this.x+x, this.y+y)
}

val obj = Point(1,1).move(2,2).move(2,1)
// obj: Point = Point(5.0,4.0)
```

2.3.3 Type variance

Type variance is a feature that integrates parametric and subtype polymorphism in OOPLs [3].

Given a type T with type components U_i (i.e., type parameters or type members), *variance* refers to the relation between the subtyping of T and the subtyping of its type components U_i. For example: is `List[Rect]` a subtype of `List[Shape]`? Or viceversa? Or are they unrelated?

Here are the possibilities:

- `T[A]`: T is invariant in A; i.e., T does not vary with respect to A
  So, for example, if `A` and `B` are different types (possibly in a subtyping relationship), then `List[A]` and `List[B]` are unrelated.

- `T[+A]`: T is covariant in A; i.e., T varies in the same direction as the subtyping relationship on A
  So, for example: if `Rect` is a subtype of `Shape`, then `List[Rect]` is a subtype of `List[Shape]`

- `T[-A]`: T is contravariant in A; i.e., T varies in the opposite direction to the subtyping relationship on A.
  So, for example: if `Rect` is a subtype of `Shape`, then `List[Rect]` is a supertype of `List[Shape]`

Now, let's consider a 1-ary function type `Function1[-T,+R]`. It is covariant in its return type and contravariant in its input type. Functions should conform to such a variance scheme to support safe substitutability. In fact, we ask: when is it safe to substitute a function `f:A=>B` with a function `g:C=>D`?

- `val a: A = ...; f(a)`
  The parameters provided by the users of `f` must be accepted by `g` as well, thus `C>:A` (contravariance as $C >: A \Rightarrow g <: f$)

- `val b: B = f(..)` The value returned to the users fo `f` must be support at least the interface of `B`, thus `D<:B` (covariance as $D <: B \Rightarrow g <: f$)

So, it is common to refer to parameters as *contravariant positions* and to return types as *covariant positions*. The use of variance annotations allows the compiler to check that the generic types are used in a manner which is consistent to these rules.

```
trait A[+T] { def f(t: T) }
// Error: covariant type T occurs in contravariant position
```

Last but not least, it is important to remark that mutability makes covariance unsound. Let's assume that `scala.collection.mutable.ListBuffer` were covariant:

```
import scala.collection.mutable.ListBuffer
val lstrings = ListBuffer("a","b")  // Type: ListBuffer[String]
val lst: ListBuffer[Any] = lstring  // It would fail (OK under our
    assumptions)
lst += 1     // Legal to add an Int to a ListBuffer[Any]
             // But 'lst' actually points to a list of string!!!!!
```

2.3.4 Abstract types vs. type parameters

In many cases, code that uses type parameters can be rewritten with abstract types, and viceversa. This is another situation where object-oriented programming and functional programming merge nicely in Scala.

The encoding from type parameters to type members, in the case of a parameterized class C, is described in [1] and has 4 parts:

1. Class definition

```
class C[+T1 >: LB1 <: UB1, -T2 >: LB2 <: UB2] { ... }

// Maps to

class C {
  type T1 >: LB1 <: UB1
  type T2 >: LB2 <: UB2
} // Note: variance is dealt with when referring to type C
```

2. Class instantiation (with concrete type *Int*)

```
new C[Int]

// Maps to

new C { type t = Int }
```

3. Generic base class construction (with concrete type *Int*)

```
abstract class C[T] { val v: T }
class D extends C[Int] { val v = 1 }

// Maps to

abstract class C { type T }
class D extends C { type t = Int; val v = 1 }
```

4. Parameterized types are rewritten with refinements to account for variance; consider `Function1[-T1,+R]`

```
// Rule 1: trait Function1[-T1,+R]
//    ===> trait Function1 { type T1; type R }

trait A; trait B extends A; trait C extends B; trait D extends
    C

val f: Function1[C,B] = new Function1[A,D]{ }

// Maps to
```

```
val f: Function1 { type T1 >: C; type R <: B } = // This rule
    (4)
  new Function1 { type T1 = A; type R = D } // Rule 2
```

Here are a few points on the similarity and difference between abstract types and type parameters:

- Usually, type parameters are used when the concrete types are to be provided when the class is instantiated, and abstract types are used when the types are to be supplied in a subclass
- The use of type parameters is not very scalable: if you have a lot of type parameters code tends to clutter, while abstract types help to keep the code clean
- The previous point is particularly true when type parameters are subject to (possibly verbose) type constraints
- Abstract type members are a little more verbose than a lightweight use of type parameters
- At the time of type instantiation, with type parameters you do not see the name of the instantiating parameter, thus you may lose some readability

```
trait Graph[NodeType, EdgeType]
// ... maybe in another file ...
new Graph[String, Double] { ... }
// Not very clear what String and Double refer to

trait Graph { type NodeType; type EdgeType }
// ... maybe in another file ...
new Graph {
  type NodeType = String
  type EdgeType = Double
}
```

2.4 Implicits

The Scala *implicit system* is a static feature that allows programmers to write concise code by leaving the compiler with the duty of inferring some missing information in code. This is achieved through the arrangement, according to a set of scoping rules, of code providing that missing data and a well-defined lookup mechanism.

An implicit lookup can be triggered in two cases:

1. a *missing parameter list* in a method call or constructor (if that parameter list is declared as `implicit`)

2. a *missing conversion* between from a type to another (which is *necessary* for the program to type-check) – this happens automatically in three situations (unless an implicit conversion has already been performed)

   (a) when the type of an expression differs from the expected type

   (b) in `obj.m` if member `m` does not exist

   (c) when a function is invoked with parameters of the wrong type

Then, using the `implicit` keyword, you can provide implicit data and implicit conversions. A note on terminology: an implicit conversion function `T=>U` from type `T` to type `U` is also called an *implicit view* (because it allows to *view* a `T` as a `U`).

The following listing exemplify the different situations where the implicit mechanism triggers:

```
// A) MISSING PARAMETER LIST

def m(implicit s: String, i: Int) = s+i

m("a", 0)     // "a0"   (You can still explicitly provide them)
m             // Error: could not find implicit value for 's'
implicit val myString = "x"
m             // Error: could not find implicit value for 'i'
```

```
implicit val myInt = 7
m            // "x7"

// B) MISSING CONVERSION

// B1) Expression of unexpected type

case class IntWrapper(x: Int) {
   def ^^(p: Int): Int = math.pow(x,p).toInt
}
implicit def fromIntToIntWrapper(x: Int) = IntWrapper(x)
val iw: IntWrapper = 8 // iw: IntWrapper = IntWrapper(8)

// B2) Non-existing member access

2^^5       // 32 (there is no ^^ method in Int)

// B3) Function call with wrong param types

def pow(iw: IntWrapper, power: Int) = iw^^power

pow(3, 4)  // 81 (pow accepts an IntWrapper, not an Int)
```

2.4.1 Implicit scope

Scala defines well-defined rules for what concerns implicit lookup. First of all, ambiguity in implicit resolution results in a compilation error.

```
implicit def x = 'x'
implicit val y = 'y'

def f(implicit c: Char) { }

f // Error: ambiguous implicit values:
  //   both method x of type => Char
  //   and value y of type => Char
  //   match expected type Char
```

Secondly, when the compiler looks for `implicit` data or views from a certain lookup site

1. It first looks if there is a conforming implicit entity among the unqual-

ified bindings
E.g., if there is an object `o` in scope and that object has an implicit field member `o.i`, that value is *not* considered because it must be accessible as a single identifier

2. Then, it looks
i) in case of implicit parameter lookup, at the implicit scope of the implicit parameter type, and
ii) in case of an implicit conversion, at the implicit scope of the *target* type of the conversion

The *implicit scope* of a type is the set of the companion objects of its associated types. For a type, its associated types include the base classes of its parts. The parts of a type `T` are (according to the SLS[2]):

- if it is a parameterized type, its type parameters
- if it is a compound type, its component types
- if it is a singleton type p`.type`, the parts of p (e.g., the enclosing singleton object)
- if it is a type projection, i.e., `T = A#B`, the parts of `A` (e.g., the enclosing class or trait)
- otherwise, just `T`

Let's verify these rules:

```scala
// A) Companion objects of the type parameters
trait A; object A { implicit def itoa(i: Int) = Map[A,A]() }

def f[X,Y](m: Map[X,Y]) = m

f[A,A](1)  // OK. Converts the Int value to a Map[A,A]

// B) Companion objects of types in an intersection type

trait B
trait C; object C {
  implicit def conv(i: Int) = new B with C { def foo { } }
}

def g(arg: C with B) = arg

g(1)  // OK. Converts the Int value to a B with C { def foo:Unit }

// C) Parts of the object of the singleton type

abstract class Provider[T](_x: T) { implicit val x = _x }

object P extends Provider(this)

def f(implicit p: P.type) = { p }
f(P) // OK.

