

# Functional Programming

## Functional Programming

Week1 :

call by value vs call by name :

Conditionals :

Blocks , Scopes and first functional program :

Tail recursion :

Week 2 :

Currying :

Classes :

Substitution and Extensions :

1. Substitution :

2. Extension methods (1) :

3. Operators :

Week3 :

Abstract classes :

Object definition :

Programs :

Traits :

Cons List :

A complete definition with **Generics** :

Generic functions :

Pure Object Orientation :

Week 4 :

Decomposition :

Solution : Pattern Matching :

Lists and more pattern matching :

Enum :

Type Bounds :

Variance :

Week 5 :

List methods :

Simple merge sort implementation :

Filtering and mapping :

Reductions :

Structural Induction :

Week 6 :

Call by name

Arrays and String :

Sequences :

Combinatorial Search and For-Expressions :

Maps :

Week 7 :

Queries with For :

How are FOR expressions translated ?

Generalizing for expressions :

Example : random value generator :

Monads :

Monad Laws :

Exceptions :

Week 9 :

Structural induction on Trees :

Lazy Lists :

Infinite sequences :

Week 10 :

Contextual Abstraction :

Using clauses :

Context bound :

Given instances :

Importing Given instances :

Type classes :

Conditional instances :

Abstract algebra and type classes :

Important remarks :

Implicit Function types :

## **Week1 :**

### **call by value vs call by name :**

**Call-by-value** : evaluate the parameters *then* execute the function on these values.

e.g :

```

sumOfSquares(3, 2+2)
sumOfSquares(3, 4) // evaluated 2+2 BEFORE executing sumOfSquares
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25

```

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

**Call-by-name :** Apply the function to unreduced arguments.

e.g :

```

sumOfSquares(3, 2+2)
square(3) + square(2+2) // executing sumOfSquare before evaluating 2+2
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4

```

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.

e.g :

```

def loop : Int = loop
def f(a:Int , b : Int ) : Int = a // Call-by-value
f(1,loop) // does not terminate
def f(a:Int, b : => Int) : Int = a // Call-by-Name ( => )
f(1,loop) // terminates

```

Both strategies reduce to the same final values as long as

- ▶ the reduced expression consists of pure functions, and
- ▶ both evaluations terminate.

## Conditionals :

```
def abs(x: Int) = if x ≥ 0 then x else -x
```

`if-then-else` in Scala is an expression (has a value) not a statement .

## Blocks , Scopes and first functional program :

Sqrt with Newton's method :

```
def sqrt(x: Double) = {  
  def sqrtIter(guess: Double, x: Double): Double =  
    if isGoodEnough(guess, x) then guess  
    else sqrtIter(improve(guess, x), x)  
  def improve(guess: Double, x: Double) = (guess + x / guess) / 2  
  def isGoodEnough(guess: Double, x: Double) = abs(square(guess) - x) < 0.001  
  
  sqrtIter(1.0, x)  
}
```

`sqrtIter` , `improve` and `isGoodEnough` should not be outside `sqrt` because :

- they pollute the global scope.
- they should not be accessible to the user of the `sqrt` code given they are specific to `sqrt` .

the `{}` could be removed for more clarity.

## Tail recursion :

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

```
gcd(14, 21)  
→ if 21 == 0 then 14 else gcd(21, 14 % 21)  
→ if false then 14 else gcd(21, 14 % 21)  
→ gcd(21, 14 % 21)  
→ gcd(21, 14)  
→ if 14 == 0 then 21 else gcd(14, 21 % 14)  
→ gcd(14, 7)
```

`gcd(14,21) = gcd(21,14) = gcd(14,7) = gcd(..., ...) = ... = 7` : flat structure, `gcd` only calls `gcd` => **tail recursion**.

```
def factorial(n: Int): Int = if n == 0 then 1 else n * factorial(n - 1)
```

```
factorial(4)  
→ if 4 == 0 then 1 else 4 * factorial(4 - 1) 3-> → 4 * factorial(3)  
→ 4 * (3 * factorial(2))  
→ 4 * (3 * (2 * factorial(1)))  
→ 4 * (3 * (2 * (1 * factorial(0))))  
→ 4 * (3 * (2 * (1 * 1)))
```

`factorial` calls `factorial` *and multiplies* , nested structure => **not tail recursion**

To tell the compiler than a function is tail recursive:

```
import scala.annotation.tailrec
@tailrec
def gcd(a: Int, b: Int): Int = ...
```

## Week 2 :

### Higher order functions :

*Def*: Functions that take other functions as parameters and return functions.

example :

```
def sum( f : Int ⇒ Int )( a : Int , b : Int ) : Int
```

- takes as argument : the function `f : Int ⇒ Int`.
- returns : a function that has 2 arguments `a: Int, b: Int` and returns `Int`

**Function types** : `A ⇒ B` is the type of a function that takes an argument of type A and returns a result of type B. e.g : `Int ⇒ Int` maps integers to integers.

### Anonymous Functions :

```
(x: Int, y: Int) ⇒ x + y
```

Use case :

```
def sumCubes(a: Int, b: Int) = sum(x ⇒ x * x * x, a, b) // type of function is
inferred by the compiler
```

### Currying :

Suppose we have this header for `sum` : `def sum(f : Int ⇒ Int , a : Int , b : Int)`

```
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(fact, a, b)
```

Notice that a and b get passed unchanged from `sumInts` to `sum` .

Q : Can we do it shorter ?

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

def sumInts = sum(x => x)
def sumCubes = sum(x => x * x * x)
def sumFactorials = sum(fact)
```

`sum` is now a function that takes 1 argument (`f`) and returns a function that takes 2 arguments `a` and `b`.

We can still do shorter :

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

Q: What is the type of `sum` ?

A: `(Int=>Int) => ((Int,Int)=>Int)` equivalent to

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

Note that function types **associate to the right** so it is equivalent to :

`Int => Int => Int` is equivalent to `Int => (Int => Int)`

## Classes :

we can define classes in Scala :

```
class Rational(x: Int, y: Int):
  def numer = x // number and denom here are functions (recalculated each call)
  def denom = y

  // adding operations
  def addRational(r: Rational, s: Rational):
    Rational = Rational(r.numer * s.denom + s.numer * r.denom, r.denom *
s.denom)
  // overriding toString
  override def toString = s"$numer/$denom" // s means formatted string
  // what is after a $ is evaluated

val x = Rational(1, 2) // x: Rational = Rational@2abe0e27
x.numer // 1
x.denom // 2
val y = Rational(5, 7)
val z = Rational(3, 2)
x.add(y).add(z)
```

Suppose that `numer` and `denom` need more calculations , like the following :

```
class Rational(x: Int, y: Int):  
  private def gcd(a: Int, b: Int): Int =  
    if b == 0 then a else gcd(b, a % b)  
  def numer = x / gcd(x, y) // these are recalculated at each call  
  def denom = y / gcd(x, y) // BAD IDEA  
  
  // BETTER IDEA : IMMUTABLE VARIABLES :  
  val numer = x / gcd(x, y)  
  val denom = y / gcd(x, y)
```

It is possible to use `this(...)` like in Java.

## Substitution and Extensions :

### 1. Substitution :

Suppose that we have a class definition :

