Functional Programming and the Scala Language

Lecture 6

Eugene Zouev
Innopolis University
Spring Semester 2018

Basic Scala Collections:

`List`

Scala Collections: List

How to declare and create a list?

```
val fruit = List("apples", "oranges", "pears")
```

```
val nums = List(1, 2, 3, 4)
```

```
val matrix = List( List(1, 0, 0),
                   List(0, 1, 0),
                   List(0, 0, 1) )
```

```
val empty = List()  // empty list
```

- Lists are **generic** structures
- Lists are **homogeneous** structures
- Lists are **immutable** structures
- Lists are **covariant** structures

Scala Collections: `List`

Basic operators on lists

`::` infix binary operator for ***cons***tructing lists

Lisp: CONS

`element :: list`

Right associativity

Common form: this is a list where *`element`* is its first element, and elements from *`list`* go after it

```
val nums1 = 1 :: (2 :: (3 :: (4 :: Nil)))
val nums2 = 1 :: 2 :: 3 :: 4 :: Nil
val nums3 = List(1, 2, 3, 4)
```

`head` the method returns the first element of the list

Lisp: CAR

```
val nums = List(1, 2, 3, 4)
val f = nums.head  // returns 1
```

`tail` the method returns the list starting from the second element of the initial list

Lisp: CDR

```
val nums = List(1, 2, 3, 4)
val t = nums.tail  // returns (2, 3, 4)
```

`isEmpty` the method returns **`true`** if the list is empty, and **`false`** otherwise

```
val nums = List(1, 2, 3, 4)
val e = nums.isEmpty  // returns false
```

Scala Collections: List

Example & assignment

Sorting list elements by insertions

The idea is as follows: in order to sort a non-empty list represented as x :: xs its tail xs gets sorted first, and then the first element x is inserted to the appropriate position of the result.

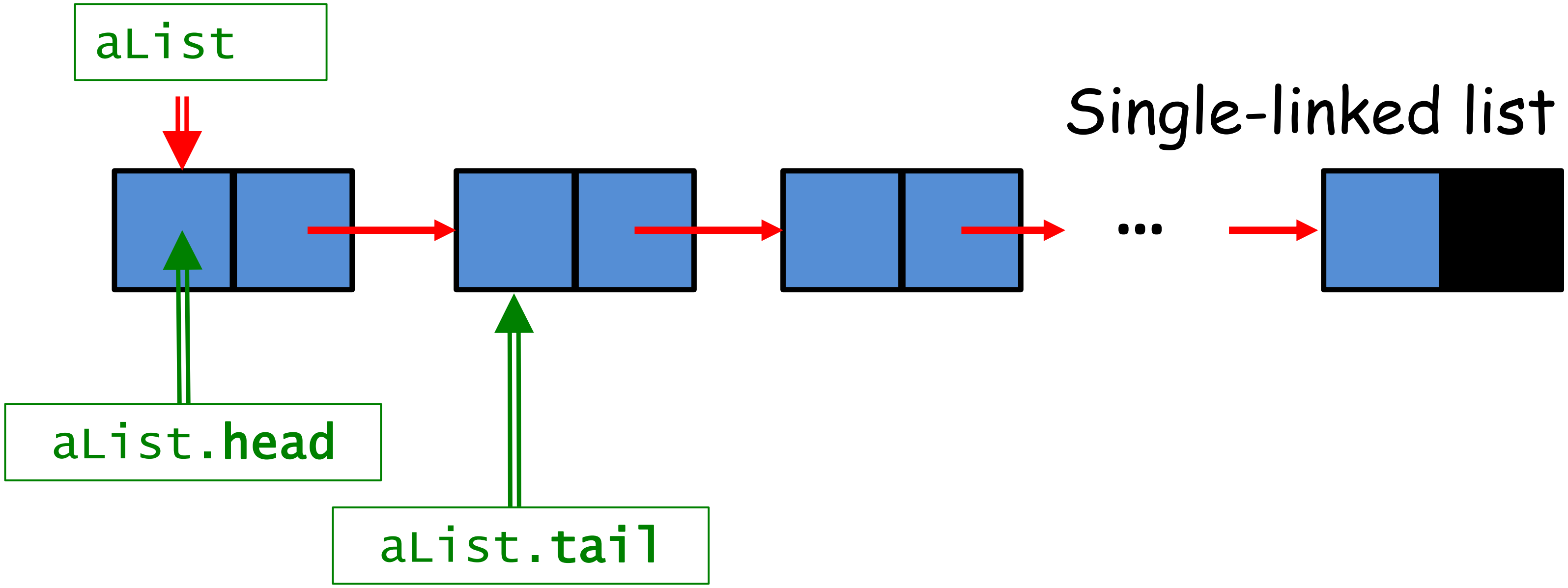
```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]) : List[Int] =
  if (xs.isEmpty || x <= xs.head) x::xs
  else xs.head :: insert(x, xs.tail)
```

The algorithm uses all basic operation on lists

The assignment was to **test** the algorithm on some real list consisting of random-generated integer values and **estimate the complexity** of the algorithm.