// D) Parts of the type projecting from

object x {
  case class Y(i: Int)

  implicit val defaultY = Y(0)
}

implicitly[x.Y]      // x.Y = Y(0)
implicitly[x.type#Y] // x.Y = Y(0)
// In this case, the type projecting from is a singleton type

// Another example for D), with package objects

package a.b.c { class C }
package a.b {
  package object c {
    implicit val defaultC: C = new C
  }
}

implicitly[a.b.c.C] // OK
```

Where `implicitly[T](implicit e: T) = e` is defined in `scala.Predef`.

In general, it is best not to abuse the flexibility of the implicit scoping. As a rule of thumb, implicits should be put on the package object or in a singleton object with name `XxxImplicits`.

2.4.2 Implicit classes

Implicit classes[1] are classes declared with the `implicit` keyword. They must have a primary constructor that takes exactly one parameter. When an implicit class is in scope, its primary constructor is available for implicit conversions.

```
implicit class Y { } // ERROR: needs one primary constructor param

implicit class X(val n: Int) {
  def times(f: Int => Unit) = (1 to n).foreach(f(_))
}

5 times { print(_) } // 12345
```

It is interesting to note that an implicit class can be generic in its primary constructor parameter:

```
implicit class Showable[T](v: T) { val show = v.toString }

Set(4,7) show  // String = Set(4, 7)
false show     // String = false
```

2.4.3 More on implicits

Context bound

A type parameter `T` can have a *context bound* `T : M`, which requires the availability (at lookup site, not at the definition site) of an implicit value of type `M[T]`.

Let's consider an example. Scala provides a trait `scala.math.Ordering[T]` which has an abstract method `compare(x:T, y:T):Int` and provides a set of methods built on it, such as `min, max, gt, lt` and so on. Moreover, implicit values for the most common data types are defined in the `Ordering`

[1]Reference: `http://docs.scala-lang.org/overviews/core/implicit-classes.html`

companion object. Then, we may want to define a function that requires to work on types for which some notion of ordering is defined. For example, let's define a function the returns the shortest and longest string in a collection:

```
implicitly[Ordering[String]] // Predefined ordering:
//  Ordering[String] = scala.math.Ordering$String$@65c5fae6
// Let's override it with a custom ordering for strings

implicit val strOrdering = new Ordering[String]{
  def compare(s1: String, s2: String) = {
    val size1 = s1.length;
    val size2 = s2.length;
    if(size1<size2) -1 else if(size1 > size2) 1 else 0
  }
}

def minAndMax[T : Ordering](lst: Iterable[T]) = (lst.min, lst.max)

minAndMax(List("hello","x","aaa")) // (String, String) = (x, hello)
minAndMax(List("hello","x","aaa"))(Ordering.String)
// (String, String) = (aaa,x)
```

Note that the `minAndMax` accepts an `Ordering[T]` explicitly. This reveals that context bounds are actually a syntactic sugar; `minAndMax` is rewritten as follows:

```
def minAndMax[T](lst: Iterable[T])
                (implicit ord: Ordering[T]) = (lst.min, lst.max)
```

Generalized type constraints

Generalized type constraints are objects that provide evidence that a constraint hold for two types. As it is stated in the Scala API documentation [5]:

- `sealed abstract class =:=[-From,+To] extends (From)=>To`

  An instance of `A =:= B` witnesses that type `A` is equal to type `B`

- `sealed abstract class <:<[-From,+To] extends (From)=>To`
  An instance of `A <:< B` witnesses that `A` is a subtype of `B`

Note that the type constructor `=:=[A,B]` can be used with infix notation `A=:=B`.

In practice, these constraints are used through an *implicit evidence parameter*. This allows, for example, to enable a method in a class only under certain circumstances

```
case class Pair[T](val fst:T, val snd:T){
  def smaller(implicit ev: T <:< Ordered[T]) = if(fst < snd) fst
   else snd
}

class A

case class B(n: Int) extends Ordered[B] {
  def compare(b2: B) = this.n - b2.n
}

val pa = Pair(new A, new A)
pa.smaller // Error: Cannot prove that A <:< Ordered[A].

val pb = Pair(B(3), B(6))
pb.smaller // B = B(3)
```

In this case, instance method `pair.smaller` can be invoked only for pairs of a type `A` that is a subtype of `Ordered[A]`. The implicit parameter is said to be an *evidence* parameter in the sense that the resolution of the implicit value represents a proof that the constraint is satisfied (so these are also known as *reified type constraints*).

`TypeTags` and reified types

The Java compiler uses *type erasure* in the implementation of generics. This means that at runtime there is no information about the type parameters of generic classes. Scala also implements type erasure to ease integration with Java.

```
import scala.reflect.runtime.universe._

def f[T](lst: List[T]) = lst match {
  case _:List[Int] => "list of ints";
  case _:List[String] => "list of strs";
}
// warning: non-variable type argument Int in type pattern List[Int
    ]
// (the underlying of List[Int]) is unchecked since it is
    eliminated
// by erasure:       case _:List[Int] => "list of ints";
//                            ^

f(List("a","b")) // "list of ints"
```

Note that the ability to work with generic types at runtime has been a subject of research for some time, also due to performance implications, as explained in [6].

To solve this issue, Scala provides `TypeTags` (which replaced `Manifests` in Scala 2.10), which are used together with the implicit mechanism to provide at runtime the type information that would otherwise be available only at compile-time.

```
import scala.reflect.runtime.universe.{TypeTag, typeOf}

def f[T : TypeTag](lst: List[T]) = lst match {
  case _ if typeOf[T] <:< typeOf[Int] => "list of ints"
  case _ if typeOf[T] <:< typeOf[String] => "list of strings"
}

f(List(1,2,3))   // "list of ints"
f(List("a","b")) // "list of strings"
```

Note the use of context bound. When an implicit value of type `TypeTag[T]` is required, the compiler provides it automatically.

Chapter 3

Advanced Programming Techniques in Scala

Outline

1. ”Pimp my library” pattern
2. Type Classes
3. Service-oriented component model via traits, abstract members, self-types, mixin composition
4. Cake Pattern
5. Family Polymorphism
6. Internal DSL development

3.1 ”Pimp my library” pattern

The *Pimp my library* pattern is a common technique, based on implicits, aimed at extending existing types with additional methods and fields.

Let’s consider an example in the Scala standard library. By default, you have access to a facility for generating a `Range` from an `Int`:

```scala
val range = 5 to 10 // Range.Inclusive = Range(5,6,7,8,9,10)
```

This works because object `Predef` (which is implicitly imported in all Scala compilation units) inherits from trait `LowPriorityImplicits`, which defines many implicit conversion methods and, in particular, a conversion method from `Int` to `RichInt`. Then, `RichInt` defines a few utility methods, including `to(end:Int):Inclusive` to produce an inclusive range.

The pattern is clear: when we need to extend some existing type, we can:

1. Define a type with the "extension methods" that express the new intended behavior

2. Define an implicit conversion function from the original type to the newly defined type, together with some policy for the import of these implicit views

This approach is particularly useful when the original type cannot be instantiated (e.g., because the class is `final` or `sealed`).

In summary, this simple idiom allows you to adapt (cf. Adapter design pattern), decorate (cf. Decorator design pattern), or extend existing classes in a transparent way (thanks to implicit views which are applied by the compiler at compile-time).

As an example, let's extend ints with a `times` method which repeats an action for the provided number of times.

```scala
implicit class MyRichInt(private val n: Int) extends AnyVal {
  def times(f: =>Unit) = new Range.Inclusive(1,n,1).foreach{_=>f}
}

5 times { print('a') } // aaaaa
```

The implicit class (see Section ??) `MyRichInt` has been defined as implicit view from `Int`s to instances of itself. As a side note, it is also a *value class* (as it extends `AnyVal`), so it wins some efficiency by avoiding object allocation at runtime.

3.2 Type classes

Type classes are a feature popularized in the Haskell programming language. A *type class* provides an abstract interface for which it is possible to define many type-specific implementations.

In Scala, the type class idiom consists in[7]:

1. A trait that defines the abstract interface
2. A companion object for the type class trait that provides a set of default implementations
3. Methods using the typeclass, declared with a context bound

As an example, let's try to implement the `Ordering[T]` type class (similar to the one in the Scala standard library):

```scala
// Type class
trait Ordering[T] {
  // Abstract interface
  def compare(t1: T, t2: T): Int

  // Concrete members
  def ===(t1: T, t2: T) = compare(t1,t2) == 0
  def <(t1: T, t2: T) = compare(t1,t2) < 0
  def >(t1: T, t2: T) = compare(t1,t2) > 0
  def <=(t1: T, t2: T) = <(t1,t2) || ===(t1,t2)
  def >=(t1: T, t2: T) = >(t1,t2) || ===(t1,t2)

  def max(t1: T, t2: T): T = if(>=(t1,t2)) t1 else t2
  def min(t1: T, t2: T): T = if(<=(t1,t2)) t1 else t2
}

// Companion object with implicit, default implementations
object Ordering {
  implicit val intOrdering = new Ordering[Int]{
    def compare(i1: Int, i2: Int) =
      if(i1<i2) -1 else if(i1>i2) 1 else 0
  }
}