```
class C(x1, ..., xm){ ... def f(y1, ..., yn) = b ... }
```

Q : How is the following expression evaluated ? `C(v1, ..., vm).f(w1, ..., wn)`

A : Three substitutions happen written :

```
[w1/y1, ..., wn/yn][v1/x1, ..., vm/xm][C(v1, ..., vm)/this] b
```

- Substitution of `y1, ..., yn` of `f` by the arguments `w1, ..., wn` .
- Substitution of the `x1, ..., xm` of the class `C` by the class arguments `v1, ..., vm` .
- Substitution of the self reference `this` by the value of the object `C(v1, ..., vn)` .

### 2. Extension methods (1) :

Having to define all methods that belong to a class inside the class itself can lead to very large classes, and is not very modular.

Methods that do not need to access the internals of a class can alternatively be defined as extension methods.

For instance, we can add `min` and `abs` methods to class `Rational` like this:

```
extension (r: Rational):  
  def min(s: Rational): Boolean = if s.less(r) then s else r  
  def abs: Rational = Rational(r.numer.abs, r.denom)
```

- Extension methods **CANNOT** access internals ( `private` ) or `this(...)` .

### 3. Operators :

- We write `x + y` , if x and y are integers, but
- We write `r.add(s)` if r and s are rational numbers.

In Scala, we can eliminate this difference with **operators**.

```
extension (x: Rational):  
  def + (y: Rational): Rational = x.add(y)
```

### Precedence rules :

```
(all letters)  
|  
^  
&  
< >  
= !  
:  
+ -  
* / %  
(all other special characters)
```

(all other special characters) having the **highest priority**.

## Week3 :

### Abstract classes :

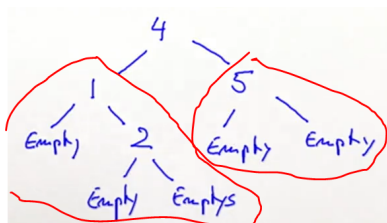
```
abstract class IntSet:  
  def incl(x: Int): IntSet // ABSTRACT MEMBERS : no implementation provided  
  def contains(x: Int): Boolean  
  // incl returns the union of {x} and this set
```

Like in Java, abstract classes cannot be instantiated.

We will try to implement a set as a binary tree.

A set can either be :

1. A tree for the **empty set**.
2. A tree consisting of **one** integer with two subtrees.



```
class Empty() extends IntSet:  
  def contains(x: Int): Boolean = false  
  def incl(x: Int): IntSet = NonEmpty(x, Empty(), Empty())
```



```

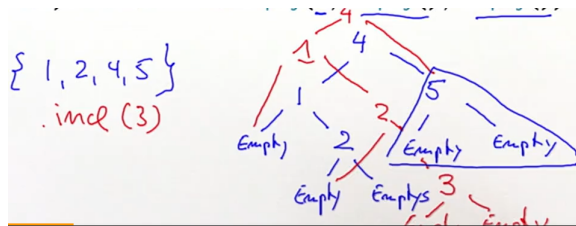
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet:
  def contains(x: Int): Boolean =
    if x < elem then left.contains(x)
    else if x > elem then right.contains(x) else true

  def incl(x: Int): IntSet =
    if x < elem then NonEmpty(elem, left.incl(x), right)
    else if x > elem then NonEmpty(elem, left, right.incl(x))
  else this
end NonEmpty

```

## 1. Persistence :

Note that we the sets are immutable, we always return a new tree while **reusing some subtrees**.(blue one in the following e.g)



A data structure that is created by maintaining the old one is called **persistent**.

## 2. Dynamic binding :

```

class Empty() extends IntSet:
  ...
  def union(other : IntSet): IntSet = other
class NonEmpty((elem: Int, left: IntSet, right: IntSet)) extends IntSet :
  ...
  def union(other : IntSet) : IntSet =
    left.union(right).union(that).incl(elem)

// Why does this terminate ? Union is called with strictly smaller sets each
// time so union( "an empty set " ) will be called at some point.

```

Note that a call to `union` doesn't execute the same function if `s` is `Empty` or `NonEmpty` . That is called Dynamic **binding**. ( Polymorphism )

## Object definition :

In the `IntSet` example, one could argue that there is really **only a single** empty `IntSet` .

This defines a **singleton** object named `Empty`. No other `Empty` instance can be (or needs to be) created.

```

object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty) end Empty

```

## Companion object :

If a class and object with the same name are given in the same sourcefile, we call them companions. Example:

```
class IntSet ...
object IntSet:
  def singleton(x: Int) = NonEmpty(x, Empty, Empty)
```

Similar to Java, static nested class.

How is naming a class and an object the same name possible ? Scala has **two** namespaces , one for objects and the other for classes.

## Programs :

Like Java , scale source files can have a main methods

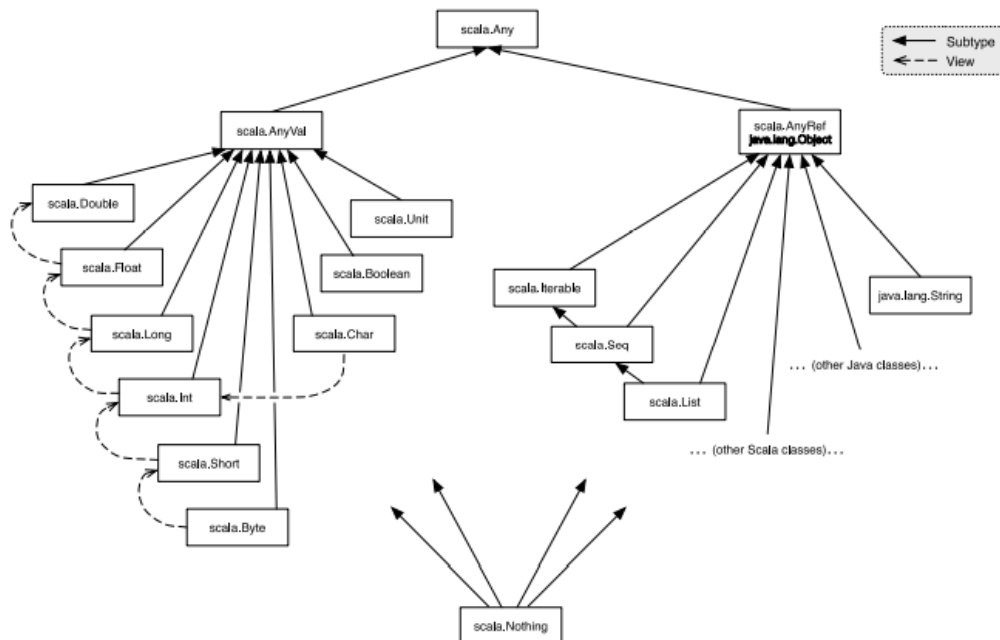
```
object Hello:
  def main(args: Array[String]): Unit = println("hello world!")
// written Shorter as the following :
@main def birthday(name: String, age: Int) =
  println(s"Happy birthday, $name! $age years old already!")
```

to run : `scala Hello`

## Traits :

Java `Interface` but stronger :they can have parameters and can contain fields and concrete methods.

