**Functional programming principles in Scala**

**Programming paradigms**

A paradigm describes distinct concept sor though patterns in some scientific discipline.

Main programming paradigms: Imperative programming, Functional programming and Logic programing, al of them Object-oriented programming.

***Imperative programing*** is about modifying mutable variables, using assignments and control structures such as if-then-else, loops, breack continue, return. How to adoid conceptualizing programs Word by Word?

***Functional programming*** (**FP**) means programming without mutable variables, assignments, loops and other imperative control structures.

In a wider sense, functional programming means focusing on the functions.

In particular, functions in a FP language are first-class citizens. This means:

-They can be defined anywhere, including inside other functions.

-Like any other value, they can be passed as parameters to functions and returned as results.

-As for other values, there exists a set operators to compose functions.

**Elements of programming**

Every non trivial programming language provides:

Primitive expressions representing the simplest elements, ways to combine expressions and ways to abstract expressions, which introduce a name for an expression by which it canthen be referred to.

A non-primitive expression is evaluated as follows:

1 Take the leftmost operator

2 Evaluate its operands (left before rigth)

3 Apply the operator to the operands

The evaluation process stops once it results in a value.

The substitution model

The idea of underlying this model is that all evaluation does is reduce an expression to a value. It can be applied to all expressions, as long as they have no side effects. The substitution model is formalized in λ-calculus, which gives a foundation for functional programming.

Then, does every expression reduce to a value (in a finite number of steps)?

No. Here is a counter-example:

Def loop: int = loop

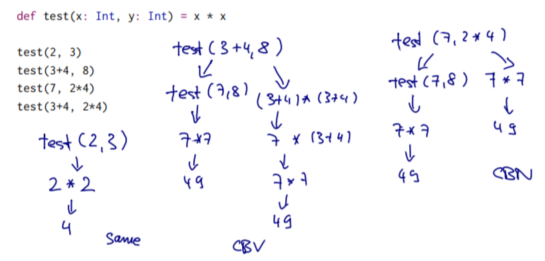
Evaluation strategy

Both evaluation strategies reduce to the same final values as long as the reduced expression consists of pure funcions, and both evaluations terminate.

***Call-by-value*** has the advantage that it evaluates every function argument only once.

***Call-by-name*** has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

Parameters by defauld are used as call-by-value, if you want it explicitly evaluated as call by name, you could use the notatio Parameter = ident:=> Type



**Blocks and visibility**

The definitions inside the block inside a block are only visible from within the block

The definitions inside a block shadow definitions of the same names ouside the block

**Evaluating a function application**

One evaluates a function application f(x1,…,xn) by replacing the application with the body of the function f, in which the actual parameters v1,…,vn replace the formal parameters of f.

This can be formalized as a rewriting of the program itself:

def f(x1, ..., xn) = B; ... f(v1, ..., vn)

def f(x1, ..., xn) = B; ... [v1/x1, ..., vn/xn] B

Here, [v1/x1, ..., vn/xn] B means: The expression B in which all occurrences of xi have been replaced by vi.

It is called a **subsitiution**.

Consider gcd the function that computes the greatest common divisor of two numbers. Here’s an implementation of gcd using Euclid’s algorithm.

**Tail recursion**

**def** gcd(a: Int, b: Int): Int =  
 **if** (b == 0) a **else** gcd(b, a % b)

gcd(14, 21) is evaluated as follows:  
→ gcd(14, 21)  
→ **if** (21 == 0) 14 **else** gcd(21, 14 % 21)  
→ **if** (**false**) 14 **else** gcd(21, 14 % 21)  
→ gcd(21, 14 % 21)  
→ gcd(21, 14)  
→ **if** (14 == 0) 21 **else** gcd(14, 21 % 14)  
→ gcd(14, 7)  
→ gcd(7, 0)

→ **if** (0 == 0) 7 **else** gcd(0, 7 % 0)

→ 7

Consider factorial:  
**def** factorial(n: Int): Int =  
 **if** (n == 0) 1 **else** n \* factorial(n - 1)

→ factorial(4)  
→ **if** (4 == 0) 1 **else** 4 \* factorial(4 - 1)  
→ 4 \* factorial(3)  
→ 4 \* (3 \* factorial(2))  
→ 4 \* (3 \* (2 \* factorial(1)))  
→ 4 \* (3 \* (2 \* (1 \* factorial(0)))  
 → 4 \* (3 \* (2 \* (1 \* 1)))  
 → 120

Implementation Consideration: If a function calls itself as its last action, the function’s stack frame can be reused. This is called tail recursion. ⇒ Tail recursive functions are iterative processes.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called tail-calls

Both tail recursión and tail calls reuse the stack frame in its last action, tail recursión cals itself and tail calls could be another function. The gdc func is an exampe of tail recursion whereas the factorial is neither a tail recursion nor a tail call.

