Chapter 21
Tuples

21.1 Introduction

In this chapter we introduce Scala tuples, which are useful (and very simple construct) for grouping instances together.

21.2 Tuples

Tuples are container style objects that can hold multiple types of instances. Examples of some simple tuples are presented below:

- (49, "John") // a 2 element tuples
- (49, "John", "Hunt", 12.75) // a 4 element tuples
- ("John", 76, Invoice(123)) // a 3 element tuples

As can be seen from these examples, tuples can hold different types of elements (whereas an array instance holds a set of the same type such as Strings, Ints, Doubles or Persons). The first example is tuple which presented here is:

- (49, "John")

This holds an Int and a String. The first element in the tuple is of type Int and the second is of type String. The second example is a three element tuples:

- (49, "John", "Hunt", 12.75)

In this example, the first element is of type Int, the second of type String, the third again of type String and the fourth of type Double. In fact it is possible to define tuples with 1–22 elements in them.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_21

21.3 Tuple Characteristics

Tuples exhibit a number of characteristics that are worth noting. Tuples

- Are container style objects
- Are immutable sequences of instances
- Can contain objects of different types (unlike lists and arrays)
- Are useful if you need to return more than one object from a method
- Are not collections as they do not support the *normal* collection style methods
- Allow access to their contents via a dot (.) notation
- Can be simply created without the need for the new keyword
- Are backed by specific Tuple classes.

21.4 Tuple Classes

Each tuple is backed by a class named Tuple followed by the number of elements in the tuple. For example,

(42, "Denise")—is an example of a Tuple2 instance
("John")—is an example of a Tuple1 instance
(49, "John", "Hunt", 12.75)—is an example of a Tuple4 instance

In actual fact there are classes named for all the different combinations of Tuple from 1 to 22, for example,

Tuple1, Tuple2, Tuple3, . . .Tuple21, Tuple22

Mostly this fact is hidden from you as a programmer, but it is worth being aware of. Essentially when you create a new tuple example, an instance of the appropriate type is created with each of the elements used to infer the type to use for those positions. Thus,

("Pete", 21)

Tells Scala to create an instance of the Tuple2 type, with the first element set to String and the second element set to Int types.

21.5 Creating a Tuple

There are a number of different ways in which you can create a new Tuple. The longhand form is to create a Tuple of the correct type and use type parameterisation to indicate the types to use for each position. For example,

```
val t = new Tuple2[Int, String](1, "John")
```

This creates a new instance of the Tuple2 type with the first element being of type Int and the second element being of type String. The actual values held by the tuple instance are then 1 and "John".

This is a bit long winded for Scala and thus a shorter form of the above allows the types to be inferred by Scala. For example, the above can be written without the type parameterisation information as:

```
val t = new Tuple2(1, "John")
```

Now Scala infers that the first element is an Int, and the second element is a String from the values being held by the tuple (namely 1 and "John").

However, this is still a bit repetitive for Scala. Given that we are creating a tuple and there are two elements. Scala can infer that what we are creating a Tuple2 instance. Therefore we do not need to include the name of the type being instantiated. We also do not need to include the keyword *new*. Thus the previous example can be reduced to:

```
val pair = (1, "John")
```

This creates an instance of the Tuple2 class with the first value of type Int and set to 1 and the second element of type String and set to "John".

Similarly we could write:

```
val t = (54, "John", "Hunt", 12.75)
```

which creates an instance of the Tuple4 class with the types being Int, String, String and Double, respectively.

The very shorted form of the Tuple syntax is to use the '->' syntax. For example,

```
val pair2 = 1 -> "John"
```

which again creates a Tuple2 instance containing 1 and "John" with the first element of type Int and the second element of type String.

21.6 Working with Tuples

Once you have created a tuple using one of the forms described in the previous section you can access information about the tuple as well as the information held in the tuple. For example, to access the size of the tuple you can use productArity:

```
val tuple = (54, "John", "Hunt", 12.75)
// Length of a tuple
println(tuple.productArity)
```

The output of this program is 4.

To access the individual elements within a tuple we can use the "._<n>" notation where <n> indicates the position or element to be accessed, for example,

```
val tuple = (54, "John", "Hunt", 12.75)
println(tuple._1)
println(tuple._2)
println(tuple._3)
println(tuple._4)
```

Note that the index used for accessing Tuples is one based (where as that used for Arrays and Lists is Zero based). As Tuples are immutable there is no way to update an element within a tuple. However we can make a copy of a tuple and while making a copy change one of the values. For example,

```
val tuple = (54, "John", "Hunt", 12.75)
println(tuple)
println(tuple.copy(_2 = "Bob"))
```

The output of this program is:

```
(47,John,Hunt,12.75)
(47,Bob,Hunt,12.75)
```

21.7 Iterating Over a Tuple

If you want to iterate over a tuple, then you can obtain a product iterator built on top of your tuple. A product iterator provides a wrapper that can access each element in a tuple in turn. A function can then be applied to each element in turn. Thus we can write a simple tuple iterator that will print out the values in our tuple one at a time.

```
tuple.productIterator.foreach { x => println(x) }
```

This states that we want to create an iterator over the contents of the tuple and then for each of the elements produced apply the function that takes a single parameter (x) and applies the println(x) method. A shorthand form of this can imply the parameter as it is only used to temporarily pass a given element in the tuple to the *println* method. We can therefore infer this in Scala and write the following (where the under bar '_' represents a placeholder for the parameter x):

```
tuple.productIterator.foreach{ println _ }
```

The output of either of the above versions of the *foreach* operation is:

```
47
John
Hunt
12.75
```

21.8 Element Extraction

It is also possible to extract the values held in a tuple using simple matching variables. For example, given a tuple t1 containing two elements an Int and a String, we can extract them as follows:

```
val t1 = (1, "John")
val (a, b) = pair
println(a + ": " + b)
```

Here the *val* 'a' is of type *Int* and the *val* 'b' is of type *String*. The *val* 'a' will hold 1, and the *val* 'b' will hold a reference to the string "John". The output of this program is thus:

```
1: John
```