```
trait Planar:
  def height: Int
  def width: Int
  def surface = height * width
class Square extends Shape, Planar, Movable ...
```



- - -> : can be converted (viewed) as

## Cons List :

A fundamental data structure in many functional languages is the **immutable linked list**.

It is constructed from two building blocks:

1. `Nil` :the empty list
2. `Cons` :a cell containing an element and the remainder of the list

```

trait IntList ...
class Cons(val head: Int, val tail: IntList) extends IntList ...
// NOTE THE USE OF val
// this defines at the same time a parameter and a field of a class
class Nil() extends IntList ...

```

A `IntList` is either a `Nil()` or a `Cons(x, xs)` .

## A complete definition with Generics :

```

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 = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")

```

## Generic functions :

```
def singleton[T](elem: T) = Cons[T](elem, Nil[T])
// We can then write:
singleton[Int](1)
singleton[Boolean](true)
```

## Pure Object Orientation :

*Def*: A pure object-oriented language is one in which every value is an object.

*Q*: Is Scala a pure OO language ?

At first primitive types and functions seem like exceptions.

**Primitive types** are in fact not implemented as Class. (E.g `scala.Int` : 32bits , `scala.Boolean` as Java Boolean ) , for reasons of efficiency.

But they are not *conceptually* treated differently. In fact one can implement them using classes :

```
package idealized.scala
abstract class Boolean extends AnyVal:
  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)
  ...
end Boolean
object true extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = t
object false extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = e
```

[Here](#) is how to implement `Scala.Int` .

**Functions** : In scala functions are `objects` with `apply` methods

The function type `A => B` is just an abbreviation for the class

`scala.Function1[A, B]` , which is defined as follows.

```
package scala
trait Function1[A, B]:
  def apply(x: A): B
```

Example :

```
f = (x: Int) => x * x
// is expended to
f = new Function1[Int, Int]:
  def apply(x: Int): Int = x * x
```

This anonymous class can itself expend to

```
{ class $anonfun() extends Function1[Int, Int]:  
  def apply(x: Int) = x * x  
  $anonfun()  
}  
f(7) // expends to  
f.apply(7)
```

## Week 4 :

### Decomposition :

Expr

Suppose you want to write a small interpreter for arithmetic

/ \

expressions.

Number

Sum

Sum(Number(1) , Number(2)).eval = 3

```
trait Expr :  
  def eval(e: Expr): Int =  
    if e.isNumber then e.numValue  
    else if e.isSum then eval(e.leftOp) + eval(e.rightOp)  
    else throw Error("Unknown expression " + e)
```

1. `isNumber` , `isSum` gets quickly gets tedious when adding other datatypes .
2. There's no static guarantee you use the right accessor functions.  
You might hit an Error case if you are not careful.

### Non solution :Type Tests and Type casts

USE `def isInstanceOf[T]: Boolean` to type *test* and `def asInstanceOf[T]: T` to cast.

Their use in Scala is discouraged, because there are better alternatives

### Solution 1 : Object Oriented Decomposition

```
trait Expr:  
  def eval: Int  
  class Number(n: Int) extends Expr:  
    def eval: Int = n  
  class Sum(e1: Expr, e2: Expr) extends Expr:  
    def eval: Int = e1.eval + e2.eval
```

- ▶ OO decomposition mixes data with operations on the data.
- ▶ Good for encapsulation and data abstraction.
- ▶ Dependencies between classes => Increases **Complexity**.
- ▶ It makes it easy to add new kinds of data but **hard** to add new kinds of operations

OO decomposition only works well if operations are on a single object.

NOT for expressions of type `a * b + a * c -> a * (b + c)`

### Solution : Pattern Matching :

```
def eval(e: Expr): Int = e match
  case Number(n) => n // pattern => expression
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

A `MatchError` exception is thrown if no pattern matches the value of the selector.

To be able to do pattern matching `Number` and `Sum` should be **case classes**.

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

### Lists and more pattern matching :

► the empty list `Nil`, and

► the construction operation `::` (pronounced cons):

`x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`

```
val fruit: List[String] = List("apples", "oranges", "pears")
fruit = "apples" :: ("oranges" :: ("pears" :: Nil)) // similar
```

**Right associativity convention** : Operators ending in “:” associate to the right.

`A :: B :: C` is interpreted as `A :: (B :: C)`.

All operations on lists can be expressed in terms of the following three:

- `head` the first element of the list
- `tail` the list composed of all the elements except the first.
- `isEmpty` true if the list is empty, false otherwise .

example of using lists and pattern matching :

```
def isort(xs: List[Int]): List[Int] = xs match
  case List() => List()
  case y :: ys => insert(y, isort(ys))

def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => List(x)
  case y :: ys => if x > y then y :: insert(x,ys) else x :: xs
```

## Enum :

```
Here's our case class hierarchy for expressions again:  
trait Expr  
object Expr:  
  case class Var(s: String) extends Expr  
  case class Number(n: Int) extends Expr  
  case class Sum(e1: Expr, e2: Expr) extends Expr
```

This is so common in scala that there's a shorthand for that.

```
enum Expr:  
  case Var(s: String)  
  case Number(n: Int)  
  case Sum(e1: Expr, e2: Expr)  
  case Prod(e1: Expr, e2: Expr)
```

There's more to Enumeration, they can take parameters and can define methods.

```
enum Direction(val dx: Int, val dy: Int):  
  case Right extends Direction(1, 0)  
  case Up extends Direction(0, 1)  
  case Left extends Direction(-1, 0)  
  case Down extends Direction(0, -1)  
  def leftTurn = Direction.values((ordinal + 1) % 4)  
end Direction  
  
val r = Direction.Right  
val u = r.leftTurn // u = Up  
val v = (u.dx, u.dy) // v = (1, 0)
```

## Type Bounds :

`assertPos` should return the set itself if all elements are positive and throw  
`exception` otherwise.

```
def assertAllPos(s: IntSet): IntSet
```

The above definition doesn't show that `assertAllPos` of an `Empty` returns an `Empty`  
and `assertAllPos` of `NonEmpty` returns an `NonEmpty`. There's better :

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

Here, `<: IntSet` is an upper bound of the type parameter S.

- ▶ `S <: T` means: S is a **subtype** of T, and
- ▶ `S >: T` means: S is a **supertype** of T, or T is a subtype of S.

We can *mix* them : `[S >: NonEmpty <: IntSet]`

## The Liskov Substitution Principle

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

### Variance :

Given: `NonEmpty <: IntSet` is `List[NonEmpty] <: List[IntSet]` ? yes.

when `A <: B  $\Rightarrow$  C[A] <: C[B]` we call `C[T]` **covariant**.