Should every function be tail recursion? not in fact. But it could be a good idea when Deep recursive chains could happen to run in constant stack frame and avoid stack overflow exceptions. In the other hand, if your input data are not susceptible to Deep precausive chains then clarity trumps efficiency every time, in that cases don’t worry about tail recursion.

One can require that a function is tail-recursive using a @tailrec annotation:

**def** factorial(n: Int): Int = {  
 @tailrec  
 **def** iter(x: Int, result: Int): Int =  
 **if** (x ==0 ) result  
 **else** iter(x , result \* x)  
 iter(n,1 )  
}  
  
factorial(3) shouldBe 6  
factorial(4) shouldBe 24  
Higher Order Functions

Functional languages treat functions as first-class values. This means that, like any other value, a function can be passed as a parameter and returned as a result. This provides a flexible way to compose programs. Functions that take other functions as parameters or that return functions as results are called higher order functions.

Let’s define:  
**def** sum(f: Int => Int, a: Int, b: Int): Int =  
**if** (a > b) 0  
**else** f(a) + sum(f, a + 1, b)  
We can then write:  
**def** sumInts(a: Int, b: Int) = sum(id, a, b)  
**def** sumCubes(a: Int, b: Int) = sum(cube, a, b)  
**def** sumFactorials(a: Int, b: Int) = sum(fact, a, b)  
where  
**def** id(x: Int): Int = x  
**def** cube(x: Int): Int = x \* x \* x  
**def** fact(x: Int): Int =

In that case f is not a given function it is a parameter

**Function Types**

The type A => B is the type of a function that takes an argument of type A and returns a result of type B. So, Int => Int is the type of functions that map integers to integers.

Passing functions as parameters leads to the creation of many small functions.

Sometimes it is tedious to have to define (and name) these functions using def

Compare two strings: We do not need to define a string using def:

Instead of **def** str =**"abc"**; println(str)  
We can directly write  
*println*(**"abc"**)

Becausse strings exists as literals. Analogously we would like function literals, which led us write a function without giving it a name, these are called anonymous functions.

Example: A function that raises its argument to a cube: (x: Int) => x \* x \* x

Here, (x: Int) is the parameter of the function, and x \* x \* x is it’s body.

The type of the parameter can be omitted if it can be inferred by the compiler from the context.

If there are several parameters, they are separated by commas:

(x: Int, y: Int) => x + y

An anonymous function (x1 : T1, ..., xn : Tn) ⇒ E can always be expressed using def as follows: def f(x1 : T1, ..., xn : Tn) = E; f where f is an arbitrary, fresh name (that’s not yet used in the program).

▶ One can therefore say that anonymous functions are syntactic sugar, means that make writing functions sometimes easier,

Then the last examle is simplyfied by using anonymous functions

Let’s define:  
**def** sum(f: Int => Int, a: Int, b: Int): Int =  
**if** (a > b) 0  
**else** f(a) + sum(f, a + 1, b)  
We can then write:  
**def** sumInts(a: Int, b: Int) = sum(x=>x, a, b)  
**def** sumCubes(a: Int, b: Int) = sum(x=>x\*x\*x, a, b)  
**def** sumFactorials(a: Int, b: Int) = sum(x=> **if** (x == 0) 1 **else** fact(x - 1), a, b)

The avobe sum function uses liniar recursion. And the below one is writen using tail-recursion.