Scala Collections: `List`

This is **one** of possible list internal representations:



Scala Collections: List

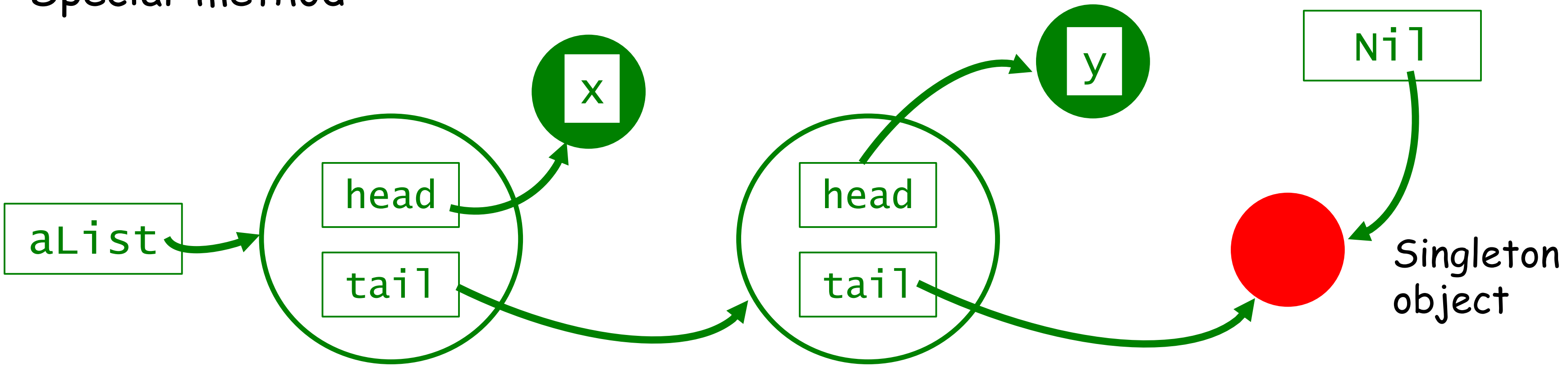
Scala's list representation

```
package scala
abstract class List[+T]
{
   def isEmpty: Boolean
   def head: T
   def tail: List[T]
   def :: ...
}
```

'+' sign indicates **covariance**.