Does this work for all types ? let's consider `Arrays` (**mutable**)

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a // TYPE ERROR HEERE
b(0) = Empty() // CAN DO WITH ARRAY[INTSET] BUT NOT WITH ARRAY[NONEMPTY]
val s: NonEmpty = a(0)
```

Type Error Line 2 : `Array[NonEmpty]` not a subtype of `Array[IntSet]` .

Why ? Because otherwise it would contradict *Liskov Principle* **not** all you can do with `Array[IntSet]` , you can do with `Array[NonEmpty]` . (Line 3,4)

Say `C[T]` is a parameterized type and A, B are types such that  $A \leq B$  .

- `C[A] <: C[B]` C is **covariant** (e.g `List` )
- `C[A] >: C[B]` C is **contravariant**
- neither `C[A]` nor `C[B]` is a subtype of the other C is **nonvariant** (e.g `Array` )

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

### Function types :

$A1 \Rightarrow B1 \leq A2 \Rightarrow B2$  if  $A2 \leq A1$  and  $B1 \leq B2$  .

**Covariant** in return type. **Contravariant** in input type

```
trait Function1[-T, +U]:
  def apply(x: T): U
```

### Variance rules :



- covariant type parameters can only appear in method results.
- contravariant type parameters can only appear in method parameters.
- invariant type parameters can appear anywhere.

## Week 5 :

### List methods :

`xs.length` , `xs(n) ⇔ xs.apply(n)` , `xs.reverse` , `xs.updated(n, x)` , `xs.indexOf(x)` (returns -1 if `x` not in `xs` ) , `xs.contains(x)` .

Creating new Lists	All of the following are LINEAR
<code>xs.last</code>	The list's last element, exception if <code>xs</code> is empty.
<code>xs.take(n)</code>	A list consisting of the first <code>n</code> elements of <code>xs</code> , or <code>xs</code> itself if it is shorter than <code>n</code> .
<code>xs.drop(n)</code>	The rest of the collection after taking <code>n</code> elements.
<code>xs ++ ys</code>	Concatenation. <code>def ++ (ys: List[T]): List[T] = xs match case Nil =&gt; ys case x :: xs1 =&gt; x :: (xs1 ++ ys)</code>
<code>xs(n)</code>	(or, written out, <code>xs.apply(n)</code> ). The element of <code>xs</code> at index <code>n</code> .
<code>splitAt(n)</code>	pair of: ( <code>xs[1 to n]</code> , <code>xs[n to xs.length]</code> )

### Simple merge sort implementation :

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean): List[T] =
  val n = xs.length / 2
  if n == 0 then xs
  else
    def merge[T](xs: List[T], ys: List[T]) = (xs,ys) match
      case (Nil,ys) => ys
      case (xs,Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if lt(x,y) then x :: merge(xs1,ys)
        else y :: merge(xs,ys1)

    val (fst, snd) = xs.splitAt(n)
    merge(msort(fst), msort(snd))
```

`lt : (T, T) => Boolean` is a comparison function.

### Pairs :

```
val label = pair._1
val value = pair._2

val (label, value) = pair
```

## Filtering and mapping :

methods	all LINEAR
<code>xs.filter(p)</code>	Elements of <code>xs</code> verifying <code>p</code> .
<code>xs.filterNot</code>	Same as <code>xs.filter(x ⇒ !p(x))</code> ;
<code>xs.partition(p)</code>	Same as <code>(xs.filter(p), xs.filterNot(p))</code> , but computed in a single traversal of the list <code>xs</code>
<code>xs.takeWhile(p)</code>	The longest prefix of list <code>xs</code> consisting of elements that all satisfy the predicate <code>p</code> .
<code>xs.dropWhile(p)</code>	The remainder of the list <code>xs</code> after any leading elements satisfying <code>p</code> have been removed.
<code>xs.span(p)</code>	Same as <code>(xs.takeWhile(p), xs.dropWhile(p))</code> but computed in a single traversal of the list <code>xs</code> .
<code>xs.map</code>	create a new list with <code>f</code> applied to all elements of <code>xs</code> .

## Reductions :

Combining elements of lists with a given operator.

```
def sum(xs: List[Int]) = (0 :: xs).reduceLeft((x, y) ⇒ x + y)

([0, xs1, xs2, ... , xsn])
⇒ [ 0+xs1 , xs2 , ... , xsn ]
⇒ [ 0+xs1+xs2 , ... , xsn ]
⇒ 0+xs1+xs2+ ... +xsn
```

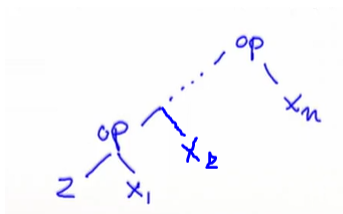
In more generality: `List(x1, ... , xn).reduceLeft(op) = x1.op(x2). ... .op(xn)`

In the same way there's `reduceRight` :

```
List(x1, ... , x{n-1}, xn).reduceRight(op) = x1.op(x2.op( ... (x{n-1}.op(xn)) )
```

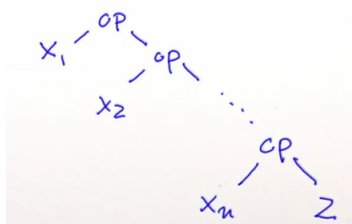
## FoldLeft :

```
def reduceLeft(op: (T, T) => T): T = this match
  case Nil => throw IllegalArgumentException("Nil.reduceLeft")
  case x :: xs => xs.foldLeft(x)(op)
// takes an accumulator
def foldLeft[U](z: U)(op: (U, T) => U): U = this match
  case Nil => z
  case x :: xs => xs.foldLeft(op(z, x))(op)
```



## FoldRight:

```
def reduceRight(op: (T, T) => T): T = this match
  case Nil => throw UnsupportedOperationException("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
// takes an accumulator
def foldRight[U](z: U)(op: (T, U) => U): U = this match
  case Nil => z
  case x :: xs => op(x, xs.foldRight(z)(op))
```



`FoldLeft` and `FoldRight` are equivalent when `op` is associative and commutative.

`FoldLeft` is `tailrec` so more efficient.

## FoldLeft/FoldRight are very useful :

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs.foldRight(ys)(_::_)
```

```
def reverse[a](xs: List[T]): List[T] =
  xs.foldLeft(List[T]())((x, zs) => x :: zs)

// List[T]() is necessary for type inference
```

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldLeft(List[U]())((x, z) => f(x) :: z)
def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)((y, n) => n + 1)
```

## Structural Induction :

We would like to verify that concatenation is associative, and that it admits the empty list Nil as neutral element to the left and to the right:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
xs ++ Nil = xs
Nil ++ xs = xs
```

We will use *Structural Induction* : To prove a property  $P(xs)$  for all lists  $xs$ ,

- show that  $P(Nil)$  holds (base case),
- for a list  $xs$  and some element  $x$ , show the induction step: if  $P(xs)$  holds, then  $P(x :: xs)$  also holds

recall the implementation of `++` :

```
extension [T](xs: List[T])
def ++ (ys: List[T]) = xs match
  case Nil => ys
  case x :: xs1 => x :: (xs1 ++ ys)
```

two facts :

1.  $Nil ++ ys = ys$
2.  $(x :: xs1) ++ ys = x :: (xs1 ++ ys)$

*Proof: Induction on  $xs$*

- **base case** : Nil

$(Nil ++ ys) ++ zs = ys ++ zs$  by 1st clause

$Nil ++ (ys ++ zs) = ys ++ zs$  also by 1st clause

- **Inductive step** : suppose  $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$  holds , prove it on  $x :: xs$

on LHS :

$((x :: xs) ++ ys) ++ zs$

$= (x :: (xs ++ ys)) ++ zs$  // by 2nd clause of `++`

$= x :: ((xs ++ ys) ++ zs)$  // by 2nd clause of `++`

$= x :: (xs ++ (ys ++ zs))$  // by induction hypothesis

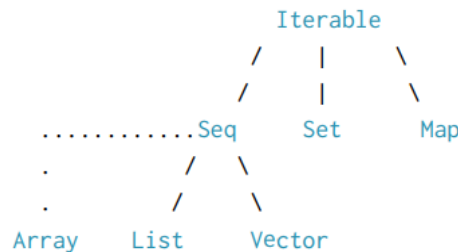
on RHS :

$(x :: xs) ++ (ys ++ zs)$

## Week 6 :

### Collections :

Collection Hierarchy :



### Arrays and String :

- Arrays and Strings support the same operations as Seq ( filter , map ... )
- They cannot be subclasses of Seq because they come from Java.

### Sequences :

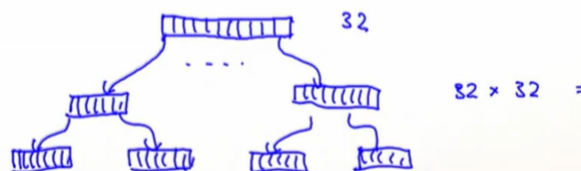
#### Range :