**def** sum(f: Int => Int, a: Int, b: Int): Int ={  
 **def** loop(a: Int, acc: Int): Int = {  
 *println*(**"a : "**+a)  
 *println*(**"acc : "**+acc)  
 **if** (a > b) acc  
 **else** loop(a+1, f(a) + acc)  
 }  
 loop(a,0)  
}

At this point we will write this parameters in a shorter way:

**def** sum(f: Int => Int): (Int, Int) => Int = {  
 **def** sumF(a: Int, b: Int): Int =  
 **if** (a > b) 0  
 **else** f(a) + sumF(a + 1, b)  
 sumF  
}

sum is now a function that returns another function. The returned function sumF applies the given function parameter f and sums the results. Stepwise Applications We can then define: **def** sumInts = sum(x => x)  
**def** sumCubes = sum(x => x \* x \* x)  
**def** sumFactorials = sum(fact)

These functions can in turn be applied like any other function:

sumCubes(1, 10) + sumFactorials(10, 20)

In the previous example we can avoid the sumInts, sumCubes, … middlemen as:

sum (cube)(1, 10)

Wich as follows will see another way to code this functionality by using currying functions instead of using the nested function sumF:

**Currying**

The definition of functions that return functions is so useful in functional programming that there is a special syntax for it in Scala called Currying.

**def** sum(f: Int => Int)(a: Int, b: Int): Int =  
 **if** (a > b) 0 **else** f(a) + sum(f)(a + 1, b)

That functions with multiple parameters are syntactic sugar

Then the type of the currying function is:

(Int => Int) => (Int, Int) => Int

Note that functional types associate to the right. That is to say that:

Int => Int => Int

c

Int => (Int => Int)

1. Write a product function that calculates the product of the values of a function for the points on a given interval.

2. Write factorial in terms of product.

3. Can you write a more general function, which generalizes both sum and product?

My versión

**def** general(f: (Int, Int)=> Int)(a: Int, b: Int): Int = {  
 f(a,b)  
}  
  
**def** prod(a:Int, b:Int):Int = {  
 **if**(a > b) 1 **else** a \* prod(a+1,b)  
}  
  
**def** sum(a:Int, b:Int):Int = {  
 **if**(a > b) 0 **else** a + sum(a+1,b)  
}  
  
  
**def** factorial(x: Int) = general(sum)(1,x)  
factorial(5)

Martin Odersky´s versión

**def** mapReduce(f: Int => Int, combine: (Int,Int)=> Int, zero: Int)(a: Int, b: Int): Int ={  
 **if** (a > b) zero  
 **else** combine(f(a), mapReduce(f, combine, zero)(a+1,b))  
}  
  
**def** product(f: Int=>Int)( a: Int, b: Int) = mapReduce(f, (x,y)=>x\*y, 1)(a,b)  
product(x=>x\*x)(2,3)  
  
**def** fact(x: Int) = product(x=>x)(1,x)  
fact(5)

**Functions and data**

A ***class*** is defined by a data structure encapsulation.

**class** Rational(x: Int, y: Int) {  
 **def** numer = x  
 **def** denom = y  
}

This definition introduces two entities:

A new ***type***, named Rational.

A constructor Rational to create elements of this type

Scala keeps the names of types and values in ***different namespaces*** identified by the hascode. So there’s no conflict between different defintions of Rational.

We call the elements of a class type ***objects***.

The object is created by prefixing an application of the constructor of the class with the operator new. **new** Rational(1, 2)

Objects of the class Rational have two ***members***, numer and denom.

**class** Rational(x: Int, y: Int) {  
 **def** numer = x  
 **def** denom = y  
 **def** add(r: Rational) =  
 **new** Rational(numer \* r.denom + r.numer \* denom,  
 denom \* r.denom)  
 **def** mul(r: Rational) = ...  
 ...  
 **override def** toString = numer + **“/”** + denom  
}

Remark: the modifier override declares that toString redefines a method that already exists (in the class java.lang.Object).

**private def** gcd(a: Int, b: Int): Int = **if** (b == 0) a **else** gcd(b, a % b)  
**val** numer = x / gcd(x, y)  
**val** denom = y / gcd(x, y)

gcd and g are private members; we can only access them from inside the Rational class.

This ability to choose different implementations of the data without affecting clients is called **data abstraction**.

Let’s say our Rational class requires that the denominator is positive. We can enforce this by calling the require function.

**class** Rational(x: Int, y: Int) {  
 *require*(y > 0, **"denominator must be positive"**)  
 ...  
}

require is a predefined function. It takes a condition and an optional message string. If the condition passed to require is false, an *IllegalArgumentException* is thrown with the given message string.Require is used to enforce a precondition on the caller of a function.

Besides require, there is also assert, which also takes a condition and an optional message string as parameters. **val** x = sqrt(y);assert(x >= 0)

Like require, a failing assert will also throw an exception, in this case AssertionError.