// Usage
```

```
def min[T : Ordering](lst: List[T]): T = {
  if(lst.isEmpty) throw new Exception("List is empty")

  val ord = implicitly[Ordering[T]]
  var minVal = lst.head
  for(e <- lst.tail) {
    if(ord.<(e,minVal)) minVal = e
  }
  minVal
}

min(List(5,1,4,7,-3))  // Int = -3
```

Note the use of the context bound constraint on the type parameter for `min` and how we need to perform an implicit lookup to to get the `Ordering[T]` instance on which we can invoke the methods of the typeclass interface.

It is clear that we could have achieved a similar result by using inheritance. However, type classes have some benefits:

- You can provide many implementations of the type class interface for the same type.
- Type classes separate the implementation of an abstract interface from the definition of a class. Thanks to this separation of concern, you can easily adapt existing types to the type class interface.
- By playing on the scoping rules for implicits, you can override the default type class implementations
- A type may implement multiple type classes without cluttering its class definition. Then, you can specify multiple type bounds on a type variable `T : CB1 : CB2 : ...`, requiring that type to provide an implicit implementation object for multiple type classes.

3.3 Component modelling and implementation

In computer science, the notion of *component* and *component-oriented software development* have been interpreted in many different ways, i.e., according to many different (and possibly not formalized) component models. A *component model* defines:

1. components – telling things such as what is a component, how to specify a component, how to implement a component, what a component's runtime lifecycle consists in
2. connectors – i.e., mechanisms for component composition and component interaction

Generally, for our purpose, a component can be abstractly defined as a *reusable* software entity with well-defined boundaries, as defined by a software *interface*. An interface is a means of specifying both *provided services* and *required services* (i.e., *dependencies*).

The article [1] supports the idea that for building reusable components in a scalable way, three features or abstractions are key:

1. Abstract type members (see Section 2.2.8)
2. Explicit self-types (see Section 2.2.7)
3. Modular mixin composition (see Section 1.5.2)

Thus, according to such a proposed service-oriented component model, we have the following mappings:

- Concrete classes $\Longleftrightarrow$ components
- Abstract classes or traits $\Longleftrightarrow$ component interfaces
- Abstract members $\Longleftrightarrow$ required services

- Concrete members $\Longleftrightarrow$ provided services

In this context, we can interpret abstract member overriding as a mechanism for providing required services. As concrete members always override abstract member, we get *recursively pluggable components where component services do not have to be wired explicitly* [1]. In this sense, mixin composition turns out to be a flexible approach for the assembly of component-based systems.

And where do self-types fit into this frame? Well, self-types are a more concise alternative to abstract members (read "required services"), with some differences. Self-types can be seen as an effective way to specify *component dependencies* as they ensure that when a component is being instantiated, it must be connected with the specified dependency. Note that while the self-type is one, thanks to compound types (see Section 2.2.5) you can provide for multiple dependencies.

Let's visualize these concepts with an example:

```
class Account(val id: String,
              var total: Double = 0)

trait AbstractItem {
  def name: String
  def price: Double
}
case class Item(name: String, price: Double) extends AbstractItem

// ****************************
// *** COMPONENT: Inventory ***
// ****************************
trait Inventory {
  type TItem <: AbstractItem

  def items: Set[TItem]
  def availability(item: TItem): Int
  def changeItems(m: Map[TItem,Int]): Unit

  def itemsWithAvailability: Map[TItem,Int] =
    items.map(it => (it, availability(it))).toMap
```

```
}

trait InventoryImpl extends Inventory {
  private var _items = Map[TItem,Int]()

  def items: Set[TItem] = _items.keySet
  def availability(it: TItem) = _items.getOrElse(it,0)
  def changeItems(m: Map[TItem,Int]) {
    m.keys.foreach(k => {
      val num = m.getOrElse(k, 0)
      val currNum = availability(k)
      if(currNum > 0){
         val newNum = currNum + num
         require(newNum >= 0)
         _items += k -> newNum
      } else if(num>0) {
         _items += k -> m(k)
      }
    })
  }
}

// ***********************
// *** COMPONENT: Cart ***
// ***********************
// Depends on the Inventory component
trait Cart {
  self: Inventory =>

  def changeCartItem(item: TItem, num: Int): Unit
  def cartContent: Map[TItem,Int]
  def resetCart: Unit
  def checkout(acc: Account): Unit
}
trait CartImpl extends Cart {
  inv: Inventory =>

  private var _items = Map[TItem,Int]()

  def changeCartItem(it: TItem, n: Int) = {
     inv.changeItems(Map(it -> (-n)))
     _items += (it -> n)
  }
  def cartContent = _items
  def resetCart = {
    _items.keys.foreach(k => {
      inv.changeItems(Map(k -> _items(k)))
    })
    _items = Map()
  }
  def checkout(acc: Account) {
    _items.keys.foreach(k => acc.total = acc.total - k.price)
    _items = Map()
  }
}

// ******************************
// *** COMPONENT: Application ***
// ******************************
// Depends on the Cart and Inventory components
trait ShoppingSystem { this: Cart with Inventory => }

// ********************************************
// *** Application with component instances ***
// ********************************************
object Shopping extends ShoppingSystem with
  CartImpl with InventoryImpl {
  type TItem = Item
}
val acct = new Account("Bob", 70)
val item1 = Item("Book", 9.99)
val item2 = Item("Bonsai", 69.99)

Shopping.changeItems(Map(item1 -> 3, item2 -> 1))
Shopping.changeCartItem(item1, 2)
Shopping.availability(item1)         // 1
// Shopping.changeCartItem(item1, 2) // Exception
Shopping.resetCart
Shopping.availability(item1)         // 3
Shopping.changeCartItem(item2, 1)
Shopping.checkout(acct)
acct.total                           // 0.010000000005116
Shopping.cartContent                 // Map()
Shopping.availability(item2)         // 0
```

This is a toy example, but it points out an approach to service-oriented component development. Note that he implementations of the components are `traits`; if they were classes, they could not be mixed in. Also, note how the concrete type of `TItem` is specified at the last moment when defining

`Shopping`. Then, it is interesting to see how the application logic component (`ShoppingSystem`) is defined as dependent on the other components (via compound self-type). As a result, the application façade object can be used as a cart or as an inventory in a inheritance-like (*is-a*) fashion. It is quite weird and not very effective for what concerns conceptual integrity: it would be better to have the application object *delegate* these functionalities to its components. We should apply the principle (from GOF) *Favor object composition over class inheritance*, and we come up with the Cake pattern.

3.4 Cake Pattern

The last section pointed out that a better way to satisfy component dependencies is via composition (rather than inheritance). Let's adjust that example according to the *Cake pattern*. In this pattern [8]:

- You define components as traits, specifying dependencies via self-types
- Then, each component includes
  - A trait that defines the service interface
  - An `abstract val` that will contain an instance of the service
  - Optionally, implementations of the service interface

Thus:

```scala
class Account(val id: String,
              var total: Double = 0)

trait AbstractItem {
  def name: String
  def price: Double
}
case class Item(name: String, price: Double) extends AbstractItem

// ****************************
// *** COMPONENT: Inventory ***
// ****************************
```

```scala
trait InventoryComponent {
  val inventory: Inventory

  type TItem <: AbstractItem

  trait Inventory {
    def items: Set[TItem]
    def availability(item: TItem): Int
    def changeItems(m: Map[TItem,Int]): Unit

    def itemsWithAvailability: Map[TItem,Int] =
      items.map(it => (it, availability(it))).toMap
  }

  trait InventoryImpl extends Inventory {
    private var _items = Map[TItem,Int]()

    def items: Set[TItem] = _items.keySet
    def availability(it: TItem) = _items.getOrElse(it,0)
    def changeItems(m: Map[TItem,Int]) {
      m.keys.foreach(k => {
        val num = m.getOrElse(k, 0)
        val currNum = availability(k)
        if(currNum > 0){
           val newNum = currNum + num
           require(newNum >= 0)
           _items += k -> newNum
        } else if(num>0) {
           _items += k -> m(k)
        }
      })
    }
  }
}

// ***********************
// *** COMPONENT: Cart ***
// ***********************
// Depends on the Inventory component
trait CartComponent { invComp: InventoryComponent =>
  val cart: Cart

  trait Cart {
    def changeCartItem(item: TItem, num: Int): Unit
    def cartContent: Map[TItem,Int]
    def resetCart: Unit
    def checkout(acc: Account): Unit
  }

  trait CartImpl extends Cart {
    val inv = invComp.inventory // NOTE: ACCESS TO ANOTHER
   COMPONENT'S IMPL
    private var _items = Map[TItem,Int]()

    def changeCartItem(it: TItem, n: Int) = {
       inv.changeItems(Map(it -> (-n)))
       _items += (it -> n)
    }
    def cartContent = _items
    def resetCart = {
      _items.keys.foreach(k => {
        inv.changeItems(Map(k -> _items(k)))
      })
      _items = Map()
    }
    def checkout(acc: Account) {
      _items.keys.foreach(k => acc.total = acc.total - k.price)
      _items = Map()
    }
  }
}