```
val r: Range = 1 until 5 // 5 EXCLUDED
val s: Range = 1 to 5 // 5 INCLUDED
1 to 10 by 3 // 3 IS THE STEP
6 to 1 by -2
```

### Vector :

List has *linear* time access : Access to the middle element is slower than first...

On the contrary, Vector has similar access time for all its elements.



It is implement (as shown above) in the following way : (n=number of elements)

- if  $n \leq 32$  then it is an array of 32 elements
- if  $n \leq 32 \times 32$  then it is an array of 32 elements each containing an array of 32 elements .
- if  $n \leq 32 \times 32 \times 32$  : array of array of array

Sequence operations :	
<code>xs.exists(p)</code>	true if there is an element x of xs such that p(x) holds, false otherwise.
<code>xs.forall(p)</code>	true if p(x) holds for all elements x of xs , false otherwise.
<code>xs.zip(ys)</code>	A sequence of pairs drawn from corresponding elements of sequences xs and ys.
<code>xs.unzip</code>	Splits a sequence of pairs xs into two sequences consisting of the first, respectively second halves of all pairs.
<code>xs.flatMap(f)</code>	Applies collection-valued function f to all elements of xs and concatenates the results. <code>xs.flatMap(f) = xs.map(f).flatten</code>
<code>xs.sum</code> , <code>xs.product</code> , <code>xs.max</code> , <code>xs.min</code>	the sum,product,min and max

Examples showing use of `seq` operations for conciseness :

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(_ * _).sum

def isPrime(n: Int): Boolean =
  (2 to n - 1).forall(d => n % d != 0)
```

## Combinatorial Search and For-Expressions :

Generate all pairs (i, j)  $1 \leq j < i \leq n$  :

```
(1 until n).flatMap(i =>
  (1 until i).map(j => (i, j)))
```

`For` helps for more conciseness and clarity :

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)

// THIS RETURNS A LIST OF THE (i,j)
```

### IMPORTANT

The for-expression maybe seem similar to loops in imperative languages, except that it **builds a list** of the results of all iterations.

The type of the sequence returned by the for loop **is the type of the sequence on which the first for loop is applied.**

**For and If :**

```
for p ← persons if p.age > 20 yield p.name
// EQUIVALENT TO
persons
  .filter(p => p.age > 20)
  .map(p => p.name)
```

### IMPORTANT 2 : for

```
i <- 1 until n
j <- 1 until n
```

is equivalent to **nested** loops.

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =
  (for (x, y) ← xs.zip(ys) yield x * y).sum

NOT EQUIVALENT TO

(for x ← xs; y ← ys yield x * y).sum // HERE ALL PAIRS (X,Y) are considered
```

**Maps :**

Class `Map[Key, Value]` extends the collection type `Iterable[(Key, Value)]` .

```
val romanNumerals = Map("I" → 1, "V" → 5, "X" → 10)
val capitalOfCountry = Map("US" → "Washington", "Switzerland" → "Bern")
```

The syntax `key -> value` is just an alternative way to write the pair `(key, value)` .

`toList` on a `Map` produces a `List` of pairs `(key,value)` .

**Query on map :**

```
capitalOfCountry("Andorra")
// java.util.NoSuchElementException: key not found: Andorra

capitalOfCountry.get("US") // Some("Washington")
capitalOfCountry.get("Andorra") // None
```

- The `Option` type :

```
trait Option[+A]
  case class Some[+A](value: A) extends Option[A]
  object None extends Option[Nothing]
```

we can decompose the `Option` type with pattern matching :

```
def showCapital(country: String) =
  capitalOfCountry.get(country) match
    case Some(capital) => capital
    case None => "missing data"

showCapital("US") // "Washington"
showCapital("Andorra") // "missing data"
```

- Updates :

- `m + (k -> v)` : The map that takes key `k` to value `v` and is otherwise equal to `m` .
- `m ++ kvs` The map `m` updated via `+` with all key/value pairs in `kvs` .

```
val m1 = Map("red" -> 1, "blue" -> 2) // > m1 = Map(red -> 1, blue -> 2)
val m2 = m1 + ("blue" -> 3) // > m2 = Map(red -> 1, blue -> 3)
val m3 = m1 ++ ("blue" -> 0, "yellow" -> 4) // m2 = Map(red -> 1, blue -> 0, yellow -> 4)
```

- `m - k` : The map `m` without key `k` .

- Default values : `capitalOfCountry.withDefaultValue( v )` produces a new map that has `v` as default value.

Example use of maps on `Polynoms` ( `+` operation ) :

```
class Polynom(nonZeroTerms: Map[Int, Double]):

  def terms = nonZeroTerms.withDefaultValue(0.0)
  // ===== Solution 1 : More efficient
  def + (other: Polynom) =
    Polynom(other.terms.foldLeft(terms)(addTerm))
  def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
    val (exp, coeff) = term
    terms + (exp, coeff + terms(exp))
  // ===== Solution 2 : More concise
  def + (other: Polynom) =
    Polynom(terms ++ other.terms.map((exp, coeff)
      => (exp, terms(exp) + coeff)))
```

## Week 7 :

### Queries with For :

We can use `for` to query information instead of using a combination of `filter` , `map` and `mapReduce` .



`authors` is a list of names ( `string` ).

Query the name of all the authors that have more than a book:

```
for
  b1 ← books
  b2 ← books
  if b1 ≠ b2
    a1 ← b1.authors
    a2 ← b2.authors
    if a1 = a2
      yield a1

// if authors a1="Daniel" wrote b1 and b2 this will return
// List("Daniel","Daniel") because it considers (b1,b2) and (b2,b1)

// SOLUTION 1 : use the function distinct
val repeated =
  for
    b1 ← books
    b2 ← books
    if b1.title < b2.title
      a1 ← b1.authors
      a2 ← b2.authors
      if a1 = a2
        yield a1
  repeated.distinct
```

*Better alternative* : `for` returns a sequence of the same type as the sequence we iterate over. so Iterate over set => result will be a set.

```
val bookSet = books.toSet
for
  b1 ← bookSet
  b2 ← bookSet
  if b1 ≠ b2
    a1 ← b1.authors
    a2 ← b2.authors
    if a1 = a2
      yield a1
```

## How are FOR expressions translated ?

3 rules :

1. `for x <- e1 yield e2` is translated to `e1.map(x => e2)`
2. `for x <- e1 if f; s yield e2` is translated to `for x <- e1.withFilter(x => f); s yield e2`
  - `s` is a (possibly empty) sequence of generators ( `for` ) and filters ( `if` ).
  - `withFilter` is a variation of `filter` that doesn't produce a new list but instead just executes the next `map/flatMap` on elements that pass the filter.
3. `for x <- e1; y <- e2 yield e3` is translated to `e1.flatMap(x => e2.map(y => e3))`

Example :

```
for b ← books; a ← b.authors if a.startsWith("Bird")
yield b.title
⇒
books.flatMap( b ⇒ for a ← b.authors if a.startsWith("Bird") yield b.title)

⇒ books.flatMap( b ⇒
    for a ← b.authors.withFilter(a⇒a.startsWith("Bird"))
    yield b.title)

⇒ books.flatMap(b⇒b.authors.withFilter(a⇒a.startsWith("Bird")).
    map(a⇒b.title))
```

## Generalizing for expressions :

Interestingly, the translation of for is not limited to lists or sequences, or even collections .