Assert is used as to check the code of the function itself.

In Scala, a class implicitly introduces a constructor. This one is called the ***primary constructor*** of the class. The primary constructor takes the parameters of the class and executes all statements in the class body.

Scala also allows the declaration of auxiliary constructors. These are methods named this

**class** Rational(x: Int, y: Int) {  
 **def this**(x: Int) = **this**(x, 1)  
 ...  
}

new Rational(2) > 2/1

"how is a type definition different from def ?"

"def" is the keyword used to define something that has a call by value resolution Scala, we basically use def whenever we want to declare a function. Whereas a type definition is syntactic sugar that replaces certain type name. With a type definition, you can set a new name to any type you were already using. By writing : **type** newInt = Int

You can now use newInt instead of Int when statically declaring type on your program **def** square(x: newInt): newInt = x\*x

This is a dumb example as you would have no reason to want to replace the name "Int" by something longer or less common such as "newInt".However having type definitions can be useful when you have more complicate types, like function types. " Int => Boolean

" is the type of a function which takes as only argument an Int, and returns a Boolean

**type** Set = Int => Boolean

So whenever you encounter a variable or parameter which has a type "Set", you now know it is in fact a function that can take a Int as parameter and return a Boolean.

**Class Hierarchies**

**Abstract Classes**

Consider the task of writing a class for sets of integers with the following operations.

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

def unión(other: Inset): InSet  
}

IntSet is an ***abstract class,*** abstract clases can contain members which are missing an implementation. Consequently, no instances of an abstract class can be created with the operator new.

It is also possible to redefine an existing, non-abstract definition in a subclass by using override

**Class extensions**

Let’s consider implementing sets as binary tres.

There are two types of possible tres: a tree for the empty set, and a tree consisting of an integer and two sub-trees.

Here are their implementations

**class** Empty **extends** IntSet {  
 **def** contains(x: Int): Boolean = **false  
 def** incl(x: Int): IntSet = **new** NonEmpty(x, **new** Empty, **new** Empty)  
}

**class** NonEmpty(elem: Int, left: IntSet, right: IntSet) **extends** IntSet {  
 **def** contains(x: Int): Boolean =  
 **if** (x < elem) left contains x  
 **else if** (x > elem) right contains x  
 **else true  
 def** incl(x: Int): IntSet =  
 **if** (x < elem) **new** NonEmpty(elem, left incl x, right)  
 **else if** (x > elem) **new** NonEmpty(elem, left, right incl x)  
 **else this**

**def** union(other: IntSet) {

if

}}

With this implementation of the NonEmpty clas having the following example with (5,7,9,10,13) if we include a 3, a new branch with (3,5,7) will be created. This is called ***persistent data sctuctures***, because even if we do changes, the old versión of the data is maintained, it is one of the conrer stones of scaling functional programming for collections and so on.

Empty and NonEmpty both extend the class IntSet. This implies that the types Empty and NonEmpty conform to the type IntSet.

An object of type Empty or NonEmpty can be used wherever an object of type IntSet is required.

IntSet is called the superclass of Empty and NonEmpty.

Empty and NonEmpty are subclasses of IntSet.

In Scala, any user-defined class extends another class. If no superclass is given, the standard class Object in the Java package java.lang is assumed.

The direct or indirect superclasses of a class C are called **base classes** of C. So, the base classes of NonEmpty are IntSet and Object.

In the IntSet example, one could argue that there is really only a single empty IntSet, so we can express it better with an object definition: **object** Empty **extends** IntSet, which defines a ***lñ*** named Empty

**How clases are organized**

Imports come in several forms: import week3.Rational // imports just Rational import week3.{Rational, Hello} // imports both Rational and Hello import week3.\_ // imports everything in package week3

Some entries are automatically imported in any Scala program: these are:

* All members of package scala
* All members of package java.lang
* All members of the singleton object scala.predef

[www.scala-lang.org/api](http://www.scala-lang.org/api)

In scala as same as in Java, a class can only have one superclass, this is called single inheritance. But what if a class has several natural supertypes to which it conforms or from it wants to inherit code?

Here you could use traits.

A trait is declared like an abstract class, just with trait intead of abstract class.