// ******************************
// *** COMPONENT: Application ***
// ******************************
// Depends on the Cart and Inventory components
trait ShoppingComponent {
  this: CartComponent with InventoryComponent =>
}

// ********************************************
// *** Application with component instances ***
// ********************************************
object Shopping extends ShoppingComponent with
  InventoryComponent with CartComponent {
  type TItem = Item
  val inventory = new InventoryImpl { }
  val cart = new CartImpl { }
}

val inv = Shopping.inventory
val cart = Shopping.cart
val acct = new Account("Bob", 70)
val item1 = Item("Book", 9.99)
val item2 = Item("Bonsai", 69.99)

inv.changeItems(Map(item1 -> 3, item2 -> 1))
cart.changeCartItem(item1, 2)
inv.availability(item1)          // 1
// cart.changeCartItem(item1, 2) // Exception
cart.resetCart
inv.availability(item1)          // 3
cart.changeCartItem(item2, 1)
cart.checkout(acct)
acct.total                       // 0.010000000005116
cart.cartContent                 // Map()
inv.availability(item2)          // 0
```

Notes:

- The `Shopping` object centralizes the **wiring** of components by implementing the components' abstract `vals` (`inventory` and `cart`)
- Note how the component implementations "receive" their dependencies (i.e., the instances of other components' implementations) through the (abstract) `val` fields
- As the component implementation instances do not need dependency injection via constructor parameters, we can instantiate them directly (*without manual wiring*). In other words, we just have to choose an implementation and we do not need to do any wiring as these components have already been wired
- The component instance `vals` can be declared `lazy` to deal with initialization issues

As design patterns should not be confused with their implementations, we note that the previous example just shows one particular realization of some more general patterns which are a design response to the following issues (or *forces*):

- How to flexibly build systems out of modular components
- How to declaratively specify the dependencies among components
- How to wire components together to satisfy the dependencies

Thus, the Cake pattern can be described more precisely as a Scala-specific pattern for dependency injection and component composition.

We could play with Scala features to morph this pattern into multiple variations and possibly communicate better our intents:

```scala
trait ComponentA {
  val a: A

  class A
}

trait ComponentB {
  val b: B

  class B
}

object ComponentC { type Dependencies = ComponentB with ComponentC
   }
trait ComponentC { self: ComponentC.Dependencies =>
  val c: C

  class C {
    // uses 'a' and 'b' internally
  }
}

trait ABCWiring1 extends ComponentA with ComponentB with ComponentC
    {
  lazy val a = new A
  lazy val b = new B
  lazy val c = new C
}

trait ApplicationWiring extends ABCWiring1

trait ApplicationComponents extends ComponentA with ComponentB with
    ComponentC
```

```scala
object Application extends ApplicationComponents with
    ApplicationWiring {
  println(s"$a $b $c")
}
```

The name of the "Cake pattern" brings to mind some notion of *layering* which may refer to the way in which the components are *stacked* to compose a full application, or a notion of component stuffing where a component trait includes interfaces, implementations, and wiring plugs.

3.5 Family Polymorphism

As it is a challenge to model families of types that vary together, share common code, and preserve type safety [8], *family polymorphism* has been proposed for OOPLs as a solution to supporting reusable yet type-safe mutually recursive classes [9].

To contextualize the problem, let's consider an example of graph modelling as in the original article by Ernst [10]. We would like to implement classes for

- a basic `Graph` with `Node`s and `Edge`s
- a `ColorWeightGraph` where `Node`s are colored (`ColoredNode`) and `Edge`s are weighted (`WeightedEdge`).

but we would like to do so in a way that:

- We can reuse base behaviors
- We cannot mix elements by different kinds of graphs

Let's attempt a solution without family polymorphism:

```scala
// ABSTRACT GRAPH
trait Graph {
  var nodes: Set[Node] = Set()
  def addNode(n: Node) = nodes += n
}
trait Node
abstract class Edge(val from: Node, val to: Node)

// BASIC GRAPH
class BasicGraph extends Graph
class BasicNode extends Node
class BasicEdge(from:BasicNode, to:BasicNode) extends Edge(from,to)

// GRAPH WITH COLORED NODES AND WEIGHTED EDGES
class ColorWeightGraph extends Graph {
  override def addNode(n: Node) = n match {
    case cn: ColoredNode => nodes += n
    case _ => throw new Exception("Invalid")
  }
}
class ColoredNode extends Node
class WeightedEdge(from: ColoredNode,
     to: ColoredNode, val d: Double) extends Edge(from,to)

val bg = new BasicGraph
val cg = new ColorWeightGraph
val n = new BasicNode
val cn = new ColoredNode
// cg.addNode(n) // Exception at runtime
bg.addNode(cn)   // Ok (type-correct),
                 // but we didn't want ColoredNodes in a BasicGraph
```

There are two problems here:

- There is no static constraint that restrict users to not mix up the two families
- In `ColorWeightGraph`, we cannot define `addNode` as accepting a `ColoredNode`, because covariant change of method parameter types is not allowed (it is a contravariant position)

These issues can be solved via family polymorphism:

```
trait Graph {
  type TNode <: Node
  type TEdge <: Edge
  type ThisType <: Graph

  trait Node { }

  trait Edge {
    var from: TNode = _; var to: TNode = _
    var fromWF: ThisType#TNode = _; var toWF: ThisType#TNode = _;
    def connect(n1: TNode, n2: TNode){ from = n1; to = n2 }
    def connectAcrossGraphs(n1: ThisType#TNode,
                            n2: ThisType#TNode){
       fromWF = n1;
       toWF = n2
    }
  }

  def createNode: TNode;
  def createEdge: TEdge
}

class BasicGraph extends Graph {
  override type TNode = BasicNode
  override type TEdge = BasicEdge
  override type ThisType = BasicGraph

  class BasicNode extends Node { }
  class BasicEdge extends Edge { }

  def createNode = new BasicNode;
  def createEdge = new BasicEdge
}

class ColorWeightGraph extends Graph {
  override type TNode = ColoredNode
  override type TEdge = WeighedEdge
  override type ThisType = ColorWeightGraph

  class ColoredNode(val color: String="BLACK") extends Node { }
  class WeighedEdge(val weight: Double=1.0) extends Edge { }

  def createNode = new ColoredNode;
  def createEdge = new WeighedEdge
}

val g = new BasicGraph
val cwg = new ColorWeightGraph
val e = g.createEdge
val n1 = g.createNode
val n2 = g.createNode

val cwe = cwg.createEdge
val cwn1 = cwg.createNode
val cwn2 = cwg.createNode

e.connect(n1,n2)        // Ok, within same graph (of same family)
cwe.connect(cwn1, cwn2) // Ok, within same graph (of same family)
//e.connect(n1,cwn2)    // ERROR!!! Cannot mix families

val g2 = new BasicGraph {}
val n21 = g2.createNode
val n22 = g2.createNode

// e.connect(n21,n22)
// ERROR: cannot connect an edge of a graph to nodes
//   of another graph, even if the graphs are of the same type

e.connectAcrossGraphs(n1,n22)
// Ok. Within the same family but across graph instances
// e.connectAcrossGraphs(n1,cwn1) // Of course, cannot mix families
```

Notes:

- `Graph` represents the *schema* of the family of graphs
- `BasicGraph` and `ColorWeightGraph` extend the `Graph` trait and represent two distinct families of graphs
- Families have type members introduced by `type` definitions
- Remember that when a class is defined inside a class, a different class is reified for each different instance of the outer class. Then, note how type projection has beeen used to allow the mixing of graphs (within the same family).

In this case, the family traits also provide factory methods. An alternative approach would be to define `BasicGraph` and `ColorWeightGraph` as singleton objects, and then import their type members into the current scope.