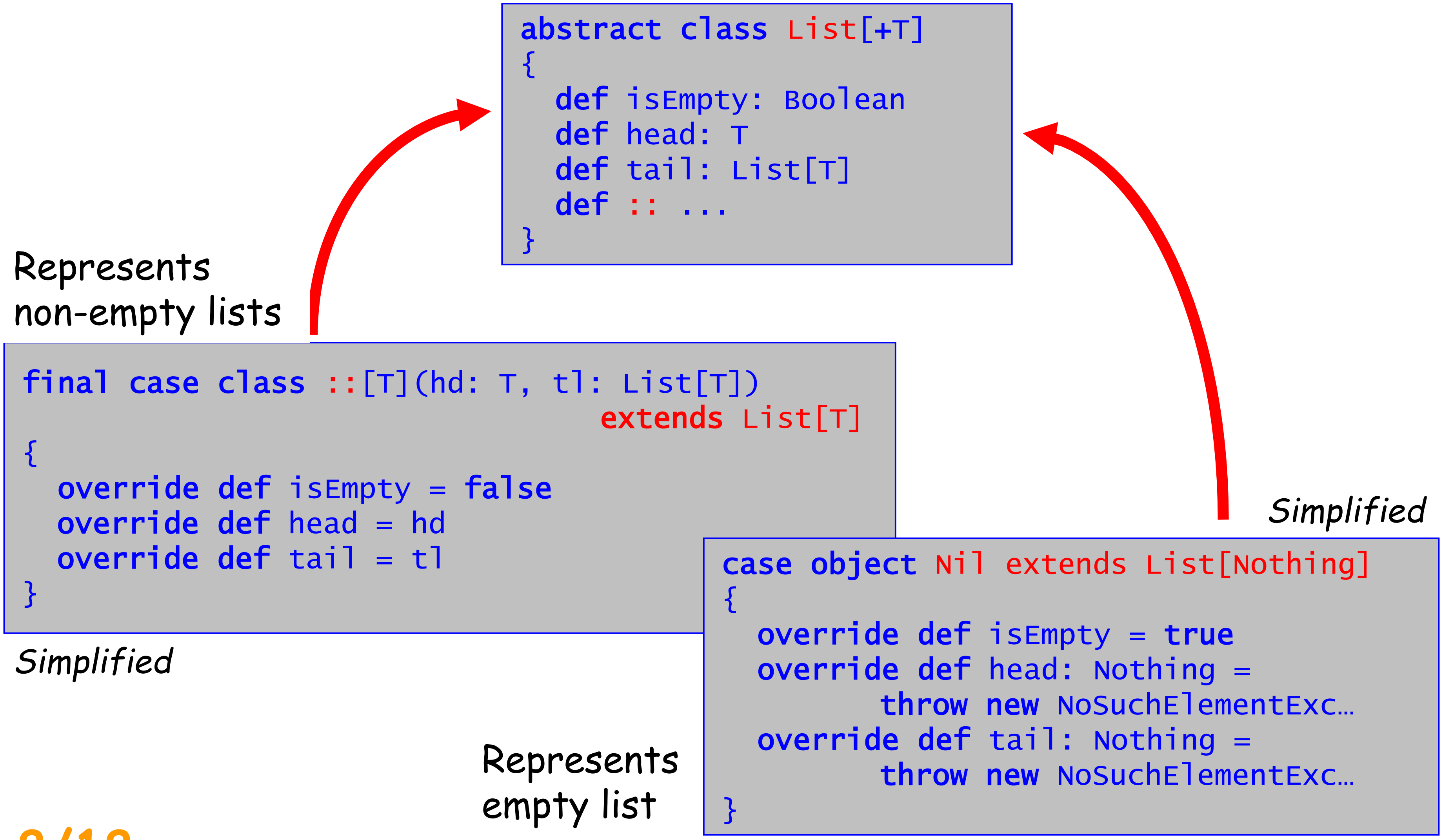
'Special' method

```
val aList = List(x,y)
```



Scala Collections: List

Scala's list hierarchy



```
abstract class List[+T]
{
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
  def :: ...
}
```

Represents non-empty lists

```
final case class ::[T](hd: T, tl: List[T])
                                  extends List[T]
{
  override def isEmpty = false
  override def head = hd
  override def tail = tl
}
```