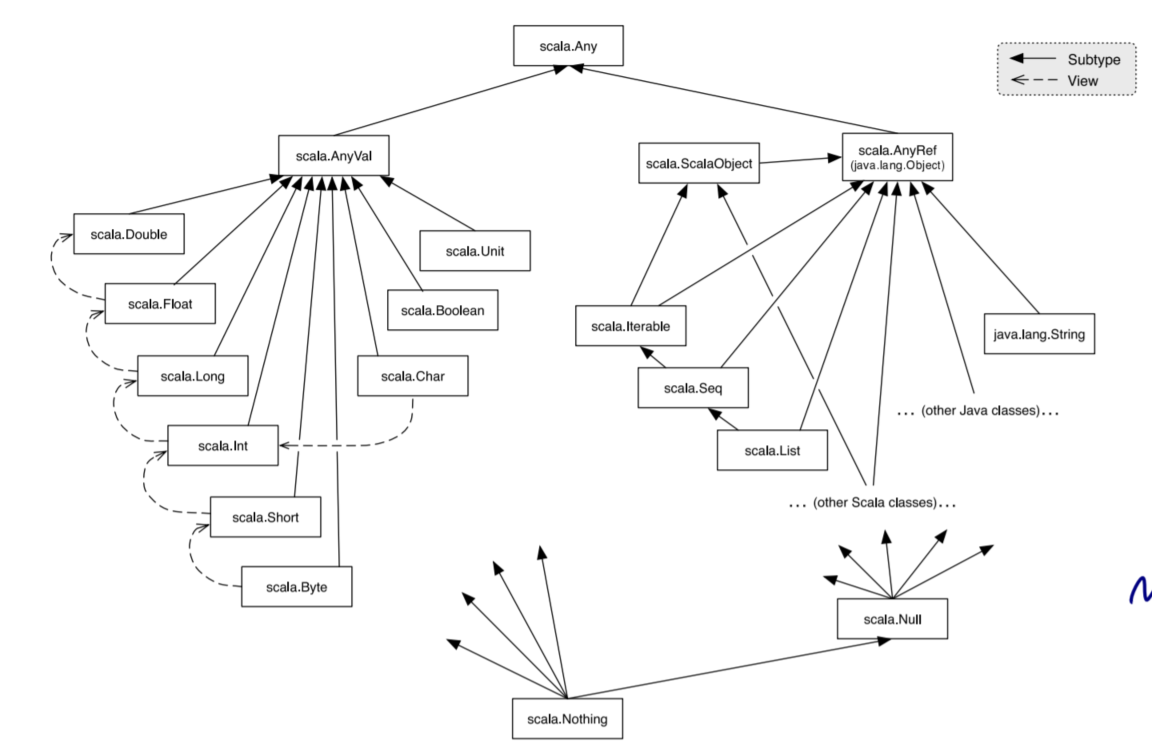
**trait** Planar {  
 **def** height: Int  
 **def** width: Int  
 **def** surface = height \* width  
}

Classes, objects and traits can inherit from at most one class but arbitrary many traits.

**class** Square **extends** Shape **with** Planar **with** Movable ...

Traits resemble interfaces in Java, but are more powerful because they can contains fields and concrete methods, whereas interfaces can just contain definitions.

On the other hand, traits cannot have parameters, only classes can.



At the top of the type hierarchy we find:

Any: the basetype of all types which implements (‘==’,’!=’,’equals’,’hasCode’,’toString’),

AnyRef is a subtype of Any, which also is called the reference types

and it is an alias for java.lang.Object, (the root class of all java clases) as for example java.lang.String and other Java classes, insead tha scala clases such as scala.Iterable, scala.Seq, scala.List and more, that also implements the trait scala.ScalaObject.

And AnyVal which is the base type of all value types which are just primitive types that Scala inherits from java.

The relation between the numeric types are not subtypes of each others, beacause a being a subtype means that you don´t have to re arrange the bits in order to go from one type to the other, whereas here definetly if woant to transformate between two numeric types, you

Then there are two types that are subtypes you do need to change your representation and the conversion could be not lost free 1.1 to 1.

