Chapter 28
Type Parameterisation

28.1 Introduction

This chapter introduces and discusses type parameterisation. We have already seen examples of type parameterisation in the collection types where a List can be parameterised to hold only Strings, Person objects or integers. In this chapter, we will look at how you can create your own parameterised types and the options available for controlling the types that can be used with those parameterisations.

28.2 The Set Class

As a concrete example of type parameterisation, we will explore the Set class. The Set class allows the type of element that it will hold to be specified between square brackets '[..]' after the name of the class and before any parameters passed to the Set constructor. For example

```
object SetTest extends App {

  val s = Set[String]("John", "Denise", "Phoebe",
                      "Adam")
  println(s)

}
```

In the above listing, the Set class is being parameterised to only hold items of type String (hence the Set[String]). If we wanted it to be limited to only holding integers then we would use Set[Int] and to only hold Person types then we would use Set[Person]. The type parameterisation is the application of [String] or [Int] to the type Set. Of course, this is Scala, so Scala is quite capable of inferring this from the contents of the Set, thus

© Springer International Publishing AG 2018

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_28

```
val s2 = Set("John", "Denise", "Phoebe", "Adam")
```

s2 also references a Set that has been parameterised to only hold Strings.

In both cases the class Set is referred to as a *generic* class. If you look at the definition of Set in the Scaladoc then you will see that it is defined as being:

```
Set[T]
```

Which indicates that Sets can hold elements of a type T where T is a placeholder for the type to be defined. Thus in Set[String] and Set[Int] the placeholder T has been replaced by String and Int, respectively. The result of applying a type to a Set is a *parameterized* Set.

28.3 Adding Type Parameterisation

28.3.1 The MyQueue Mutable Class

You can create your own types that are generic type and that can be parameterised by concrete type. For example, the mutable collection class MyQueue, presented in the following listing, is a programmer defined generic type. It defines a placeholder T that will be used to represent various concrete types. Note by convention the letter T is used to indicate a type to specify (but we could use any letter or sequence of letters; thus if the types for a key and a value were being specified we might use K and V).

```
package com.jjh.scala.collection

import scala.collection.mutable.ListBuffer

class MyQueue[T] {

  private val content: ListBuffer[T] = new
ListBuffer[T]

  def head: T = content.head

  def size = content.size

  def enqueue(x: T) = content :: x :: Nil

  def dequeue: T = {
    content.remove(0)
  }

  override def toString =
    "[" + content.mkString("| ") + "]"

}
```

The class `MyQueue` uses T as if it were an actual type in the body of the class. Thus the type of element held by the ListBuffer is T, the type of element that can be enqueued using the enqueue method is T and the result returned by the dequeue method is T. The type of the head element is also T.

This class is used in exactly the same way as one of the built-in collection classes as shown below:

```
package com.jjh.scala.collection

object MyQueueTest extends App {
  val q = new MyQueue[String]()
  q.enqueue("John")
  q.enqueue("Denise")
  q.enqueue("Phoebe")
  q.enqueue("Adam")
  println(q.head)
  val name = q.dequeue
  println(name)
  println(q.head)
  q.enqueue("Paul")
  println(q.head)
  println(q)
}
```

In this example, the type String is used to parameterise `MyQueue` such that it now holds Strings. The effect is that the latter T is replaced by the type String throughout the instance of the MyQueue class referenced by the variable 'q'. This effect is that the type of head is String, the type used as a parameter to enqueue is String and the type returned by dequeue is String. Thus it is the same as writing:

```
def head: String = content.head
...
def enqueue(x: String) = content += x

def dequeue: String = {
  content.remove(0)
}
```

However, we can also create a MyQueue of Ints, for example

```
val q2 = new MyQueue[Int]()
```

This is now the equivalent of writing:

```
def head: Int = content.head
...
def enqueue(x: Int) = content += x

def dequeue: Int = {
  content.remove(0)
}
```

As well as String and Int any type can be used as the concrete type including Double, Boolean as well as user-defined types such as the class Person or the Trait Model.

Thus generic class and type parameterisation provide a powerful construct for creating type safe, reusable code.

Note that you can create generic Classes and Traits as both can be instantiated with a concrete type. You cannot create generic Objects as you do not instantiate an Object (this is handled for you by the Scala runtime).

The output from the Queue test program is shown below:

```
[John| Denise| Phoebe| Adam]
John
John
Denise
Denise
[Denise| Phoebe| Adam| Paul]
```

28.3.2 The Immutable Queue Class

We can also create generic types that are immutable and that take parameters into the primary constructor. Note that the types used in the primary constructor can also refer to the type T. Thus the head passed into the Queue is of type T and the tail is a List of type T. Also note that as this is an immutable Queue when you add something to this queue a new copy of the queue is created containing the existing data plus the new element. Of course, the type of the push method is T.

```
package com.jjh.scala.collection

class Queue[T](val head: T, val tail: List[T]) {

  def enqueue(x: T) = new Queue(x, head :: tail)

  def peek = head

  def dequeue = new Queue(tail.head, tail.tail)

  override def toString =
    s"$head: ${(tail mkString ",")}"

        }
```

The Queue class creates a new Queue when a value is added to the Queue, returns a new Queue class (with the current head removed) in response to a dequeue and provides a peek operation to see what is currently at the head of the queue.