The presence of `map` , `flatMap` and `withFilter` suffices to be able to use `for` on some type.

## Example : random value generator :

we know how to generate a random number with the Java library :

```
val rand = java.util.Random()
rand.nextInt()
```

Let's define a trait `Generator[T]` that generates random values of type `T` :

```
trait Generator[+T]:
  def generate(): T
```

- we will instantiate it to create a random `Int` generator :

```
val integers = new Generator[Int]:
  val rand = java.util.Random()
  def generate() = rand.nextInt()
// this creates a subclass of Generator and instantiates it
```

- `Boolean` generator :

```
val booleans = new Generator[Boolean]:
  def generate() = integers.generate() > 0
```

- `pairs` generator :

```
val pairs = new Generator[(Int, Int)]:
  def generate() = (integers.generate(), integers.generate())
```

```
val booleans = for x ← integers yield x > 0 // we would like to write this
```

so we need to define `map` and `flatMap` for that

```
trait Generator[+T]:  
  def generate(): T  
  
extension [T, S](g: Generator[T])  
  
  def map(f: T ⇒ S) = new Generator[S]:  
    def generate() = f(g.generate())  
  
  def flatMap(f: T ⇒ Generator[S]) = new Generator[S]:  
    def generate() = f(g.generate()).generate()
```

we can now write :

```
val booleans = for x ← integers yield x > 0 // which translates to  
val booleans = integers.map ( x ⇒ x>0 ) // which translates to  
val booleans = new Generator[Boolean]:  
  def generate() = integers.generate() > 0
```

**More generators:**

```
def single[T](x: T): Generator[T] = new Generator[T]:  
  def generate() = x  
  
def range(lo: Int, hi: Int): Generator[Int] =  
  for x ← integers yield lo + x.abs % (hi - lo)  
  
def oneOf[T](xs: T*): Generator[T] =  
  for idx ← range(0, xs.length) yield xs(idx)  
  
def lists: Generator[List[Int]] =  
  for  
    isEmpty ← booleans // randomly check if our list should be empty  
    list ← if isEmpty then emptyLists else nonEmptyLists  
  yield list  
  
def nonEmptyLists =  
  for  
    head ← integers // first element is random  
    tail ← lists // rest of the list is random ( maybe empty)  
  yield head :: tail
```

**Application :**

- ▶ Come up with some test inputs to program functions and a postcondition.
- ▶ The postcondition is a property of the expected result.
- ▶ Verify that the program satisfies the postcondition.

```
def test[T](g: Generator[T], numTimes: Int = 100)(test: T => Boolean): Unit =
  for i <- 0 until numTimes do
    val value = g.generate()
    assert(test(value), s"test failed for $value")
    println(s"passed $numTimes tests")
```

```
test(pairs(lists,lists))(
  (l1,l2) => l1.size + l2.size == concat(l1 , l2).size
)
// tests if concat preserve the property of the sizes
// if this test fails it means concat has a bug
```

## Scala check :

```
forAll { (l1: List[Int], l2: List[Int]) =>
  l1.size + l2.size == (l1 ++ l2).size
}
```

## Monads :

Data structures with `map` and `flatMap` seem to be quite common.

These types of data structures are called `Monads` :

```
extension [T, U](m: M[T])
  def flatMap(f: T => M[U]): M[U]

  def unit[T](x: T): M[T]
```

Why `unit` ?

let's look at how to implement `map` with the help of `flatMap`

```
m.map(f) == m.flatMap(x => unit(f(x)))
         == m.flatMap(f andThen unit)
```

- `List` is a monad with `unit(x) = List(x)`
- `Set` is monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`
- `Generator` is a monad with `unit(x) = single(x)`

## Monad Laws :

To qualify as a monad, a type has to satisfy three laws:

1. **Associativity:** `m.flatMap(f).flatMap(g) == m.flatMap(f(_).flatMap(g))`  
`( ==m.flatMap(x=>f(x).flatMap(g)) )`
2. **Left unit :** `unit(x).flatMap(f) == f(x)`

*Let's check these rules for Option*

```
extension [T](xo: Option[+T])
  def flatMap[U](f: T ⇒ Option[U]): Option[U] = xo match
    case Some(x) ⇒ f(x)
    case None ⇒ None
```

- Left Unit :

```
Some(x).flatMap(f) = Some(x) match case Some(x) ⇒ f(x) case ...
= f(x)
```

- Right Unit :

```
opt.flatMap(Some(x))
= opt match
  case Some(x) ⇒ Some(x)
  case None ⇒ None
= opt
```

- Associativity :

```
opt.flatMap(f).flatMap(g)
= (opt match { case Some(x) ⇒ f(x) case None ⇒ None }) // flatMap(f)
  match { case Some(y) ⇒ g(y) case None ⇒ None }

= opt match
  case Some(x) ⇒
    f(x) match { case Some(y) ⇒ g(y) case None ⇒ None }
  case None ⇒
    None match { case Some(y) ⇒ g(y) case None ⇒ None }

= opt match
  case Some(x) ⇒
    f(x) match
      { case Some(y) ⇒ g(y) case None ⇒ None } // flatMap (g)
  case None ⇒ None

= opt match
  case Some(x) ⇒ f(x).flatMap(g)
  case None ⇒ None
```

```
= opt match
  case Some(x) ⇒
    f(x) match { case Some(y) ⇒ g(y) case None ⇒ None }
  case None ⇒ None
= opt match
  case Some(x) ⇒ f(x).flatMap(g)
  case None ⇒ None
= opt.flatMap(x ⇒ f(x).flatMap(g))
```

1. Associativity says essentially that one can “inline” nested for expressions :

```
for
  y ← for x ← m; y ← f(x) yield y
  z ← g(y)
yield z
= for x ← m; y ← f(x); z ← g(y)
  yield z
```

2. Right unit says:

```
for x ← m yield x = m
```

## Exceptions :

Exceptions in Scala are defined similarly as in Java.