Nothing is at the very bottom of all Scala types, it is a sub type of every other type and there is no value that has type nothing. It has to main usages:

* To signal abnormal termination, when an exception is thrown it returns nothing, **def** error(msg: String)= **throw new** Error(msg) *//error:Nothing*
* As an element type of empty collections (extending from scala.Null), and null is a subtype of all the classes that are reference types (classes that inherits from Object/AnyRef. So the type of null value is Null, and it is incompatible with subtypes of AnyVal

**if** (**true**) 1 **else** falseIf (**true**) 1 **else false** it returns AnyVal

**Type parametrization**

Cons-Lists

A fundamental data structure in many functional languages is the immutable linked list. It is constructed from two building blocks:

Here is an outline of a class hierarchy that represents lists of integers in this fashion:

**package** week4  
**trait** IntList ...  
**class** Cons(**val** head: Int, **val** tail: IntList) **extends** IntList ...  
**class** Nil **extends** IntList ...

A list then is either an empty list new Nil, or a list new Cons(x, xs) consisting of a head element and a tail list xs.

It seems too narrow to define only list with Int elements. We’d need another class hierarchy for Double lists, and so on, one for each possible element type. We can generalize the definition using a type parameter:

**package** week4  
**trait** List[T]  
**class** Cons[T](**val** head: T, **val** tail: List[T]) **extends** List[T]  
**class** Nil[T] **extends** List[T]

Type parameters are written in square brackets, e.g [T]

**trait** List[T] {  
 **def** isEmpty: Boolean  
 **def** head: T  
 **def** tail: List[T]  
}  
**class** Cons[T](**val** head: T, **val** tail: List[T]) **extends** List[T] {  
 **def** isEmpty = **false**}  
**class** Nil[T] **extends** List[T] {  
 **def** isEmpty = **true  
 def** head: Nothing = **throw new** NoSuchElementException(ŏNil.headŏ)  
 **def** tail: Nothing = **throw new** NoSuchElementException(ŏNil.tailŏ)  
}

**Generic functions**

Like classes, functions can have type parameters. For instance, here is a function that creates a list consisting of a single element.

**def** singleton[T](elem: T) = **new** Cons[T](elem, **new** Nil[T])

We can then write

singleton[Int](1)  
singleton[Boolean](**true**)

**Type inference**

In fact, the Scala compiler can usually dedue the correct type parameters from the value arguments of a function call.

So, in most cases, type parameters can be left out. You could also write:

singleton(1)  
singleton(**true**)

The compiler will infer the constant 1 in **type** Int.

Polimorphism

Polymorphism means that a function type comes “in many forms”.

In programming it means that the types can have instances of any types, or the function can be applied to arguments of many types.

We have seen two principal forms of polymorphism:

* **Subtyping**: instances of subclass can be passed to a base class ( if we have a parameter that accepts a list, we can pass either a Nilo r a Cons)
* **Generics**: Instances of a function or class that are created by type parametrization

The two forms of polymorrphism complement each other well and Scala has both of them, typically you would say that subtyping was traditionally a form that object-oriented languages had first, whereas generics was a form that functional languages had first.

Exercise: Write a function that takes an integer n and a list and selects the n’th element of the list. Elements are numbered from 0. If index is outside the range from 0 up the lenght of the list minus one, a IndexOutBoundsException should be thrown. ,ñ

**def** getNthElement[T](n: Int, l:List[T]): T = {  
 **if** (l.isEmpty) **throw new** IndexOutOfBoundsException  
 **if**(n==0) l.head  
 **else** getNthElement(n - 1, l.tail)  
}

**Wildcard patterns**

The wildcard pattern (\_) matches any object whatsovever. You have already seen it used as a default catch-all alternative, ike this:

expr match {

  case BinOp(op, left, right) =>

    println(expr + " is a binary operation")

  case \_ => // handle the default case

}

Wildcardds can also be used to ignore parts of an object that you do not care about. For example, the previous example does ot actually care what the elements of a binary operation are; it just checs whether or not it is a binary operation. Thus, the code can just as well use the wildcard pattern ofr the elements of the BinOp:

expr match {

  case BinOp(\_, \_, \_) => println(expr + " is a binary operation")

  case \_ => println("It's something else")

}

**Typed patterns**

You can use typed pattern as a convinient replacement for type tests and type casts.

def generalSize(x: Any) = x match {

  case s: String => s.length

  case m: Map[\_, \_] => m.size

  case \_ => -1

}

The pattern "s: String" is a typed pattern; it matches every (non-null) instance of String. The pattern variable s then refers to that string.

**Sealed classes**

Whenever you write a pattern match, you need to make sure you havecovered all of the possible cases. Sometimes you can do this byadding a default case at the end of the match, but that only appliesif there is a sensible default behavior. What do you do if there isno default? How can you ever feel safe that you covered all thecases?

