More Operations on Lists

Effective Programming in Scala

More Operations on Lists

In the previous lesson we have seen how to construct lists and how to do some basic operations with them.

In this lesson, we will cover the following:

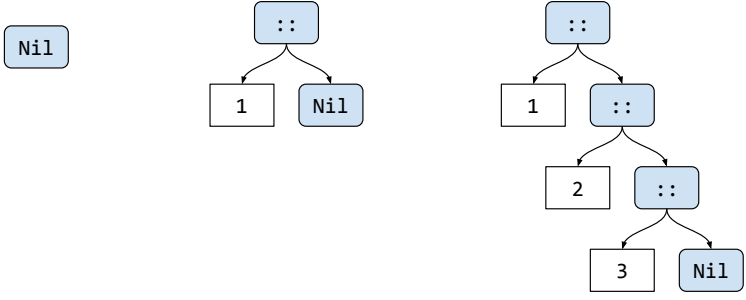
- how to insert an element into a list?
- how do we access to one particular element of a list?
- what is the data layout of lists, how does it impact the way we work with lists?

List Data Layout

To answer these questions, let's first have a look at the data layout of lists.

A `List` is:

- the empty list `Nil`, **or**
- a pair containing a head element and a tail that is a `List`.



Lists Are Immutable

Another important point is that Scala lists are **immutable**: once you create a list, you can't change its elements.

If you want a different list, you have to create another one.

We will elaborate later on the motivations for using immutable data types. For now, let's see how to work with them.

Inserting an Element into a List

Consider the following list of contacts, containing Alice and Bob:

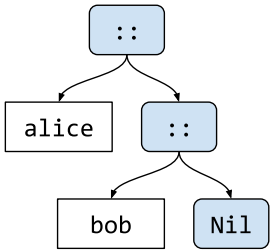
```
val contacts1 = List(alice, bob)
// contacts1 = List(alice, bob)
```

We can create another list with Carol, Alice, and Bob like so:

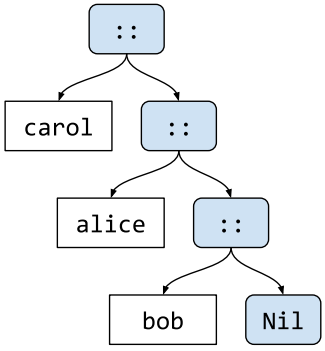
```
val contacts2 = carol :: contacts1
// contacts2 = List(carol, alice, bob)
```

We haven't changed the list `contacts1`. Instead, we have constructed a new list, `contacts2`, whose head element is `carol` and whose tail list is `contacts1`.

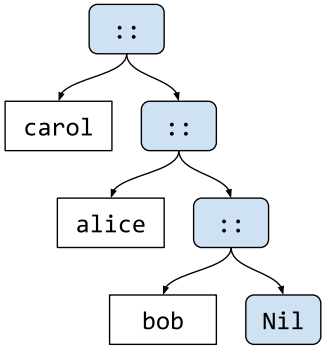
Inserting an Element into a List (2)



Inserting an Element into a List (3)



Inserting an Element into a List (3)



Constructing a new list by prepending an element to an existing list is a constant-time operation: we don't copy the tail list, we just **reuse** it.

This type of data structures is also called **persistent data structures** because we never change the previous state of a list.

Another Perspective at Constructing Lists

Calling the `List` constructor with several elements is equivalent to prepending these elements to the empty list `Nil`:

```
List(alice, bob) == alice :: bob :: Nil
```

Another Perspective at Constructing Lists

Calling the `List` constructor with several elements is equivalent to prepending these elements to the empty list `Nil`:

```
List(alice, bob) == alice :: bob :: Nil
```

Note that the `::` operator is **right-associative**:

```
alice :: bob :: Nil == (alice :: (bob :: Nil))
alice :: bob :: Nil == Nil.::(bob).::(alice)
```

Another Perspective at Constructing Lists

Calling the `List` constructor with several elements is equivalent to prepending these elements to the empty list `Nil`:

```
List(alice, bob) == alice :: bob :: Nil
```

Note that the `::` operator is **right-associative**:

```
alice :: bob :: Nil == (alice :: (bob :: Nil))
alice :: bob :: Nil == Nil.::(bob).::(alice)
```

- More generally, operators that end with ":" are right-associative. This contrasts with other operators such as `+`

```
0 + 1 + 2 == ((0 + 1) + 2) == 0.+(1).+(2)
```

Pattern Matching on Lists

Conversely, we can decompose a list into its head and tail by using pattern matching.

For instance, here is a program that prints the name of the first contact of an address book:

```
addressBook.contacts match
  case contact :: tail => println(contact.name)
  case Nil             => println("No contacts")
```

Pattern Matching on Lists (2)

Decomposing a list into its head and tail is symmetrical to constructing a list from a head and a tail.

For instance, to print the name of the second contact of an address book:

```
val addressBook = AddressBook(alice :: bob :: Nil)
addressBook.contacts match
  case first :: second :: Nil => println(second.name)
  case _                      => println("Unexpected number of contacts")
```

Pattern Matching on Lists (2)

Decomposing a list into its head and tail is symmetrical to constructing a list from a head and a tail.

For instance, to print the name of the second contact of an address book:

```
val addressBook = AddressBook(alice :: bob :: Nil)
addressBook.contacts match
  case first :: second :: Nil => println(second.name)
  case _                      => println("Unexpected number of contacts")
```

- Note that we use the **wildcard pattern** "_" to match an empty list of contacts, a list of exactly one contact, and a list with more than two contacts.

Exhaustivity Checks on Lists

Here is what the compiler would tell us if we wrote "`case Nil`" instead of "`case _`"

```
addressBook.contacts match
  case first :: second :: Nil => println(second.name)
  case Nil                    => println("Unexpected number of contacts")
```

```
warning: match may not be exhaustive.

It would fail on pattern case: List(_, _, _, _ *), List(_)
```

Random Access

An alternative way to access the elements of a list is to use its operations `head`, `tail`, or random access:

```
val fruits = List("apples", "oranges", "pears")
fruits.head == "apples"
fruits.tail == List("oranges", "pears")
fruits.tail.head == "oranges"
fruits(0) == "apples"
fruits(2) == "pears"
```

- Note that these operations raise an **error** if you try to access the head or the tail of an empty list, or if you use an index that is out of the bounds of the list.
- Also, note that random access is not efficient on lists (see the next slides).

Performance Characteristics

The table below summarizes the performance characteristics of the `List` data structure:

Operation	Complexity
`::`	Constant time
`head`	Constant time
`tail`	Constant time
Random access	Linear time
`size`	Linear time

Summary

Lists are linear immutable sequences.

They can be constructed and decomposed with the "`::`" operator.

They are not optimized for random access but support efficient `head` and `tail` decomposition.