An exception class is any subclass of `java.lang.Throwable` , which has itself subclasses `java.lang.Exception` and `java.lang.Error` . Values of exception classes can be thrown.

```
try e[throw ex] catch case x: Exc => handler

def validatedInput(): String =
  try getInput()
  catch
    case BadInput(msg) => println(msg); validatedInput()
    case ex: Exception => println("fatal error; aborting");
```

We can treat exceptions like normal types. `scala.util.Try` type enables this.

```
abstract class Try[+T]
  case class Success[+T](x: T) extends Try[T]
  case class Failure(ex: Exception) extends Try[Nothing]

Try(expr) // gives Success(someValue) or Failure(someException)
```

here's an implementation:

```
Here's an implementation of Try.apply:
import scala.util.control.NonFatal
object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch
      case NonFatal(ex) => Failure(ex)
```

`FlatMap` and `map` are defined on exceptions , can use `for`

```

for
  x ← computeX
  y ← computeY
yield f(x, y)
// if computeX/computeY returns Success(x)/Success(Y) we get Success(f(x,y))
// else we get Failure(ex)

```

```

extension [T](xt: Try[T]):

  def flatMap[U](f: T ⇒ Try[U]): Try[U] = xt match
    case Success(x) ⇒ try f(x) catch case NonFatal(ex) ⇒ Failure(ex)
    case fail: Failure ⇒ fail
  def map[U](f: T ⇒ U): Try[U] = xt match
    case Success(x) ⇒ Try(f(x))
    case fail: Failure ⇒ fail

```

## Week 9 :

### Structural induction on Trees :

To prove a property  $P(t)$  for all trees  $t$  of a certain type :

- show that  $P(l)$  holds for all leaves  $l$  of a tree,
- for each type of internal node  $t$  with subtrees  $s_1, \dots, s_n$ , show that  $P(s_1) \wedge \dots \wedge P(s_n)$  implies  $P(t)$ .

Let's show : `s.incl(x).contains(x) = true`

recall `Inset`

```

abstract class IntSet:
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)

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

```

**Base case :** `Empty` :

```
Empty.incl(x).contains(x)
= NonEmpty(x, Empty, Empty).contains(x) // by definition of Empty.incl
= true // by definition of NonEmpty.contains
```

## Inductive step:

for left and right subtrees `l` and `r`, we have :

`l.incl(x).contains(x) = true` and `r.incl(x).contains(x) = true`

- `y = x`

```
NonEmpty(x, l, r).incl(x).contains(x)
= NonEmpty(x, l, r).contains(x) // by definition of NonEmpty.incl
= true // by definition of NonEmpty.contains
```

- `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)
= NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl
= r.incl(x).contains(x) // by definition of NonEmpty.contains
= true // by the induction hypothesis
```

- case `y > x` is analogous

## Lazy Lists :

We want to find the n-th prime between `from` and `to` :

```
(4 to 10000).filter(isPrime)(1)
```

This works but is terrible performance wise.

**Lazy principle :** Avoid computing the elements of a sequence until they are needed for the evaluation result (which might be never).

We define `LazyList` :

- `LazyList.cons` ( can be also written `#::` equivalent to `::` for normal Lists.
- `LazyList.empty` equivalent to `Nil` for normal Lists.

They are only evaluated on demand.



```
def lazyRange(lo: Int, hi: Int): LazyList[Int] =
  if lo ≥ hi then LazyList.empty
  else LazyList.cons(lo, lazyRange(lo + 1, hi))

lazyRange(4,10000).filter(isPrime)(1)
⇒ LazyList.con(4,lazyRange(5,10000)).filter(isPrime)(1) // by def of lazyRange
⇒ lazyRange(5,10000).filter(isPrime)(1) // by filter
⇒ LazyList.con(5,lazyRange(6,10000)).filter(isPrime)(1) // by def of lazyRange
⇒ lazyRange(6,10000).filter(isPrime)(0) // def of apply
⇒ lazyRange(6,lazyRange(7,10000)).filter(isPrime)(0) // by def of lazyRange
⇒ lazyRange(7,10000).filter(isPrime)(0) // def of filter
⇒ LazyList.con(7,lazyRange(8,10000)).filter(isPrime)(0) // def of lazy range
⇒ 7
```

## Lazy evaluation :

```
lazy val x = expr // only evaluated the first time it is used , not immediately
```

Ex :

```
def expr =
  val x = {print("x");1}
  lazy val y = {print("y");2}
  def z = {print("z");3}
  z+y+x+z+y+x
// what printed ? xyz
```

Using `lazy` , we can implement a tail-lazy list:

```
def cons[T](hd: T , tl ⇒ LazyList[T]) = new TailLazyList[T]:
  def head = hd
  lazy val tail = tl
```

## Infinite sequences :

```
def from(n:Int) : LazyList[Int] = n #:: from(n+1)
val nats = from(0) // all naturals
nats.take(10).toList // :List[Int] = List(1,2, ... ,9)
```

A useful application :

```
def sieve(s: LazyList[Int]): LazyList[Int] =
  s.head #:: sieve(s.tail.filter(_ % s.head ≠ 0)) // filter all multiples of head
val primes = sieve(from(2)) // all primes in a lazyList
```

## Week 10 :

### Contextual Abstraction :

Let's try to make the sort function more context independent.

```
def msort(xs: List[Int]): List[Int] = // we want more general
⇒ def msort[T](xs: List[T]): List[T] = // problem we don't know to compare any
type T
⇒ def msort[T](xs: List[A])(lessThan: (T, T) ⇒ Boolean): List[T] =
// solution : add a comparison function
```

there's already a class that does orderings , it is `scala.math.Ordering[A]`

```
def msort[T](xs: List[T])(ord: Ordering[T]): List[T] =
...
... if ord.lt(x, y) then ...
```

Problem: Passing around Ordering arguments is cumbersome.

```
sort(ints)(Ordering.Int) // most of the time we use the same ordering
sort(strings)(Ordering.String) // to sort lists of integers , strings ...
```

We can reduce the boilerplate by making `ord` an **implicit** parameter.

```
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

Then, calls to sort can omit the `ord` parameter:

```
sort(ints) // becomes sort[Int](ints)
sort(strings) // becomes sort[String](strings)
```

We have seen that the compiler is able to infer types ( `Int` ) from values ( `ints` ) => **Type inference**.

```
sort[Int](ints) // becomes sort[Int](ints)(using Ordering.Int)
sort[String](strings) // becomes sort[String](strings)(using Ordering.String)
```

The Scala compiler is also able to do the opposite, namely to infer expressions (aka terms) from types. => **Term inference**.

### Using clauses :

An implicit parameter is introduced by a `using` parameter clause:

```
def f( x:Int )(using a:Int ) : Int = ...
f(5)(using 3) // we can also omit it and write
f(5) // it is optional
```

```
//Multiple parameters can be in a using clause:
def f(x: Int)(using a: A, b: B) = ...
f(x)(using a, b)           // notes a AND b are implicit
//Or, there can be several using clauses in a row:
def f(x: Int)(using a: A)(using b: B) = ...
//using clauses can also be freely mixed with regular parameters:
def f(x: Int)(using a: A)(y: Boolean)(using b: B) = ...
f(x)(using a)(y)(using b)
```

Parameters of a using clause can be anonymous:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] = ...
... merge(sort(fst), sort(snd)) // the ordering will be passed to merge (see
below how)
```

## Context bound :

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

```
def printSorted[T](as: List[T])(using Ordering[T]) =
  println(sort(as))
// becomes
def printSorted[T: Ordering](as: List[T]) =
  println(sort(as))
```

It means that "there must be an instance of `Ordering` that is defined on `T`" ( it is passed to sort) .