You can enlist the help of the Scala compiler in detectingmissing combinations of patterns in a match expression. To do this, the compiler needs to be able to tell which are the possiblecases. In general, this is impossible in Scala because new caseclasses can be defined at any time and in arbitrary compilation units.For instance, nothing would prevent you from adding a fifth case classto the Expr class hierarchy in a different compilation unit fromthe one where the other four cases are defined.

A sealed class cannot have any new subclasses added exceptthe ones in the same file. This is very useful for pattern matching because it means you only need to worry about the subclasses youalready know about. What's more, you get better compiler support aswell. If you match against case classes that inherit from a sealedclass, the compiler will flag missing combinations of patterns with awarning message.

If you write a hierarchy of classes intended to be patternmatched, you should consider sealing them. Simply put the sealedkeyword in front of the class at the top of the hierarchy.Programmers using your class hierarchy will then feel confident inpattern matching against it. The sealed keyword, therefore, isoften a license to pattern match. Listing 15.16shows an example in which Expr is turned into a sealed class.

  sealed abstract class Expr

  case class Var(name: String) extends Expr

  case class Number(num: Double) extends Expr

  case class UnOp(operator: String, arg: Expr) extends Expr

  case class BinOp(operator: String,

      left: Expr, right: Expr) extends Expr

Now define a pattern match where some of the possible cases are left out:

def describe(e: Expr): String = e match {

  case Number(\_) => "a number"

  case Var(\_)    => "a variable"

}

You will get a compiler warning like the following:

warning: match is not exhaustive!

missing combination           UnOp

missing combination          BinOp

### SELECTING MUTABLE VERSUS IMMUTABLE COLLECTIONS

For some problems, mutable collections work better, while for others,immutable collections work better. When in doubt, it is better to start with an immutable collection andchange it later, if you need to, because immutable collections can be easier to reason about than mutable ones.

Besides being potentially easier to reason about, immutablecollections can usually be stored more compactly than mutable ones ifthe number of elements stored in the collection is small. For instancean empty mutable map in its default representation of HashMap takes upabout 80 bytes, and about 16 more are added for each entry that's addedto it. An empty immutable Map is a single object that's sharedbetween all references, so referring to it essentially costs just asingle pointer field.

Pure Object Orientaton

A pure object oriented language is one in which every value is an object. If the language is base don classes, this means that the type of each value is a class.

At first glance, there seem to be some exceptions: Primitive types, functions, conceptualy types such as Int or Boolean do not receive special treatment in Scala. They are like other classes, defied in the package scala.

For reasons of efficiency, the Scala compiler represents the values of type sacala.Int by 32-bit integers, and the values of type scala.Boolean by Java’sBooleans, etc.

The Boolean type maps to the JVM’s primitive type boolean.

But one could define it as a class from first principles

**package** idealized.scala  
**abstract class** Boolean {  
 **def** ifThenElse[T](t: => T, e: => T): T  
  
 **def** && (x: => Boolean): Boolean = ifThenElse(x, **false**)  
 **def** || (x: => Boolean): Boolean = ifThenElse(**true**, x)  
 **def** unary\_!(): Boolean = ifThenElse(**false**, **true**)  
  
 **def** == (x: Boolean): Boolean = ifThenElse(x, x.unary\_!)  
 **def** != (x: Boolean): Boolean = ifThenElse(x.unary\_!(), x)  
   
 **def** < (x: Boolean) = ifThenElse(**false**, x)  
 ...  
}  
  
**object true extends** Boolean {  
 **def** ifThenElse[T](t: => T, e: => T): t  
}  
  
**object false extends** Boolean {  
 **def** ifThenElse[T](t: => T, e: => T) = e  
}

**abstract class** Nat {  
 **def** isZero: Boolean  
 **def** predecessor: Nat  
 **def** successor = **new** Succ(**this**)  
 **def** +(that: Nat): Nat  
 **def**-(that: Nat): Nat  
}  
  
**class** Zero **extends** Nat {  
 **def** isZero = **true  
 def** predecessor = **throw new** Error(**"0.predecessor"**)  
 **def** +(that: Nat) = that  
 **def** -(that: Nat) = **if** (that.isZero) **this else throw new** Error(**"negative number"**)  
}  
  
**class** Succ(n: Nat) **extends** Nat {  
 **def** isZero = **false  
 def** predecessor = n  
 **def** +(that: Nat) = **new** Succ(n + that)  
 **def** -(that: Nat) = **if** (that.isZero) **this else** n - that.predecessor  
}

We have seen that Scala’s numeric types and the Boolean type can be implemented lie normal classes.