Simplified

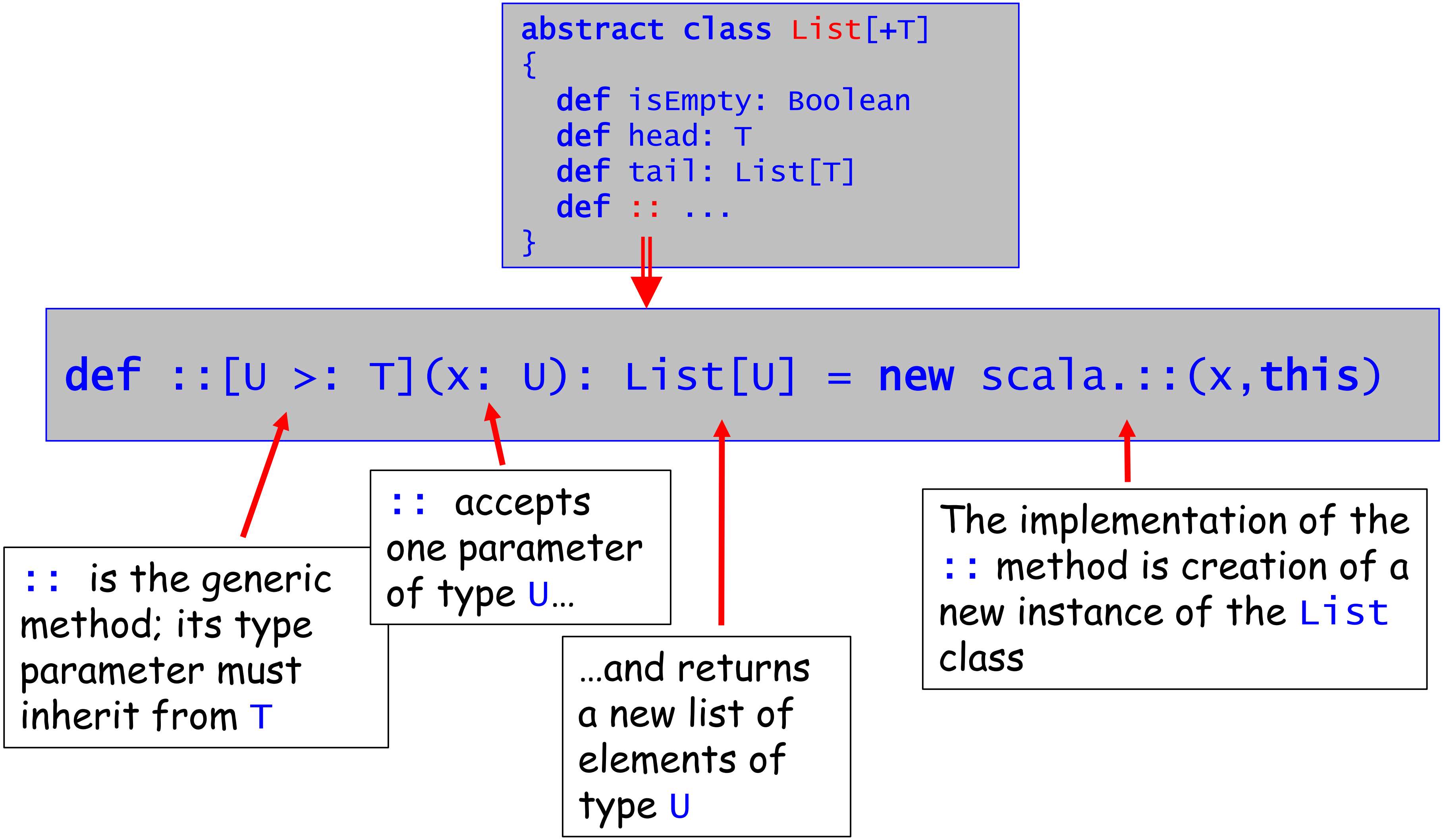
Represents empty list

```
case object Nil extends List[Nothing]
{
  override def isEmpty = true
  override def head: Nothing =
         throw new NoSuchElementExc…
  override def tail: Nothing =
         throw new NoSuchElementExc…
}
```

Simplified

Scala Collections: List

Scala's list representation



```
abstract class List[+T]
{
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
  def :: ...
}
```

```
def ::[U >: T](x: U): List[U] = new scala.::(x,this)
```

:: is the generic method; its type parameter must inherit from T

:: accepts one parameter of type U…

…and returns a new list of elements of type U

The implementation of the :: method is creation of a new instance of the List class

Scala Collections: List

Scala's list representation

Covariance means that the class

List[Int] inherits List[Any]

because

Int inherits Any

Simplified

Examples

```
val lstInt = List(1,2,3);
val lst: List[Any] = lstInt  // OK
```

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit

val apples = new Apple :: Nil
val fruits = new Orange :: apples
```

Creates a new Apple instance and creates the list with this instance

Creates a new Orange instance and adds it to the apples list - so, fruites contains one Apple instance and one Orange instance

Scala Collections: List

Some other operations on lists

`:::` the method **concatenates** (joins) two lists

Right associativity!

```
val aList1 = List(1, 2) ::: List(3, 4, 5)
```

The result is **the new list** of the form `List(1, 2, 3, 4, 5)`

```
val aList2 = List() ::: List(1, 2, 3)
```

The result is **the new list** of the form `List(1, 2, 3)`

```
val aList3 = List(1, 2, 3) ::: List(4)
```

The result is **the new list** of the form `List(1, 2, 3, 4)`

Implementation

```
def :::[U >: T](prefix; List[U]): List[U] =
   if (prefix.isEmpty) this
   else prefix.head :: prefix.tail ::: this
```

Scala Collections: List

Some other operations on lists

`length` the method returns the number of list elements

```
val len = List(1, 2, 3, 4, 5).length
```

Implementation

```
def length: Int = if (isEmpty) 0 else 1 + tail.length
```

`last` the method returns the last element of the list

```
val len = List(1, 2, 3, 4, 5).last // 5
```

Generalization of head & tail

`init` returns the initial list without its last element

```
val len = List(1, 2, 3, 4, 5).init
                             // List(1,2,3,4)
```

Implementation of `last` & `init`

The task ☺☺

Scala Collections: List

Some other operations on lists

take & drop are also generalization of head & tail

take returns the first n list elements

