Lists

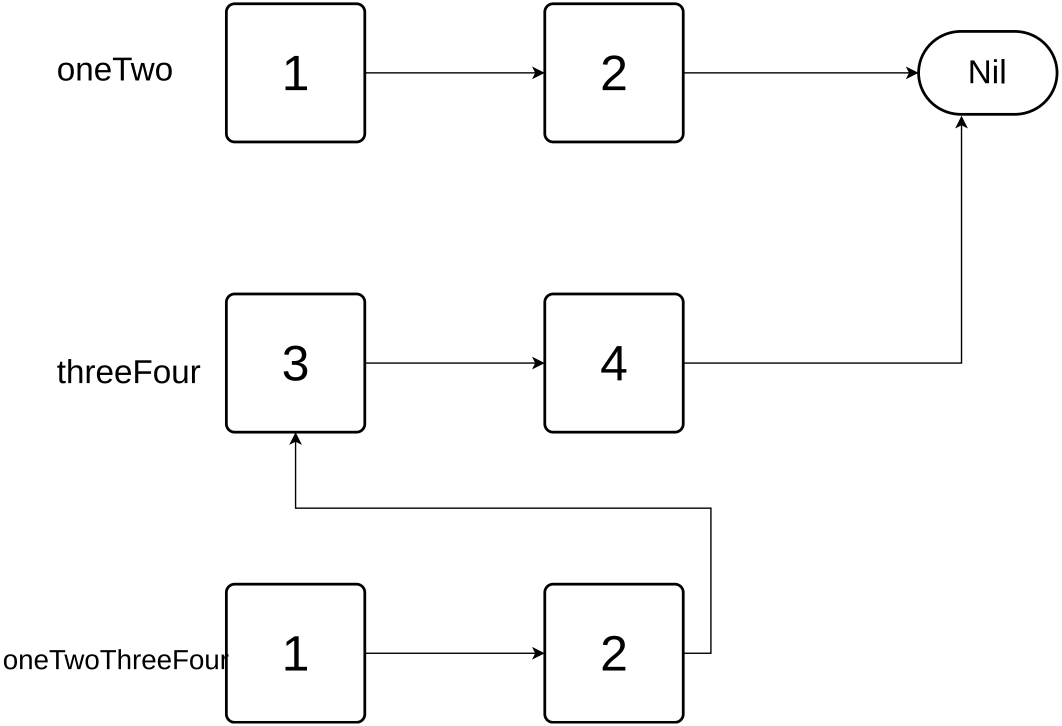
Exploring Scala's simplest functional immutable data structure

Agenda

1. The Immutable Linked List
2. Initializing / Converting to List
3. Constant Time Operations
4. Linear Time Operations
5. Higher Order Functions
6. Predicate Functions
7. Folds
8. Sorting
9. Even More Functions

The Immutable Linked List

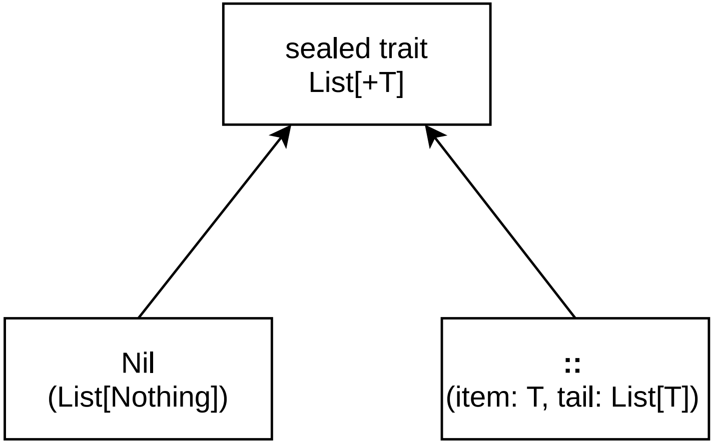
```
val oneTwo = List(1,2)                       // List(1,2)
val threeFour = 3 :: 4 :: Nil                // List(3,4)
val oneTwoThreeFour = oneTwo ::: threeFour // List(1,2,3,4)
```



oneTwo
1
2
Nil
threeFour
3
4
oneTwoThreeFour
1
2

List Properties

- Immutable
- Performance and memory efficient for head operations
- Covariant
- Always terminated by the singleton `Nil`
- Simple implementation



Initializing Lists

- Lists have a factory method on the `List` companion object

```
val oneTwo = List(1,2)
```

- Or you can use the *cons* notation

```
val threeFour = 3 :: 4 :: Nil
```

- `::` is *right associative*, so the final item must be a List (i.e. `Nil`)
- The `List` companion object also has a number of factory methods for initialization

```
List.fill(10)(0)               // List(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
List.tabulate(10)(x => x * x)  // List(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
List.range(0, 10)               // List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Converting to Lists

- Other collection types can be converted to List easily:

```
Vector('a', 'b', 'c').toList
// List[Char] = List(a, b, c)

Set(1.0, 2.0, 3.0).toList
// List[Double] = List(1.0, 2.0, 3.0)

Map(1 -> "one", 2 -> "two").toList
// List[(Int, String)] = List((1,one), (2,two))

"hello world".toList
// List[Char] = List(h, e, l, l, o,  , w, o, r, l, d)
```

- Implicits allow a String to be treated as a List of Char in most circumstances

List is Covariant

- If `A extends B` then `List[A]` is a subtype of `List[B]`
- List also widens as needed on cons and concatenation

```
val xs1 = 1 :: Nil     // List[Int] = List(1)
val xs2 = true :: xs1  // List[AnyVal] = List(true, 1)
val xs3 = "hi" :: xs2  // List[Any] = List(hi, true, 1)

def sizeOfList(xs: List[Any]): Int = xs.size

sizeOfList(xs1)  // 1
sizeOfList(xs2)  // 2
sizeOfList(xs3)  // 3
```

Constant Time Operations

- Operations at the head of a List are constant time, e.g. `.head`, `.tail`, `.isEmpty`

```
val nums = (1 to 10).toList

nums.head   // 1
nums.tail   // List(2,3,4,5,6,7,8,9,10)
nums.isEmpty // false
nums.nonEmpty // true

0 :: nums   // List(0,1,2,3,4,5,6,7,8,9,10)
nums.::(0) // List(0,1,2,3,4,5,6,7,8,9,10)
```

- `::` is also constant time, it re-uses the existing list and only creates one new node

Linear Time Operations

- Operations that are more expensive on List include

```scala
val nums1 = (1 to 5).toList

nums1.last    // 5
nums1.init    // List(1,2,3,4)
nums1.length  // 5

nums1.reverse // List(4,3,2,1)
```

- These are all linear time, each must traverse the entire list
- `init` must make a copy of the entire List minus the final element
- Lists are ideal if you work at the head exclusively, but sub-optimal for other uses

Operations that Depend on Position

- Also there are functions that depend on their parameters for their order

```
nums1(3)       // 4 (aka nums.apply(3))

nums1.drop(3)  // List(4,5)
nums1.take(3)  // List(1,2,3)

val nums2 = (6 to 10).toList

val allNums = nums1 ::: nums2  // List(1,2,3,4,5,6,7,8,9,10)

allNums.drop(8).headOption     // Some(9)
allNums.drop(20).headOption    // None
allNums.updated(4, 100)        // List(1,2,3,4,100,6,7,8,9,10)
```

- `:::` (concat) must duplicate the first List but re-uses the second
- `drop` with `headOption` will not throw an exception, even if it exhausts the list
- `updated` must make a new List up to the specified position, but re-uses the rest

Higher Order Functions

- Higher Order Functions are simply functions that take other functions

```scala
val words = List("four", "four", "char", "word")
// List(four, four, char, word)

words.map(_.reverse)
// List(ruof, ruof, rahc, drow)

words.reverse.map(_.reverse)
// List(drow, rahc, ruof, ruof)

words.map { word => word.toList }
// List(List(f, o, u, r), List(f, o, u, r), List(c, h, a, r), List(w, o, r, d))

words.flatMap { word => word.toList }
// List(f, o, u, r, f, o, u, r, c, h, a, r, w, o, r, d)

words foreach println
// four
// four
// char
// word
```

Predicate Based Functions

- A predicate is just a function returning `Boolean`, as such predicate based functions are higher order functions

```
words.filter(_.contains("a"))      // List(char)
words.filter(_.contains("f"))      // List(four, four)

words.find(_.contains("a"))        // Some(char)
words.find(_.contains("z"))        // None
words.indexWhere(_.contains("a"))  // 2
words.indexWhere(_.contains("z"))  // -1
words.indexWhere(_.contains("r"))  // 0
words.lastIndexWhere(_.contains("r"))  // 3

words.filterNot(_.contains("a"))   // List(four, four, word)
words.partition(_.contains("a"))   // (List(char),List(four, four, word))

words.takeWhile(_.contains("f"))   // List(four, four)
words.dropWhile(_.contains("f"))   // List(char, word)
```

Folds

```
val words = List("four", "four", "char", "word")
val nums = List(2,3,5,8,13,21)

val sumNums = nums.foldLeft(0)((a, b) => a + b)  // 52
val prodNums = nums.foldLeft(1)(_ * _)           // 65520

val asString = words.foldLeft("")(_ + ", " + _)  // , four, four, char, word
```

- Can also use `foldRight` or just `fold`, but `foldLeft` works best for List traversal
- There is also `reduceLeft` etc