But what about functions?

In fact function values are treated as objects in Scala.

The function type A => B is justa n abbreviation for the class scala.Function1[A, B], which is roughly defined as follows.

**package** scala  
**trait** Function1[A, B] {  
 **def** apply(x: A): B  
}

So functions are objects with apply methods.

There are also traits Function2, Function3, … for functions which take more parameters (currently up to 22)

An anonymous function such as

(x: Int) => x \* x {

Is expanded to:

{ **class** AnonFun **extends** Function1[Int, Int] {  
 **def** apply(x: Int) = x \* x  
 }  
  
 **new** AnonFun  
 }

Or, shorter, using anonymous class syntax:

**new** Funtion1[Int, Int] {  
**def** apply(x: Int) = x \* x  
}

A function call, such as f(a, b), where f is a value of some class type, is expanded to

f.apply(a,b)

So the OO-translation of

**val** f = (x: Inte) => x \* x

Would be

**val** f 0 **new** Function1[Int, Int]{  
**def** apply(x: Int) = x \* x  
}  
f.apply(7)

Note that a method such as

**def** f(x: Int): Boolean = ...

is not itself a function value.

But if f is used in a place where a function type is expected, it is converted automatically to the function value (x: Int) => f(x)

Or expanded (called eta-expansion):

**new** Funtion1[Int, Boolean] {  
**def** apply(x: Int) = f(x)  
}

Polymorphism

Two principal forms of polymorphism, subtyping and generics

At this section we will look at their interactions, bounds and variance.

Consider the method assertAllPos which takes and IntSet, returns the IntSet itself if all elements are positive and throws an exception otherwise:

**def** assertAllPos(s: IntSet): IntSet  
  
**def** assertAllPos[S <: IntSet](r: S): S = ...

Here “<: IntSet” is an upper bound of the type parameter S, it means that S can be instantiated only to types that conform to IntSet.

Generally, the notation

S <: T means: S is a subtype of T, and

On the other hand, you can also use lower bound. [S >: NonEmpty] introduces a type parameter S that can range only over suprtypes of NonEmpty. So S could be one of NonEmpty, IntSet, AnyRef, or Any.

We will see later on in this sesión where lower bounds are useful.

S >: T means: S is a super type of T, ori s a subtype of S

Finally, it is also passible to mix a lower bound with an upper bound.

For instance: [S >: NonEmpty >: IntSet] in which restricts S any type on the interval between NonEmpty and IntSet.

Covariance

There’s another interaction between subtyping and type parameters we need to consider. Given: NonEmpty <: IntSet is List[NonEmpty] <: List[IntSet]

Intuitively, this makes sense: A list of non-empty sets sis a special case of a list of arbitrary sets.

We call types for which this relationship holds covariant because their subtyping relationship varies with the type parameter.

Arrays in Java are covariant, so one would have: NonEmpty[] <: IntSet[] and it cand cause problems:

Nonempty[] a = new NonEmpty[]

IntSet[] b = a

b[0] = Empty

NonEmpty s = a[0]

To avoid this, the Liskov Subsitution Principle talls us whn a type can be a subtype of another:

If A<:B, then everything one can to do with a value of type B one should also be able to do with a value of type A.

Some types should be covariant whereas others should not. Roghly speaking, a type that accepts mutations of its elements should not be covariant. But immutable types can be covariant, if some conditions on methods are met.

Say C[T] is a paramerized type and A, B are types such that A <: B.

In general, there are three possible relationships between C[A] and C[B]:

C[A] <: C[B] : C is covariant

C[A] >: C[B] : C is contravaiant

Neither C[A] nor C[B] is subtype of the other : C is nonvariant

<https://www.coursera.org/learn/progfun1/lecture/dnreZ/lecture-4-4-variance-optional>

**Lists are recursive while arrays are flat!**

**Operators ending in : associate to the right**

**A::B::C is interpretated as A:: (B :: C)**

**val** nums = 1 :: 2 :: 3 :: 4 :: Nil  
  
 **val** nums = 1 :: (2 :: (3 :: (4 :: Nil)))  
   
 Nil.::4.::3.::2.::1