```
object BasicGraph extends Graph {
    override type TNode = BasicNode
    override type TEdge = BasicEdge
    override type ThisType = BasicGraph

    class BasicNode extends Node { }
    class BasicEdge extends Edge { }
}

import BasicGraph._

val n = new BasicNode
```

Finally, note that the nesting of types (`traits` inside `traits`) could be avoided by using type parameters instead of type members; however, this would bring to an explosion of type parameters.

3.5.1 Case study: an interpreter and simulator for aggregate programming

Prof. Mirko Viroli has prototyped a field calculus interpreter and simulator via family polymorphism. Here is a high-level representation via UML diagrams.

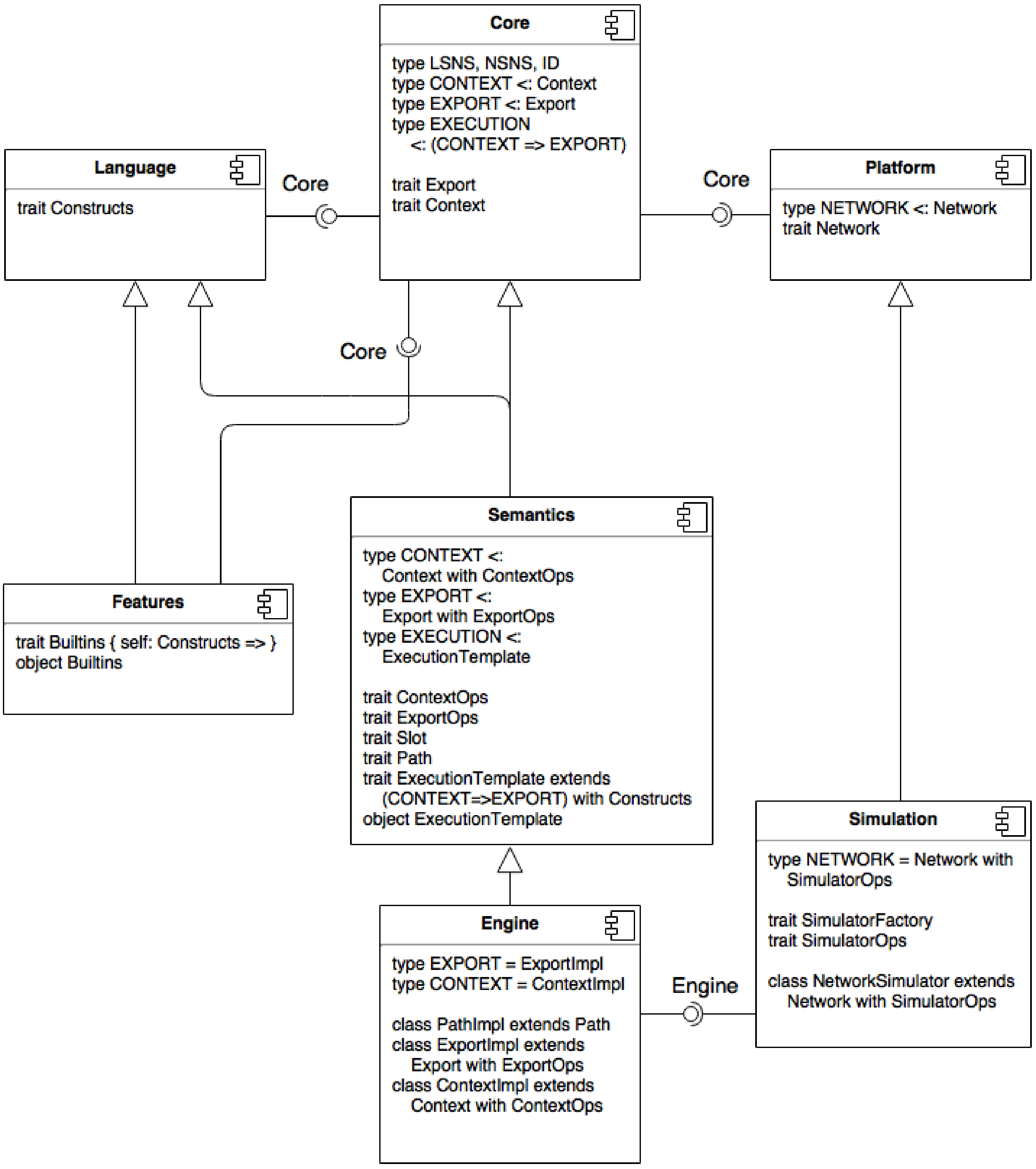

Figure 3.1: Example of family polymorphism in a real world scenario.

Here we have components that rapresent families of types. The extension relationship expresses the covariant specialization of a family. And the unfilled connections express required services (defined by self-types).

To build an "incarnation" of the system, the abstract types need to be concretized.

```scala
object Incarnation extends Engine with Features with Simulation {

  override type LSNS = String // names of local sensors
  override type NSNS = String // names of neighbor sensors
  override type ID = Int      // device IDs

  override type EXECUTION = Execution

  trait Execution extends ExecutionTemplate with Builtins {
    override type MainResult = Any
  }

}
```

3.6 Internal DSL Development

The combination of Scala's features makes it a discrete tool for building (internal) *domain specific languages (DSLs)*, even though it is . The features that support this kind of development include:

- Implicit system
- Expressive type system
- Syntactic sugar
- Functional programming features (e.g., lambdas)

3.6.1 On syntactic sugar

Let's recap many places where Scala provides syntactic sugar (not considering type-related ones):

```
// Tuples
val tu1 = Tuple5('a', "s", 7, true, 8.8)
val tu2 = ('a', "s", 7, true, 8.8)

// 2-elements tuples
val t1 = Tuple2[String,Double]("xxx", 7.5)
val t2 = "xxx" -> 7.5

// Anonymous functions
val add1: (Int,Int)=>Int = (x,y) => x+y
val add2: (Int,Int)=>Int = _+_

// apply()
add2.apply(7,3)
add2(7,3)

// update()
val m = scala.collection.mutable.Map[Int,String]()
m.update(1, "xxx")
m(1, "aaa")

// Leaving out "." for member access
List(5,3,2) map { _%2 == 0 } reverse

// Leaving out () for parameterless methods
def f() { }
f

// Using braces { } for arg lists
def f(a: Int)(b: Char)(c: Boolean){ }
f { 7 } ( 'z' ) { false }

// Methods with name ending in ":" are right-associative
// A right-assoc binary op is a method of its 2nd arg
Nil.::(2).::(1)
1 :: 2 :: Nil

case class X(x: Int = 0) { def `set:`(y: Int) = X(y) }
4 `set:` 7 `set:` X(3)        // X = X(4)
(4 `set:` (7 `set:` (X(3)))) // X = X(4)

// Setters
m.+=(7 -> "a")
m += 7 -> "a"
```

```
// Varargs
def sum(xs: Int*) = xs.foldLeft(0)(_+_)
sum(1,4,3,2) // 10

// Call-by-name parameters
def f(s: => String) = println(s)

f { (1 to 9).foldRight("")(_+_) }
```

3.6.2 On associativity and precedence

This material is taken from the Scala Language Specification [2].

Prefix operations `op e`

The prefix operator `op` must be one of the following: `+`, `-`, `!`, `~`. Prefix operations are equivalent to a postfix method call `e.unary_`*`op`*.

```
!false         // true
true.unary_! // false
4.unary_-      // -4

object a { def unary_~ = b }; object b { def unary_~ = a }
~(~(~a))  // b.type = b$@6c421123
```

Postfix operations `e op`

These are equivalent to the method call `e.op`

Infix operations `e1 op e2`

The first character of an infix operator determines the operator *precedence*. From lower to higher:

```
(All letters)
|
^
&
```

```
<  >
=  !
:
+  -
*  /  %
(All other special characters)
```

Associativity depends on the operator's last character. All operators are left-associative except those with name ending in ':' that are right-associative.

Precedence and associativity determine how parts of an expression are grouped:

- Consecutive infix operators (which must have the same associativity) associate according to the operator's associativity
- Postfix operators always have lower precedence than infix operators: `e1 op1 e2 op2 == (e1 op1 e2) op2`

Infix operations are rewritten as method calls: a left associative binary operator `e1 op e2` is translated to `e1.op(e2)`, whereas if the operator has arity greater than 1, it must be used as `e1 op (e2,...,en)`, which is translated to `e1.op(e2,...,en)`.

Here are some examples:

```
obj m1 p1 m2 p2 m3 p3 == ((obj m1 p1) m2 p2) m3 p3)
                      == obj.m1(p1).m2(p2).m3(p3)
```

3.6.3 Examples

A DSL for writing URIs

```
case class Uri(scheme: String = "http",
                  path: List[String] = List(),
                  querystring: Map[String,Any] = Map()) {
  def /(s: String) =
    this.copy(path = s :: path)

  def /?(t: (String,Any)) =
    this.copy(querystring = querystring + t)

  def &(t: (String,Any)) = this./?(t)

  override def toString = scheme + "://" +
    path.reverse.mkString("/") +
    (if(querystring.isEmpty) "" else "?" +
      querystring.keys.map(k => k+"="+querystring(k)).mkString("&")
    )
}

object UriDsl {
  def http  = Uri("http")
  def https = Uri("https")
  def ftp   = Uri("ftp")

  implicit def strToUrl(s: String): Uri = Uri(s)
};

import UriDsl._

http / "www.site.org" / "index.php" /? ("a"->7) & ("b"->true)
// Uri = http://www.site.org/index.php?a=7&b=true

"file" / "usr" / "bin" / "javac"
// Uri = file://usr/bin/javac
```

where `copy` is a method (automatically provided for case classes) that allows easy copying (possibly with differences).

The idea behind the previous example is simple: methods in object `UriDsl` work as entry points by building an `Uri` instance, then we chain method calls by having methods return a new object of the same kind.

A DSL for math operations