```
val lst = List('a','b','c','d')

val lstNew1 = lst take 2 // returns List('a','b')
val lstNew2 = lst.take(2) // the same
```

The task: try to implement take using three base methods

drop returns the list except the first n elements

```
val lst = List('a','b','c','d')

val lstNew1 = lst drop 2 // returns List('c','d')
val lstNew2 = lst.drop(2) // the same
```

Implementation:

```
def drop(n:Int):List[T] =
  if (isEmpty) Nil
  else if (n<=0) this
  else tail.drop(n-1)
```

apply returns the list element with the given number

```
val lst = List('a','b','c','d')

val c1 = lst.apply(2)  // returns 'c'
val c2 = lst apply 2   // the same
val c3 = lst(2)        // the same
```

The idea of implementation:

```
def apply(i:Int): T =
     (this drop i).head
```

Scala Collections: List

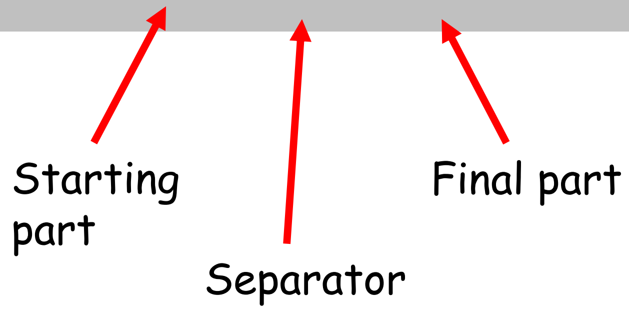
Some other operations on lists

toString returns the **standard** list literal representation

```
val lst = List(1, 2, 3, 4)

val str = lst.toString // returns "List(1, 2, 3, 4)"
```

mkString performs “customized” conversion list->String

```
val lst = List(1, 2, 3, 4)

val str1 = lst.mkString                      // "1234"
val str2 = lst.mkString("[", ",", "]")   // "[1,2,3,4]"
```



Scala Collections: List

Higher-order list methods

> “By higher-order operators, we mean higher-order functions used in operator notation. Higher-order functions are functions that **take other functions as parameters.**”

map accepts a **list** and a **function**, and returns **the new list** where each element is the result of applying the function to the corresponding element of the source list

Examples

```
val lst = List(1, 2, 3, 4).map(_ + 1) // returns List(2, 3, 4, 5)

val words = List("the", "quick", "brown", "fox")
val lens = words map (_.length)  // returns List(3,5,5,3)
```

```
def map[U](f: T=>U): List[U] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)
```

Implementation

Scala Collections: List

Higher-order list methods

foreach accepts a **list** and a **function returning nothing**, and applies the function to each source list element.

```
def foreach[U](f: T=>Unit): Unit =
  ...
```

The task:
try to implement foreach

Implementation

Example

```
val sum = 0
List(1, 2, 3, 4) foreach (sum += _)    // sum == 15
```

filter accepts a **list** and a **predicate** p **returning** Boolean, and returns the list of the elements for whom p gives **true**.

```
def filter[U](f: T=>Boolean): List[T] =
  ...
```

The task:
try to implement filter

Implementation

Example

```
val res1 = List(1, 2, 3, 4) filter (_%2==0) // List(2, 4)
val res2 = words filter (_.length==3) // List("the","fox")
```

Scala Collections: List

Higher-order list methods

forall accepts a **list** and a **predicate** p **returning** Boolean, and returns **true**, if being applied to all list elements, p **always** gives **true**. ∀

Implementation

```
def forall[U](p: T=>Boolean): Boolean =
   ...
```

The task:
try to implement forall

Example

```
List(2, 4, 6) forall (_%2 == 0)    // returns true
```

exists accepts a **list** and a **predicate** p **returning** Boolean, and returns **true**, if being applied to all list elements, p **at least once** gives **true**. ∃

Implementation

```
def forall[U](p: T=>Boolean): Boolean =
   ...
```

The task:
try to implement exists

Example

```
List(12, 24, 36) exists (_ > 40)    // returns false
```

Scala Collections: `List`

Assignments

- Implement `last` & `init`
- Implement `take` & `apply`
- Implement `foreach` & `filter`
- Implement `forall` & `exists`
- Write `reverse` function that takes a list and returns the new list with elements of the source list in the reverse order.

Almost all these functions are of 1-2-3 lines of code ☺