```
def f [T](ps)(using U1[T], . . . ,Un[T]) : // becomes
def f [T : U1 . . . : Un](ps) : R = ... // T has instances of U1 ... Un defined
on it.
```

## Given instances :

For the compiler to find the right value of the implicit ( `using` ) parameter, there must be a given instance:

```
object Ordering:
  //given <name of given instance> : Class[T] with ...
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int): Int =
      if x < y then -1 else if x > y then 1 else 0
```

Given instances can be anonymous. Just omit the instance name:

```
given Ordering[Double] with
  def compare(x: Int, y: Int): Int = ...
```

The compiler will synthesize a name for an anonymous instance:

```
given given_Ordering_Double : Ordering[Double] with // example of name generated
  def compare(x: Int, y: Int): Int = ...           // by the compiler
```

```
summon[Ordering[Int]]      // => Ordering.Int           | in this
summon[Ordering[Double]]  // => Ordering.given_Ordering_Double | case

def summon[T](using x: T) = x // predefined
```

Say, a function takes an implicit parameter of type `T`. The compiler will search a *given instance* that:

- has a type compatible with `T`. ( of type `T`, a subtype of `T` ... )
- is **visible** at the point of the function call, or is defined in a companion object **associated** with `T`.
  - **visible** :
    1. inherited, imported or defined in an enclosing scope.
  - **associated** :
    1. companion objects associated with any of `T`'s inherited types.

```
trait X
trait Y extends X
// if a given instance of X is required , compiler will look in Y's
// compiler
```

2. companion objects associated with any type argument in `T`.

```
trait Foo[T]
trait Bar[T] extends Foo[T]
// if a given instance of Bar[X] is required, compiler will look in X
// X here is the type argument
// Foo[X] will be looked into by the compiler
```

3. if `T` is an inner class, the outer objects in which it is embedded.

If there is a *single (most specific)* instance => it is taken as actual arguments  
Otherwise it's an error.

## Importing Given instances :

1. By-name:

```
import scala.math.Ordering.Int
```

## 2. By-type:

```
import scala.math.Ordering.{given Ordering[Int]}
import scala.math.Ordering.{given Ordering[?]}
```

## 3. With a wildcard:

```
import scala.math.given
```

## Possible errors :

```
//If there is no available given instance matching the queried type, an error
//is reported:
def f(using n: Int) = ()
f
^
//error: no implicit argument of type Int was found for parameter n of method f

// If more than one given instance is eligible, an ambiguity is reported:
trait C:
  val x: Int
  given c1: C with
    val x = 1
  given c2: C with
    val x = 2
  f(using c: C) = ()
f

//error: ambiguous implicit arguments:
//both value c1 and value c2
//match type C of parameter c of method f
```

## Priorities :

Several given instances matching the same type don't generate an ambiguity *if one is more specific than the other*.

`given a:A` is more specific than a definition `given b : B` if :

- a is in a closer lexical scope than b, or
- a is defined in a class or object which is a subclass of the class defining b, or
- type A is a generic instance of type B, or
- type A is a subtype of type B.

```
//example 1 :
class A[T](x: T)
  given universal[T](using x: T): A[T](x)
  given specific: A[Int](2)
  summon[A[Int]] // specific

// example 2 :
trait A:
  given ac: C
```

```
object O extends B:
val x = summon[C] // bc
// example 3 :
given ac: C
def f() =
  given b: C
  def g(using c: C) = ()
g // b will be chosen
```

## Type classes :

Let's consider `Ordering[T]` , it is a recurring pattern and is called **type classes**: generic classes that come with given instances for each type.

```
def sort[A: Ordering](xs: List[A]): List[A] = ...
```

At compile time , the compile resolves the right instance of Ordering to use => It is a form of polymorphism.

## Conditional instances :

Q: define an ordering for list.

A: We need ordering for its elements.

```
given listOrdering[A](using ord: Ordering[A]): Ordering[List[A]] with

def sort[A](xs: List[A])(using Ordering[A]): List[A] = ...
val xss: List[List[Int]] = ...
sort(xss)
// Given instances here will be resolved recursively
// => sort[List[Int]](xss)
// => sort[List[Int]](xss)(using listOrdering)
// => sort[List[Int]](xss)(using listOrdering(using Ordering.Int))
```

## Type classes and extension methods:

```
trait Ordering[A]:
  def compare(x: A, y: A): Int
  extension (x: A)
    def < (y: A): Boolean = compare(x, y) < 0
    def ≤ (y: A): Boolean = compare(x, y) ≤ 0
    def > (y: A): Boolean = compare(x, y) > 0
    def ≥ (y: A): Boolean = compare(x, y) ≥ 0
```

Q : When are these extensions *visible* ?

A : When given instance is visible.

```
def merge[T: Ordering](xs: List[T], ys: List[T]): Boolean = (xs, ys) match
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys) else y :: merge(xs, ys1)

// ▶ There's no need to name and import the Ordering instance to get
// access to the extension method < on operands of type T.
// ▶ We have an Ordering[T] instance in scope, that's where the extension
// method comes from.
```

Type classes provide a way to turn types into values. Unlike class extension, type classes :

- can be defined at any time without changing existing code.
- can be conditional.

## Abstract algebra and type classes :

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T // A set with an associative
                                         // operation , combine

// we can then define methods that work for any group
def reduce[T: SemiGroup](xs: List[T]): T =
  xs.reduceLeft(_.combine(_))
```

We can have hierarchies in type classes.

```
trait Monoid[T] extends SemiGroup[T]: // a semigroup with unit element
  def unit: T                       // the unit element
```

We want to write `reduce` on this function :

```
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
// We can use Context bounds
def reduce[T: Monoid](xs: List[T]): T = // there should exist
  xs.reduceLeft(summon[Monoid[T]].unit)(_.combine(_)) // Monoid[t]

// A simpler calling syntax can be obtained if we do some preparation in the
// Monoid trait itself.
trait Monoid[T] extends SemiGroup[T]:
  def unit: T
  object Monoid:
    def apply[T](using m: Monoid[T]): Monoid[T] = m
  // then
  def reduce[T: Monoid](xs: List[T]): T = // Monoid[T] is a call
    xs.reduceLeft(Monoid[T].unit)(_.combine(_)) // to apply
```

## Important remarks :

- **Context passing** : We often use implicit parameters to pass pieces of context (data) that can change , but rarely do.
- **Tamper proofing** :

```
object ConfManagement:
  type Viewers = Set[Person]
  class Conference(ratings: (Paper, Int)*):
    private val realScore = ratings.toMap
    // if a viewer in viewer is an author we don't return his paper
    def rankings(viewers: Viewers): List[Paper]:
      papers.sortBy(score(_, viewers)).reverse
```

*Problem* : One can do `conf.rankings(Set()).takeWhile(conf.score(_, Set()) > 80)` and have all the papers.

*Fix* : Make the Viewers type alias `opaque` :

```
opaque type Viewers = Set[Person]
```

the equality `Viewers = Set[Person]` is known only within the scope where the alias is defined. (in this case, within the `ConfManagement` object).

Everywhere else `Viewers` is treated as a separate, `abstract` type.

## Implicit Function types :

`Viewers => List[Paper]` is called an implicit function type.

It simply means that the argument of type `Viewers` will be implicit.