Constructing Collections

Effective Programming in Scala

Constructing Collections

In this section we'll learn some of the many ways that can be used to construct collections.

Constructing: Empty

The simplest example is an empty collection. Let's construct one.

```
List.empty  // List[Nothing]
mutable.ArrayBuffer.empty  // ArrayBuffer[Nothing]
Map.empty // Map[Nothing, Nothing]
```

Constructing: Empty

Scala inferred the type of the elements of our collections as `Nothing`. This is probably not what we want. We can instead specify the type.

```
List.empty[Int] // List[Int]
mutable.ArrayBuffer.empty[Double] // ArrayBuffer[Double]
Map.empty[String, Boolean] // Map[String, Boolean]
```

Constructing: Empty

Scala inferred the type of the elements of our collections as `Nothing`. This is probably not what we want. We can instead specify the type.

```
List.empty[Int] // List[Int]
mutable.ArrayBuffer.empty[Double] // ArrayBuffer[Double]
Map.empty[String, Boolean] // Map[String, Boolean]
```

Note: for `Map` we specify both the key and value types.

Constructing: Varargs

Constructing empty collections is not always useful. There is a constructor to create a collection with an arbitrary number of elements. We've already seen this for `List`.

```
List(1, 2, 3, 4)
```

It works for `ArrayBuffer` as well.

```
mutable.ArrayBuffer("a", "b", "c", "d")
```

This is sometimes known as a *vararg* constructor, because we can pass a variable number of arguments to it.

Constructing: Varargs

To use the vararg constructor for `Map` we must pass both keys and values.

```
Map("a" -> true, "e" -> true, "b" -> false)
```

This creates a map with the key "a" associated with the value `true` and the key "b" associated with the value `false`.

The syntax `a -> b` constructs a *tuple*. Let's take a quick look at tuples before we move on.

Aside: Tuples

A tuple is a collection of *fixed size* but the values in the collection may have *different types*. In contrast collections like `List` can hold a variable number of elements but each element must have the same type.

The syntax `a -> b` constructs a tuple with two elements. It is a shorthand for writing `(a, b)`, which is the general syntax for constructing tuples. For tuples of three or more elements we must surround the values with parentheses and have commas between them. Here are some examples.

```
val pair1: (String, Int) = "Alice" -> 42
val pair2: (String, Int) = ("Alice", 42)
val tuple: (Int, String, Boolean) = (42, "foo", true)
```

More generally, a tuple type `(T1, T2, ..., Tn)` is a type containing `n` elements of type `T1`, `T2`, ..., and `Tn`, respectively.

Aside: Tuple Patterns

We now know how to construct tuples. Let's look at how we get values out of them. A pattern match is the easiest way to deconstruct a tuple.

```
(10.0, "Hello") match
  case (number, greeting) => s"$greeting! The number is $number"
```

We can use a pattern in a `val`, which is another convenient way to deconstruct a tuple.

```
val (x, y) = (10.0, 20.0)
x // 10.0
y // 20.0
```

Aside: Accessing Tuple Elements

We can also directly access elements of a tuple by index. To do so we call a tuple as if it was a function, passing the index of the element as a parameter. Indices start at 0.

```
val pair = (”Alice”, 42)
pair(0) // : String = ”Alice”
pair(1) // : Int = 42
```

Constructing: Prepending and Appending Elements

If we already have a sequence we can prepend and append individual elements using the +: and :+ methods.

```
0 +: List(1, 2, 3) // List(0, 1, 2, 3)
mutable.ArrayBuffer("a", "b") :+ "c" // ArrayBuffer("a", "b", "c")
```

Constructing: Updating a Map

We cannot call `+:` and `:+` on `Map` because it does not store values in any particular order. Instead, we can add an element using the `+` method.

```
Map("a" -> true) + ("e" -> true) // Map(a -> true, e -> true)
```

Constructing: Summary

We have seen the following methods to construct collections:

- `empty` to construct an empty collection;
- the varargs constructor;
- `+:` and `:+` to prepend and append elements to sequences; and
- `+` to add an element to a map.

We have also learned about tuples.