To use this class we can specify the concrete type such as String when we create an instance. For example

```
val q =
    new Queue[String]("John", List("Denise", "Phoebe"))
```

This is used in the following program listing to create a Queue which is then processed using *peek* and *dequeue*.

```
object QueueTest extends App {
  val q =
    new Queue[String]("John",
                      List("Denise", "Phoebe"))
  println("q = " + q)
  println("q.peek = " + q.peek)
  val q2 = q.dequeue
  println("q2 = " + q2)
  println("q = " + q)
}
```

Again the result is that the placeholder 'T' has been replaced within the instance of the Queue class by the concrete type String. Thus the type of the head property is String, the type of the tail is List[String] and the type of the parameter 'x' in the enqueue method is String as is the type returned by the dequeue method.

Note that Scala could have inferred the type being used with the myQueue class; thus we could also have written:

```
val q3 = new Queue("John", List("Denise", "Phoebe"))
```

This would also have been a Queue instance parameterised by the type String.

We can actually create a Queue parameterised by Int, Boolean, Double or any user-defined type such as Person. For example

```
case class Person(name: String, age: Int)

object QueueTest2 extends App {
  val q = new Queue[Person](
    Person("John", 54),
    List(new Person("Denise", 51), Person("Phoebe",
21), Person("Adam", 19)))
  println(q)
}
```

28.4 Variance

An important question to consider when looking at generic types is how are subtypes (such as sub classes or sub traits) treated with respect to classes that have been parameterised by a parent type. For example, if we have a Set that can hold Persons, should it be able to hold references to instances of the class Employee if it is a subtype of Person? This subject is referred to as variance and within a type system in a programming language a type rule is

- Covariant if it preserves the ordering of types from more specific ones to more generic ones.
- Contravariant if it reverses the ordering.
- Invariant if neither of these applies.

In Scala, the default for generic types is invariance—that is given a type Person and a subtype Employee MyQueue[Employee] is not a subtype of MyQueue [Employee] and thus it is not possible to assign a reference to a variable of type MyQueue[Employee] to a variable of type MyQueue[Person]. For example, the following listing illustrates an attempt to do just that. However, the compiler will indicate that there is a compilation problem at line 12 when we try and assign s2–s1. This would still be true if we used a built-in type such as ListBuffer or ArrayBuffer.

```
package com.jjh.scala.variance

import com.jjh.scala.collection.MyQueue

class Person

class Employee extends Person

object TestVariance extends App {
  var s1 = new MyQueue[Person]()
  val s2 = new MyQueue[Employee]()
  s1 = s2
}
```

However, you can override this limitation with what are known as *variance annotations*. When we define our generic types we can indicate whether we would like them to be covariant or contravariant using either a '+' or a '−'before the type placeholder. For example,

- Queue[+ T](…){…} indicates covariant where Queue[String] is considered a subtype of Queue[AnyRef]
- Queue[−T](…){…} indicates contravariance where Queue[AnyRef] would be considered a super type of Queue[String].

Using these we can control whether we wish MyQueue[Employee] to be a subtype of MyQueue[Person] or not.

28.5 Lower and Upper Bounds

If you want to limit the types that can be used for T or any supertype of T then you can use a *lower* bound on the type specification. For example

```
def enqueue[U >: T](x: U) =
            new Queue(x, _head :: tail)
```

In this case if we try to enqueue an item then the type of this item must be of type T or a super type of T. Thus if we have a queue of Employees this would allow us to enqueue a Person to the Queue (although the resulting Queue only guarantee that it holds references to Person objects (or subtypes of Person).

If you want to limit the types that can be used with T or any subclass of T then you can use an *upper* bound. For example

```
def enqeue[U <: T](x: U) =
            new Queue(x, _head :: tail)
```

28.6 Combining Variance and Bounds

We can combine variable and bounds together to create flexible containers, for example

```
case class FlexiQueue[+T](head: T, tail: List[T]) {
  def enqueue[U >: T](x: U) =
                  new FlexiQueue(x, head :: tail)

  def peek = head

  def dequeue[U >: T] =
              new FlexiQueue[U](tail.head, tail.tail)
}
```

This class, the FlexiQueue class indicates that the type T will be treated covariantly and that the methods enqueue and dequeue have lower bounds indicating that the type T and its super types may be used with these methods. Thus if we create a FlexiQueue of Employees and then subsequently enqueue a Person this will return a FlexiQueue of Person types. This scenario is represented by the following listing:

```
class Person(val name: String)

class Employee(_name: String) extends Person(_name)

object FlexiQueueTest extends App {
  val q1 = FlexiQueue[Employee](
    new Employee("John"),
    List(new Employee("Denise"),
      new Employee("Phoebe")))
  println(q1)
  val q2 = q1.enqueue(new Person("Adam"))
  println(q2)
}
```

The interesting thing to note is that type inferred by Scala for q1 and q2:

- q1 holds a reference to a FlexiQueue[Employee] type of instance.
- q2 holds a reference to a FlexiQueue[Person] type of instance.

This of course makes sense because when we first created the FlexiQueue we could guarantee that all the elements in the queue were Employees. However once we added a Person to the FlexiQueue, the only thing we could now guarantee is that the contents of the queue were now at least Person instance (although others may be Employee instances).

You can also combine lower or upper bounds with specific types and covariance or contravariance. For example, if you wish to indicate that the type to be used should be a Person or a subtype of Person then you can do so using the Upper bound limit and the type Person with a covariant annotation, for example

```
class Queue[+T <: Person]() {...}
```