5 Introduction to Scala: Type Parameters, Match Expressions, and Case Classes

Drew McDermott `drew.mcdermott@yale.edu`

Lecture notes for `2016-09-14`

Revised `2016-09-16`

3 Type Parameters

Before getting to case classes, let me say a few words about *type parameters*. Scala uses square brackets everywhere to refer to types, in some predictable ways, in some rather hairy ways, which hopefully we can avoid.

If you want an array in Scala, you must specify the types of its elements, e.g., :

```
val nums: Array[Int] = new Array[Int](10)
```

This code declares `nums` to be an array of 10 integers, initially zero. Scala can infer the type, so unless there is some reason to be explicit about it, you can just write

```
val nums = new Array[Int](10)
```

The default array implementation is mutable,[1] so you can alter an element with an assignment statement:

```
nums(9) = nums(8) + 3
```

Parentheses are used around the index, which varies from 0 to the the length of the array -1, as in most programming languages.

The use of types after the symbol `Array` is not a special case. Many constructors, methods, and functions use this syntax:

$$C\texttt{[}T_0, \ldots, T_{k-1}\texttt{]}\texttt{(}A_0, \ldots, A_{n-1}\texttt{)}$$

where C is the constructor, method, or function, the T_i are type parameters or type arguments, and the A_i are ordinary parameter declarations or argument expressions. Examples:

```
new Foo[Int](3)
superFunction[A](3, "whee")
def goodMeth[A](a: Array[A]): A = ...
```

[1] You may find it hard to picture how *immutable* arrays work, but they exist. *Every* data structure can be thought of as immutable, as we will see.

The first example is a call to a constructor that takes one type argument and one regular argument. The second is a call to a function that takes one type argument and two regular ones. (The type variable must be *declared* by someone else; it's just being *used* in the call to `superFunction`.) The third is a definition of a method with one type parameter, `A`, and one regular parameter, an array whose elements are that type. The method returns an object of type `A`.

The second example is a bit misleading in that normally when a function is *called* the type parameters are omitted, because the compiler (in particular, its *type-inference* module) can infer them. When they're omitted, they're omitted completely, square brackets and all. So we'd usually write `superFunction(3, "whee")`, unless there was some reason the type-inference machinery needed help.[2]

Why do we use boring and opaque names such as "`A`" for types? Because it's the convention when the type is somewhat arbitrary, which it usually is. You usually want your method or constructor to be able to operate on or build an object with as few constraints as possible on the types of its pieces. Ideally, that would be any type at all, and not much can be said about just any old type.

Type parameters are usually drawn from the beginning of the alphabet, unless whimsy causes one to switch to `T`, `U`, `V` the same day one puts on that polka-dot tie one hasn't dared to wear to the office before.

3.1 Naming Conventions: Upper or Lower Case?

Sorry for the ambiguity in the word "case"; can't be helped. This subsection is about a less profound issue, which is whether to start an identifier with a lowercase letter or an uppercase one?

Here are the conventions.[3] (These are not enforced by the compiler (usually), but by irate readers of your code if you violate them.)
Names that start with an uppercase letter:

- Type names (including class and trait names)
- Type variables (which usually consist of that one letter)

[2]When it needs help, the compile-time error you get is often hard to decipher. Like any compiler, the Scala compiler takes getting used to. It's hard to work back from where a bug eventually manifests itself to the best way to describe it, and you just have to learn case by case what the most misleading messages really mean. Ask someone more experienced for help, even the instructor if necessary.

[3]Source: `http://docs.scala-lang.org/style/naming-conventions.html`

- Object names
- Constant names (a `val` declared in an object or trait)

All other names start with a lowercase letter, notably the names of:

- Methods
- `var`s
- `val`s declared in a class or method body
- Packages

4 Match Expressions and Case Classes

Now we come to a key construct in any language that's used to write functional programs on recursive data structures: *pattern matching*. In Scala, the syntax looks like this:

e_0 `match` {
  `case` p_1 `=>` e_1
  `case` p_2 `=>` e_2
  ...
  `case` p_n `=>` e_n
}

This is called a *match expression*. The pieces of its right-hand side are called *case clauses*.[4]

Its value is computed by evaluating e_0 and matching it against p_1; if it matches, the value of e_1 is the value of the match expression. Otherwise,

[4] In many programming languages the word `case` gets moved up, so the syntax is more like

`case` e_0 `of` {...}

where the clauses have some syntax or other. Think, "Choose the case e_0 of the following"; but in many languages what follows is a list of elements labeled by numbers or a slight generalization, so if $e_0 = 5$, we're literally picking case 5, the 5th case, the clauses being labeled 1, 2, 3, ..., or something like that. If we label the cases with characters, the wizened programmers can still think of them as disguised numbers, but as the labels get more complicated it becomes impossible to view them as ordinal numbers of any sort. Scala's notation begins to work better. It oddly resembles the C `switch` statement, except that in C each `case` must end with a `break` or endless weird bugs (usually) ensue.

the value of e_0 is matched against p_2, and so forth. If none of the p_i match, a match-error exception is thrown.

So all that remains is to explain what it means for a value to match an expression. The simplest case is where the p_i are all constants; for example, if e_0 is one of several characters and each `case` clause covers one or more of them, one can sometimes compile this into a dense dispatch table with a piecewise-linear hash function, usually making for a fast lookup (though I seriously doubt that the compiler bothers[5]).