```
trait MathOperation

implicit class IntMathOperation(val n: Int) extends MathOperation {
  def \(d: Int) = Fraction(n,d)
}

trait MeanOperation extends MathOperation {
   def of(ns: Double*) = ns.foldLeft(0.0)(_+_) / ns.length
}

case class PowerOperation(base: Double) extends MathOperation {
  def by(exp: Double) = math.pow(base,exp)
}

case class Fraction(num: Int, den: Int) {
  def +(f2: Fraction) = {
    val m = Fraction.mcm(den, f2.den)
    Fraction((m/den)*num + (m/f2.den*f2.num), m)
  }
}

object Fraction {
  def simplify(f: Fraction): Fraction = {
     val d = gcd(f.num, f.den)
     Fraction(f.num/d, f.den/d)
  }

  def gcd(x: Int, y: Int): Int = // Euclid's algorithm
    if(x == y) x
    else if(x > y) gcd(x-y,y)
    else gcd(x, y-x)

  def mcm(x: Int, y: Int): Int = (x / gcd(x,y)) * y
}

object MathDsl {
  def mean = new MeanOperation { }

  def power(n: Double) = PowerOperation(n)

  def simplify(f: Fraction) = Fraction.simplify(f)
}
import MathDsl._

mean of (6, 10, 7, 9)        // Double = 8.0
power(2) by (mean of (4,6))  // Double = 32.0
1\2 + 4\3 + 1\6              // Fraction = Fraction(12,6)
simplify { 1\2 + 4\3 + 1\6 } // Fraction = Fraction(2,1)
```

A DSL for datetime operations

```scala
object DateTimeDsl {
  import scala.concurrent.duration._
  import java.time._;
  import java.time.LocalDateTime._;
  import java.time.DayOfWeek._

  val tomorrow = now.plusDays(1)
  val yesterday = now.plusDays(-1)
  def next(dw: DayOfWeek) = {
    val curr = now.getDayOfWeek.getValue
    val next = dw.getValue
    now.plusDays((-(curr-next)+7)%7)
  }

  case class DurationExt(val d: FiniteDuration) {
    def to(dt: LocalDateTime) = before(dt)
    def before(dt: LocalDateTime) = dt.plusNanos(-d.toNanos)
    def after(dt: LocalDateTime) = dt.plusNanos(d.toNanos)
  }

  implicit def finiteDurationToExt(fd: FiniteDuration) =
   DurationExt(fd)

  val mon = MONDAY; val tue = TUESDAY; val wed = WEDNESDAY;
  val thu = THURSDAY; val fri = FRIDAY;
  val sat = SATURDAY; val sun = SUNDAY;
}
import DateTimeDsl._; import scala.concurrent.duration._

(10 seconds) after now
(1 day) after next(sat)
// (1 day) after (next sat)
//   ERROR as 'next' requires a receiver or to be unary
// (1 day) after next sat
//   ERROR as 'after' requires 1 param but left-associativity
   groups as:
//      ((1 day) after) next sat
```

A DSL for expressing assertions

Reference for this section: `http://www.slideshare.net/holograph/a-field-guide-to-dsl-design-in-scala`.

The idea is to express assertions such as `obj shouldBe empty`.

```
sealed trait Predicate[-T] {
  def test: T => Boolean
  def failure: String
  def failureNeg: String

  type Self[-T] <: Predicate[T] // self-recursive type
  def negate: Self[T]
}
trait ModalPredicate[-T] extends Predicate[T] { self =>
  type Self[-T] = ModalPredicate[T]
  def negate = new ModalPredicate[T] {
    def test = self.test andThen { !_ }
    def failure = self.failureNeg
    def failureNeg = self.failure
  }
}
trait CompoundPredicate[-T] extends Predicate[T] { self =>
  type Self[-T] = CompoundPredicate[T]
  def negate = new CompoundPredicate[T] {
    def test = self.test andThen { !_ }
    def failure = self.failureNeg
    def failureNeg = self.failure
  }
}

implicit class EntryPoint[T](data: T) {
  def should(predicate: ModalPredicate[T]) =
    require(predicate.test(data), s"Value $data ${predicate.failure
    }")

  def shouldNot(predicate: ModalPredicate[T]) =
    require(!predicate.test(data), s"Value $data ${predicate.
    failureNeg}")

  def shouldBe(predicate: CompoundPredicate[T]) =
    require(predicate.test(data), s"Value $data ${predicate.failure
    }")

  def shouldBe(n: Null)(implicit ev: T <:< AnyRef) =
    require(data == null, s"Value $data is not null")

  def shouldNotBe(predicate: CompoundPredicate[T]) =
    require(!predicate.test(data), s"Value $data ${predicate.
   failureNeg}")

  def shouldNotBe(n: Null)(implicit ev: T <:< AnyRef) =
    require(data != null, s"Value $data is null")
}

def be[T](v: T): ModalPredicate[T] = new ModalPredicate[T] {
  def test: T => Boolean = _ == v
  def failure = s"is not $v"
  def failureNeg = s"is $v"
}

def be[T](v: Predicate[T]): ModalPredicate[T] = if(v==null) new
   ModalPredicate[T] {
  def test: T => Boolean = _==v
  def failure = s"is not null"
  def failureNeg = s"is null"
} else new ModalPredicate[T] {
  def test: T => Boolean = v.test
  def failure = v.failure
  def failureNeg = v.failureNeg
}

def not[T](pred: Predicate[T]): pred.Self[T] = pred.negate

def empty[T <: scala.collection.GenTraversableOnce[_]] = new
   CompoundPredicate[T] {
  def test: T => Boolean = _.isEmpty
  def failure = "is not empty"
  def failureNeg = "is empty"
}

implicit class BooleanPredicate(b: Boolean) extends
   CompoundPredicate[Boolean] {
  def test: Boolean => Boolean = _ == b
  def failure = s"is not $b"
  def failureNeg = s"is $b"
}

(8==8) shouldBe true
// (8==8) shouldBe false // Value true is not false
(8==8) should be(true)
```

```
(8+8) shouldNot be(10)
// "ciao" shouldNot be("ciao") // Value ciao is ciao

(null:AnyRef) shouldBe null
// List() shouldBe null // Value List() is not null
"ciao" shouldNotBe null
"ciao" shouldNot be(null)
// x shouldNotBe null // Value null is null

List() shouldBe empty
List() should be(empty)
// List() shouldNot be(empty) // Value List() is empty
// List(1) shouldBe empty     // Value List(1) is not empty
// List(1) should be(empty)   // Value List(1) is not empty
```

Chapter 4

The Scala Toolchain

Outline

1. Automation of common project-related activities (dependency management, build, test execution) with sbt
2. Testing with ScalaTest (unit and integration testing) and ScalaCheck (property-based testing)
3. Microbenchmarking and performance regression testing with ScalaMeter

4.1 Software project automation with sbt

sbt (the Scala Build Tool) is a tool that can be installed using your operating system's package manager. It consists of a command-line utility program that supports many project-related activities. You might want to use sbt for the support it provides for the following tasks:

- Management of the build process
- Packaging and deployment
- Project dependency management

- Automation of common activities such as
  - Quick experiments on the project code via REPL
  - Test execution – possibly differentiating between unit and integration tests, possibly with test coverage analysis
  - Generation of documentation

By default, sbt works via conventions. We will stick to them as they are reasonable for common needs. An sbt project has the following structure:

- `my-project/`: root directory for the project
  - `build.sbt`: project build definition
  - `project/`: contains additional pieces of the build definition
  - `src/`: directory for project sources (Maven-style)
    - `main/`
      - `scala | java | resources /`
    - `test/`
      - `scala | java | resources /`
  - `target/`: generated files go here

Here's a simple version of the `build.sbt` file (note that it is actually Scala code):

```
name := "project-name"

organization := "it.unibo"

version := "1.0"

scalaVersion := "2.11.5"

resolvers += "Typesafe Repository" at "http://repo.typesafe.com/
    typesafe/releases/"
resolvers += "Sonatype snapshots" at "https://oss.sonatype.org/
    content/repositories/snapshots/"
```

```
libraryDependencies ++= Seq(
  "org.scalatest" % "scalatest_2.11" % "2.2.4" % "test"
)
```

sbt commands and tasks

Once you have a project, you can run sbt. There are two modes of operation: *interactive* (through a console) and *batch* (command-style). The following listing reports the key sbt **commands** and **tasks**:

```
;; BATCH MODE
$ sbt test    ;; Run all tests

;; INTERACTIVE MODE
$ sbt
;; Key commands
> help        ;; Shows help
> about       ;; Shows information about the project build
> tasks       ;; Lists the tasks defined for the current project
> projects    ;; Lists the available project
> project     ;; Displays or set the current project
> plugins     ;; Shows sbt plugins installed

;; Common tasks
> console     ;; Starts a Scala interpreter with project classes
              ;;   on the classpath.
> run         ;; Run a main class
> test        ;; Executes all tests
> testQuick   ;; Executes unrun, failed, or affected tests
> doc         ;; Generates API documentation
> clean       ;; Deletes files produced by the build
> compile     ;; Compiles sources
> package     ;; Produces the main artifact (e.g., JAR)
> publish     ;; Publishes artifacts to a repository (Ivy,Maven,..)
```

Integration with IDE

The best integration is when the IDE is able to recognize and use sbt under the hood. At the present time, this is the case of IntelliJ Idea, thanks to the `idea-sbt-plugin`[1].

Another approach (on the sbt-side) is to let sbt generate project files for the IDE of choice. This is supported through plugins[2][3] that can be enabled by adding the following lines in `my-project/project/plugins.sbt`

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "
    4.0.0")
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.4.0")
```

At this point, all you have to do is to generate the IDE-specific project files using sbt:

```
my-project/$ sbt eclipse
[info] About to create Eclipse project files for your project(s)

my-project/$ sbt gen-idea
```

At this point, you should be able to import the projects from your IDE of choice. For example, in Eclipse: `File -> Import -> Import existing project into workspace`.

