Using Pair and Triple

We'll cover the following

What are tuples?

Tuples are sequences of objects of small, finite size. Unlike some languages that provide a way to create tuples of different sizes, Kotlin provides two specific types: `Pair` for a tuple of size two and `Triple` for a size of three. Use these two when you want to quickly create two or three objects as a collection.

The `Pair` tuple

Here's an example of creating a Pair of Strings:

```
println(Pair("Tom", "Jerry")) //(Tom, Jerry)
println(mapOf("Tom" to "Cat", "Jerry" to "Mouse")) //{Tom=Cat, Jerry=Mouse}
```

First we create an instance of `Pair` using the constructor. Then we use the `to()` extension function, that's available on any object in Kotlin, to create pairs of entries for a `Map`. The `to()` method creates an instance of `Pair`, with the target value as the `first` value in the `Pair` and the argument provided as the `second` value in the `Pair`.

The ability to create a pair of objects with such concise syntax is useful. The need to work with a pair of objects is common in programming. For example, if you have a list of airport codes and want to get the temperature at each of these airports, then representing the airport code and temperature as a pair of values is natural. In Java, if you hold the values in an array, it'll get cumbersome to work with. Besides, we'll lose type safety since airport code is a `String` and temperature is a `double`, and the array will end up being of type `Object`—smelly. In Java we normally create a specialized class to hold the two values. This approach will provide type safety and remove some noise in code, but it increases the burden on us to create a separate class just for this purpose. Java provides no pleasant way to deal with this. Kotlin Pair solves the issue elegantly.

To see the benefit of `Pair`, let's create an example to collect the temperature values for different airport codes.

```
val airportCodes = listOf("LAX", "SFO", "PDX", "SEA")

val temperatures =
  airportCodes.map { code -> code to getTemperatureAtAirport(code) }

for (temp in temperatures) {
  println("Airport: ${temp.first}: Temperature: ${temp.second}")
}
```

airporttemperatures.kts

We iterate over the collection `airportCodes` using the functional-style `map()` iterator (which you'll learn about in Chapter 12, Internal Iteration and Lazy Evaluation) to transform each airport code in the list to the pair of (code, temperature). The result is a list of `Pair<String, String>`. Finally, we loop through the values in the list of `Pairs` to print the details of each airport code and temperature at that location. For each `Pair`, we obtain the two contained values using the `first` and `second` property, respectively.

If you're curious about the `getTemperatureAtAirport()` function used in this code, we'll implement a working code to talk to a web service later in the course. For now, let's implement a fake function to keep the focus on the benefits of Pair.

```
val airportCodes = listOf("LAX", "SFO", "PDX", "SEA")

val temperatures =
  airportCodes.map { code -> code to getTemperatureAtAirport(code) }

for (temp in temperatures) {
  println("Airport: ${temp.first}: Temperature: ${temp.second}")
}

fun getTemperatureAtAirport(code: String): String =
  "${Math.round(Math.random() * 30) + code.count()} C"
```

airporttemperatures.kts

Run the code and watch the program output the fake temperatures for the given airports. Here's a sample of what I got on a run:

```
Airport: LAX: Temperature: 25 C
Airport: SFO: Temperature: 21 C
Airport: PDX: Temperature: 30 C
Airport: SEA: Temperature: 27 C
```

This example shows the use of `Pair` in a practical setting. Use it anywhere you'll need a pair of objects or tuple. You not only get concise code, it's type safe at compile time as well.

`Pair` is useful when working with two values. While it looks special, it's just another class written in the Kotlin standard library. You may create your own classes like that where you need.

The `Triple` tuple

If you have a need for three objects, then instead of `Pair` use `Triple`. For example, if you need to represent the position of a circle, you don't have to rush to create a Circle class. Instead, you may create an instance of `Triple<Int, Int, Double>` where its `first` value represents the center X, the `second` value the center Y, and finally the `third` value, of type `Double`, holds the radius. That's less code while getting type safety.

Both `Pair` and `Triple` are immutable and are useful to create a grouping of two and three values, respectively. If you need to group more than three immutable values, then consider creating a *data class* (see Data Classes).

The Kotlin standard library takes care of your needs to keep two or three immutable values. But if you need a mutable collection of values, `Array` may be a good choice, as we'll see in the next lesson.