```
val sum2 = nums.reduceLeft(_ + _)  // 52

// but!
List.empty[Int].foldLeft(0)(_ + _) // 0
List.empty[Int].reduceLeft(_ + _)  // UnsupportedOperationException
```

Fold Alternatives

- For many common fold operations, there are ready-made alternatives. e.g. for Lists of Numerics

```
nums.sum     // 52
nums.product // 65520
```

- and for any kind of List where you want to create a string representation:

```
words.toString                // List(four, four, char, word)
words.mkString                // fourfourcharword
words.mkString(",")           // four,four,char,word
words.mkString("[", ",", "]") // [four,four,char,word]
```

Sorting

```scala
case class Person(name: String, age: Int)
val xs = List(Person("Harry", 25), Person("Sally", 23), Person("Fred", 31))

xs.sortWith((p1, p2) => p1.age < p2.age)
// List(Person(Sally,23), Person(Harry,25), Person(Fred,31))

xs.sortBy(_.name)
// List(Person(Fred,31), Person(Harry,25), Person(Sally,23))

List(5, 2, 3, 4, 8, 1, 7).sorted
// List(1, 2, 3, 4, 5, 7, 8)
```

- `sorted` requires definition of an `Ordering[T]` for `List[T]`

```scala
implicit object PersonOrdering extends Ordering[Person] {
  override def compare(x: Person, y: Person) = {
    if (x.name == y.name) x.age - y.age
    else if (x.name > y.name) 1 else -1
  }
}
xs.sorted
// List(Person(Fred,31), Person(Harry,25), Person(Sally,23))
```

Even More Functions

- Need to transpose a matrix?

```
val matrix = List(List(1,2,3), List(4,5,6), List(7,8,9))
// List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))
val transpose = matrix.transpose
// List(List(1, 4, 7), List(2, 5, 8), List(3, 6, 9))
```

- Sum up all the numbers in that matrix:

```
matrix.flatten.sum // 45
```

- Group by first letters of a word:

```
val words = List("four", "four", "char", "word")
words.groupBy(_.head)
// Map(w -> List(word), c -> List(char), f -> List(four, four))
```

Even More Functions

- Filter by a type in a List, and return just a List of that type

```scala
trait Fruit
case class Apple(name: String) extends Fruit
case class Orange(name: String) extends Fruit

val fruits = List(Apple("Fiji"), Orange("Jaffa"), Apple("Cox's")) // List[Fruit]

fruits.collect {
  case a: Apple => a
}  // List[Apple] = List(Apple(Fiji), Apple("Cox's"))
```

- `collect` is like a `filter` and `map` combined into one, takes a `PartialFunction`, and will narrow the resulting List type if possible

Permutations and Combinations

```scala
val nums = List.range(0, 10)
// List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

nums.grouped(3).take(5).toList
// List(List(0, 1, 2), List(3, 4, 5), List(6, 7, 8), List(9))

nums.sliding(3).take(5).toList
// List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4), List(3, 4, 5), List(4, 5, 6))

nums.combinations(3).take(5).toList
// List(List(0, 1, 2), List(0, 1, 3), List(0, 1, 4), List(0, 1, 5), List(0, 1, 6))

nums.permutations.take(5).toList
// List(List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), List(0, 1, 2, 3, 4, 5, 6, 7, 9, 8),
//      List(0, 1, 2, 3, 4, 5, 6, 8, 7, 9), List(0, 1, 2, 3, 4, 5, 6, 8, 9, 7),
//      List(0, 1, 2, 3, 4, 5, 6, 9, 7, 8))

val numsPlusOne = nums.map(_ + 1)

nums.corresponds(numsPlusOne)((a, b) => a + 1 == b) // true
```

Indices, zip, unzip

```
val chars = List.range('a', 'h')
// List[Char] = List(a, b, c, d, e, f, g)

val idx = chars.indices
// scala.collection.immutable.Range = Range 0 until 7

chars.zip(idx)
// List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4), (f,5), (g,6))

val zipped = chars.zipWithIndex
// List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4), (f,5), (g,6))

zipped.unzip
// (List[Char], List[Int]) = (List(a, b, c, d, e, f, g),List(0, 1, 2, 3, 4, 5, 6))
```

- And many, many more…
- https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

Exercises for Module 13

- Find the `Module13` class and follow the instructions to make the tests pass
- These exercises are based on simplified versions of a real problem I needed to solve using the collections API.
- These do use `Sets` in a simple way, to make a `Set` from a `c: Char`, just use `Set(c)`
- If you add something to a `Set` that is already in the `Set`, it will not be added again so that you can add the same thing any number of times and it will only be in there once.
- We'll learn more about `Set` in the next module, but this will be enough to complete these exercises.