4.2 Testing in Scala with ScalaTest and ScalaCheck

The references for this section are [11][12].

ScalaTest[4] is a testing framework for Scala. ScalaTest is attractive for three main reasons:

1. It supports *different testing styles*
2. It integrates with sbt, other testing/mocking tools, and IDEs (such as Eclipse and IntelliJ IDEA)
3. It provides many extension points for customizations

[1]`https://github.com/orfjackal/idea-sbt-plugin`
[2]For Eclipse: `https://github.com/typesafehub/sbteclipse`
[3]For IntelliJ Idea: `https://github.com/mpeltonen/sbt-idea`
[4]`http://www.scalatest.org`

Key elements

The key concepts and elements of ScalaTest can be summarised as follows:

- A **suite** is a collection of tests that can be run. It is represented by the `org.scalatest.Suite` trait
- A **test** is any named entity that can be run and either succeeds or fails (or is ignored)
- Suites can be nested
- Assertions are defined in the `Assertions` trait. Moreover, ScalaTest also provides a DSL for expressing assertions using the word ”should” (`Matchers` trait)
- You can associate *tags* with your tests and then use these markings to have a better control of the tests to include or exclude in executions
- A **reporter** (`Reporter` trait) is an object that collects the results from a running test suite and presents the collected information in some way to the user.
- Reporters receive information through *events* such as `TestStarting, TestFailed, SuiteCompleted, InfoProvided`, and the like (where, as the API documentation states, the terms *test* and *suite* are defined abstractly to enable a wide range of test and suite implementations).
- An **informer** (`Informer` trait) is an object that provides easy ways to send custom information to a reporter.

ScalaTest configuration in sbt

In order to use ScalaTest in your sbt project, you just have to add ScalaTest as a project dependency in your `build.sbt`:

```
libraryDependencies += "org.scalatest" % "scalatest_2.11" % "2.2.4"
     % "test"
```

Then, once you have your test suites within (by convention) `src/test/scala`, you can run the tests via the command `sbt test`.

4.2.1 Testing Styles

Now we'll have a quick look at some interesting testing styles supported by ScalaTest.

FunSuite

It represents a suite in which tests are represented as function values. Tests are defined by calling the `test(...)` method in the primary constructor of the suite. This style should be familiar to those coming from xUnit test frameworks. In addition, the `test` method accepts a string, which allows you to provide specification-like names to tests, resulting in a better communication with respect to method names `testXXX` typical in xUnit frameworks.

```
import org.scalatest.FunSuite

class MySuite extends FunSuite {
  test("A test") { assert(true); ... }

  test("Another test") { assertResult(10){5+5}; ...  }

  ignore("Test to be ignored for the moment") { ... }

  "This test" should "do something but isn't yet impl" in (pending)
}

@Ignore class SuiteToBeIgnored extends FunSuite { ... }
```

FlatSpec

This style trait is recommended for developers looking for a gentle transition from xUnit to **Behavior-Driven Development (BDD)**[5]. The structure is flat as in xUnit (hence the name), but tests are written in a *specification style* where each behavior to be tested has a specific *subject*.

```
class MySpec extends FlatSpec {
  behavior of "An empty set"

  it must "..." in { }

  "Others" can "..." in { }

  they should "..." in { }

  behavior of "Empty sets"

  they should "..." in { }
}

new MySpec execute
//   MySpec:
//     An empty set
//     - must ...
//     Others
//     - can ...
//     - should ...
//     Empty sets
//     - should ...
```

Note how this looks like a DSL. Moreover, in general (that is, not specifically to the testing style described in this section), you can also introduce your names to be used in place of `it` and `they`, which may be useful to facilitate code reading:

```
class MySpec extends FlatSpec {
  val MySubject1, MySubject2 = new ItWord

  MySubject1 should "..." in { ... }
```

[5]For more information: `http://behaviourdriven.org`

```
  MySubject2 must "..." in { ... }
}
```

FunSpec

This style consists of `describe` clauses giving information about the subject under test and test code introduced through the `it` method. The descriptions can be freely nested to incrementally refine the specification.

```
class MySpec extends FunSpec {
  describe("A subject") {
    describe("when condition A applies") {
      it("should perform task X as expected") { ... }
      it("should perform task Y as expected") { ... }
    }

    describe("when condition B applies") { ... }
  }
}
```

WordSpec

With this testing style you can structure your specification text using strings followed by `when`, `should`, `must`, `can`. When your sentences are ready to finish, you introduce a string followed by `in` and the test code block.

```
class MySpec extends WordSpec {
   "A subject" when {
     "in a situation" should {
       "perform task A as expected" in { ... }
       "perform task B as expected" in { ... }
     }
   }
}
```

FreeSpec

This testing style is not prescriptive with respect to how specification text should be written. For this reason, it is particularly suited to experienced BDD users. Specification clauses are introduced with the dash operator `-` and tests are specified with strings followed by `it` and a test code block.

```
class MySpec extends FreeSpec {
  "A subject" - {
    "when in some state A" - {
      "should perform task A as expected" in { ... }
      "should perform task B as expected" in { ... }
    }
  }
}
```

FeatureSpec

This testing style is primarily intended for acceptance or integration testing, where each test represents one *scenario* of a *feature*.

`FeatureSpec` is often used in conjunction with the `GivenWhenThen` trait, which defines the methods `given, when, then, and`. These methods take a string message and an implicit informer, and are used to improve test reporting and communication. In fact, for example, these features can be used – combined with conventions – as a way to make requirements explicit (possibly in a collaborative manner with business users) during sprints or iterations in agile settings (cf. Acceptance Test-Driven Development).

The following example shows how these traits can be used together with common conventions:

```
class MySpec extends FeatureSpec {
  info("AS A <stakeholder>")      // WHO
  info("I WANT <business goal>")  // WHAT
  info("SO THAT <value1>")        // WHY #1
  info("AND <value2>")            // WHY #2

  feature("Feature A") {
    scenario("<UserRole1> do <something> when <context1>") {
      Given("<description of context and assumptions>")
      // ...
      When("<description of what happens in the scenario>")
      // ...
      Then("<description of what we expect>")
      // ...
    }
    scenario("<UserRole2> do <something> when <context2>") { ... }
  }

  feature("Feature B") { ... }
}
```

On the choice of testing styles

The general recommendation is to choose one main testing style for unit testing and one main testing style for acceptance testing. Then, for the individual choices, it is a matter of experience, discipline, and desired level of prescriptiveness.

4.2.2 Property-based Testing

Property-based testing is a kind of testing where you express tests as high-level *properties* that should hold true. The key idea is the decoupling of the expected behavior specification from the *generation* of the test cases. The goal of a property-based testing library is to find a failing test case for a property (i.e., a *counterexample*). More precisely, we are interested in the "smallest" counterexample.

ScalaCheck[6] is a property-based testing library. It can be used autonomously or together with ScalaTest.

ScalaTest provides a property-based testing style suite trait `PropSpec` which allows you to introduce tests via calls to methods with name `property`:

```
class MyPropertyChecks extends PropSpec {
```

[6]http://www.scalacheck.org

```
  property("For <subject> should hold <propertyA>") { ... }

  property("For <subject> should hold <propertyB>") { ... }

}
```

Moreover, ScalaTest supports three flavors of property-based testing:

1. **ScalaTest-style** – may be choosed for consistency and clarity with respect to the other ScalaTest testing styles
   (a) Table-driven: test inputs are provided through tables
   (b) Generator-driven: test inputs are provided through generators
2. **ScalaCheck-style** – for use of native ScalaCheck features

Note that both the ScalaCheck-style and the generator-driven ScalaTest-style actually use ScalaCheck for checking properties and thus depend on it.

You can add a dependency to ScalaCheck in sbt by adding the following line in `build.sbt`

```
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.12.5"
    % "test"
```

Table-driven ScalaTest-style

To use this style, you mix the trait `TableDrivenPropertyChecks` into your suite. Typically you will also mix in `Matchers` to have access to BDD-like assertions. Trait `TableDrivenPropertyChecks` defines two key methods:

1. `forAll(table)((table) => unit)` – it allows to check properties against the rows of `table`; it is overloaded to work with each of the `TableForN` class (with N from 1 to 22)

2. `whenever(condition)(action)` – executes `action` when `condition` holds true

Note that according to this style, your properties are not predicates but actions consisting of assertions.

Here is an example:

```scala
import org.scalatest.{PropSpec, Matchers}
import org.scalatest.prop.TableDrivenPropertyChecks

class MySpecs extends PropSpec with
  TableDrivenPropertyChecks with Matchers {
  val lists = Table("list",                    // One column
                    List(1,2,3),
                    List("z","o"))
  val listPairs = Table( ("list 1", "list 2"), // Two columns
                         (Nil, List(3,1,6)),
                         (List("k","o"), List("z")))

  property("The reverse of reverse should be the list itself") {
    forAll(lists){ (l:List[_]) => l.reverse.reverse shouldEqual l }
  }

  property("List concatenation should increase length") {
    forAll(listPairs) { (l1:List[_], l2:List[_]) =>
      whenever(l1.length>=0) {
        (l1++l2).length should be > l2.length
      } // Note typo (> instead of >=). This is gonna fail
    }
  }
}
```

Generator-driven ScalaTest-style

To use generators, you have to mix in trait `GeneratorDrivenPropertyChecks` (or `PropertyChecks` if you want to use the table-driven as well).

```scala
class MySpecs extends PropSpec with
  GeneratorDrivenPropertyChecks with Matchers {
  property("Example of failing property on numbers") {
    forAll { (a: Int, b: Int) => (a+b) should be >= a }
  }
}

new MySpecs execute
// MySpecs:
// - Example of failing property on numbers *** FAILED ***
//   TestFailedException was thrown during property evaluation.
//     Message: -2 was not greater than or equal to -1
```

Again, note that properties are expressed through assertions, not via predicates. For what concerns generators, just note for now that, in some way, the framework knows how to feed `forAll` with input data.

ScalaCheck-style

In order to embrace this style, you will need to mix in trait `Checkers`, which provides many overloaded `check` methods that perform ScalaCheck property checks.

```
class MySpecs extends PropSpec with Checkers {
  property("List concatenation") {
    check { (a: List[Int], b: List[Int]) =>
      (a++b).length >= (a.length + b.length)
    }
  }

  property("List reverse") {
    check { (l: List[Char]) => l.reverse.reverse == l }
  }
}
```

Note how, this time, the properties are expressed as predicates.

On generators

You should have wondered how the previous examples work if we did not define any generator for our test case input data. Well, it turns out that ScalaCheck provides predefined generators for the most common data types. The interesting thing about generators is that new generators can be built out of existing generators using combinators such as `map`, `flatMap`, or `filter`. For example, we can build a new generator that produces a random character repeated 0 to 20 times, randomly, via:

```scala
import org.scalacheck.Gen

val repeatedCharGen = for(c <- Gen.choose('a','z');
                          n <- Gen.choose(0,20)
                        ) yield (1 to n) map (_ => c)

repeatedCharGen.sample  // Some(Vector(g,g))
repeatedCharGen.sample  // Some(Vector(e,e,e,e,e,e,e))
repeatedCharGen.sample  // Some(Vector(t,t,t,t,t,t,t,t,t,t,t,t))
repeatedCharGen.sample  // Some(Vector(x,x,x,x,x))
```

4.3 Performance Testing with ScalaMeter

The reference for this section is [13].

Performance is a design dimension that may be relevant in many software efforts. While we agree with Knuth's "Premature optimization is the root of all evil", it is also arguable that performance considerations cannot always be an afterthought in software design. Moreover, it is generally desirable to look for and remove significant bottlenecks, once we have a smell of them. Or we may want to be sure that no performance penalty is unexpectedly introduced by some seemingly harmless change.

Thus, in general we may be interested in

- measuring how much time a piece of code takes
- optimization of a serious bottleneck
- comparison of the performance of different algorithms
- performance regression tests for specific portions of the codebase

That is, we need some information about the performance of our code, and we need such information to be as much precise and reliable as possible.

This is particularly true for performance regression testing – otherwise, how can we determine a regression if our measurements are not reliable? One way to deal with non-deterministic results is by performing some statistical analysis.

However, when it comes to measuring performance, it is difficult to come up with accurate estimates. In fact:

- it is difficult to reproduce all the surrounding conditions that may affect performance
- the task itself of measuring does introduce some overhead
- the runtime behavior is generally non-deterministic
- it is typically unfeasible or unworthy to carry out a complete performance analysis, so you need to make trade-offs and choose a few key performance metrics

These issues justify the need for a tool such as ScalaMeter[7], whose goal is to help developers write performance tests that are more precise and reliable than a simple `endTime-startTime` log.

Installation

The runtime dependencies of ScalaMeter are JRE 7u4 (or higher) and Scala 2.10 (or higher).

To use ScalaMeter in your sbt project, you can add the following lines to `build.sbt`:

```
resolvers += "Sonatype OSS Snapshots" at
  "https://oss.sonatype.org/content/repositories/snapshots"
libraryDependencies +=
  "com.storm-enroute" %% "scalameter" % "0.8-SNAPSHOT" % "test"
testFrameworks += new TestFramework("org.scalameter.
    ScalaMeterFramework")
```

[7]http://scalameter.github.io

```
parallelExecution in Test := false
```

where the last line disables the parallel execution of tests that would otherwise invalidate your performance tests.

4.3.1 Microbenchmarking

`org.scalameter.api.Bench` is the abstract class that represents performance tests. It is a configurable test template. It is usually better to extend from one of its subclasses, that are predefined templates that fit the most common needs.

```
import org.scalameter.api._

object MyBenchmark extends Bench.LocalTime {

  // Test input
  val sizes = Gen.exponential("numNodes")(100,1000000,10)

  performance of "Range to list" in {
      using(sizes) in {
        case n => (1 to n).toList
      }
  }
}

MyBenchmark.executeTests

// ........... warmup .............
//  cores: 4
//  name: Java HotSpot(TM) 64-Bit Server VM
//  osArch: x86_64
//  osName: Mac OS X
//  version: 25.51-b03
//  Parameters(numNodes -> 100): 0.001621
//  Parameters(numNodes -> 1000): 0.014599
//  Parameters(numNodes -> 10000): 0.121857
//  Parameters(numNodes -> 100000): 0.82476
//  Parameters(numNodes -> 1000000): 8.060777
```

4.3.2 Performance regression testing

```
import org.scalameter.api._

object MyBenchmark extends Bench.Regression {
  // Configuration
  override val persistor = new SerializationPersistor()

  performance of "Range to list" in {
      using(Gen.unit("")) in {
        (_) => { }
      }
  }
}
```

Note that we need to specify a persistor. In fact, it is necessary to persist the results of test runs in order to compare new runs and determine if there is a regression or not.

Chapter 5

Summary and further directions

This technical report has provided a concise view of many conceptual and practical tools available to Scala programmers. In particular, we have covered:

- The basics of the Scala programming language
- Intermediate/advanced Scala features for library designers
- Common Scala patterns, idioms, and techniques
- The basics of some essential tools that should be used in Scala-based projects

Despite the richness in terms of information and insights, this is only the surface. It would be interesting to explore other directions as well, in particular:

- **Advanced functional programming in Scala**
  A good book on the subject is [14].
  `scalaz`[1] is a library that takes Scala's FP support to the next level

[1] `http://github.com/scalaz/scalaz`

- **Advanced generic programming in Scala**
  `shapeless`[2] is a type class and dependent type based generic programming library for Scala
- **Scala runtime reflection and compile-time reflection via macros** (from Scala 2.10)
- **Akka actor framework**

Moreover, a list of great Scala libraries and frameworks can be found at: `https://github.com/lauris/awesome-scala`.

[2]http://github.com/milessabin/shapeless

Bibliography

[1] Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41, October 2005.

[2] The scala language specification – version 2.9. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`. Accessed: 2015-11-29.

[3] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 441–469, London, UK, UK, 2002. Springer-Verlag.

[4] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM.

[5] The scala api documentation – version 2.11.0. `http://www.scala-lang.org/files/archive/api/2.11.0/`. Accessed: 2015-11-29.

[6] Mirko Viroli and Antonio Natali. Parametric polymorphism in java: An approach to translation based on reflective features. *SIGPLAN Not.*, 35(10):146–165, October 2000.

[7] Joshua D. Suereth. *Scala in Depth.* Manning Publications Co., Greenwich, CT, USA, 2012.

[8] Cay S. Horstmann. *Scala for the Impatient*. Addison-Wesley Professional, 1st edition, 2012.

[9] Chiari Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18:285–331, 2008.

[10] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[11] Scalatest user guide. `http://www.scalatest.org/user_guide`. Accessed: 2015-11-29.

[12] Scalatest api documentation. `http://doc.scalatest.org/2.2.4/index.html`. Accessed: 2015-11-29.

[13] Scalameter: Getting started guide. `http://scalameter.github.io/home/gettingstarted/0.7/`. Accessed: 2015-11-29.

[14] Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[15] Nilanjan Raychaudhuri. *Scala in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.