The `match` construct can be used to do much more powerful expression analysis than numerical data, even though no efficiency is gained over using `if-then-else` and decomposing expressions using tests and member function. The gain that most programmers are looking for is in brevity, clarity, and debuggability of source code, and pattern matching wins on all these dimensions.

Case classes are the main mechanism programmers use for adding new patterns usable in `case` clauses. The syntax is simple. It's exactly the same as for ordinary classes, except the symbol `case` appears before the symbol `class`. Case-class definitions tend to be placed together in a file when they all `extend` the same trait (or class, but this is rare): Often, the trait is placed just before the case-class definitions:

`trait` T`[`Y`]` { ... }
`case class` C_1`[`Y_1`]`(P_1) `extends` T`[`Γ_1`]` { ... }
`case class` C_2`[`T_2`]`(P_2) `extends` T`[`Γ_2`]`{ ... }
...
`case class` C_N`[`T_N`]`(P_N) `extends` T`[`Γ_N`]` { ... }

The P_i are parameter lists for the constructor, and the Y_i are type-parameter lists. These may and usually do differ drastically from case to case. The Γ_i are type expressions; if trait T has type parameters, then the arguments must be supplied. Sometimes these are just some arrangement of the class's type variables Y_i, but they don't have to be; if `A` is one of the Y_i, then T might have a `List[A]` in there somewhere:

...`extends` Y_i`[..., List[A], ...]`

If a class has no parameters, then you have to decide whether to make it a `case object`; or go ahead and have a class with no parameters (usually

[5]If the programmer really wants this, they're probably a fool; in the rare case where the programmer is not a fool, and this rush to efficiecy is worth the trouble, the programmer can probably code it themselves. Plus, we're targeting the JVM, which can only be optimized so far.

a bad idea).[6] The braces are often empty, especially for the case classes and case objects, but stay tuned.

The schema above is rather forbidding, so a familiarizing example will make it clear what we're talking about. Chapter 3 of FPS describes how to define lists. Here's another example: binary trees with a number stored at all the nodes, a `Double` at each interior node and an `Int` at each leaf node:

```
trait DIBinaryTree { }
case class DILeaf(key: Double, value: Int) extends DIBinaryTree { }
case class DINode(left: DIBinaryTree,
                  num: Double,
                  right: DIBinaryTree) extends DIBinaryTree {
  }
```

Here we have no type parameters. It is much more common to have some. To generalize the previous example, suppose we didn't want to limit the leaf data to `Int`s:

```
trait DBinaryTree[A] { }
case class DLeaf[A](key: Double, value: A) extends DBinaryTree[A] { }
case class DIntNode[A](left: DBinaryTree[A],
                       num: Double,
                       right: DBinaryTree[A]) extends DBinaryTree[A] {
}
```

Our previous data type `DIBinaryTree`, is a special case of this one: `DBinaryTree[Int]`. If we like the name, we can define it:

```
type DIBinaryTree = DBinaryTree[Int]
```

The `case` classes themselves often retain empty bodies indefinitely, but the trait often acquires method definitions. Consider the following example, a method `nearestLeaf`, defined in the trait `DBinaryTree`:

[6] If it has only type parameters, converting it to an object requires imagination. For example, it would make sense to have `Nil` be a class, with one empty list per type. `Nil[Int]` would be a different object from `Nil[Boolean]`. Instead, Scala's designers made the choice that there is only one `Nil`, a list of `Nothing`s. `Nothing` is a subtype of every type Z, because every object of type `Nothing` is also of type Z; because there *are* no objects of type `Nothing`. Ha-ha. Hence `Nil` is an object of `List[Z]`, because `List` is *covariant*: `List[`Z_1`]` is a subtype of `List[`Z_2`]` if Z_1 is a subtype of Z_2. The price you pay for not having to routinely give the type of `Nil` is having to do it occasionally, when you've forgotten all this. *Hint:* If `Nothing` shows up in an error message, try replacing `Nil` with `List[`Z`]()`, where Z is the type Scala couldn't guess should replace `Nothing`.

```
trait DBinaryTree[A] {
  def nearestLeaf(p: Double): A = {
    this match {
      case DLeaf(v) => v
      case DIntNode(l, n, r) => {
       if (p < n) l.nearestLeaf(p)
       else r.nearestLeaf(p)
      }
    }
  }
}
```

I don't know if this does anything useful. Picture the tree being used to sort things. What I want to draw your attention to is that this function is not defined in the usual OOP way. That way involves putting the code for the `DLeaf` objects in the `DLeaf` class definition, and the code for the `DIntNodes` in the `IntNode` definition. In the resulting version, the structure visible above, of a recursive function whose base case is reaching a `DLeaf`, is lost. I call the result a *virtual function*,[7] although as far as I know, there is no standard term for these things.

So, for example, consider the `toString` function, a workhorse of just about any language (not always with that name, of course). Its code is scattered around the classes of a software system, as it should be. Nonetheless, it is often necessary when running `toString` on a complex data structure to include the strings produced for its pieces, which are often of the same type as the data structure we started with. Hence `toString`, considered as a virtual function, is certainly recursive.[8]

[7]The term comes from C++. In that sense, all Java and Scala methods are virtual; look it up. You can define a virtual function in my sense in C++ using virtual methods, but you can do other stuff with C++ virtual methods, too.

[8]One might jump to the conclusion that a function with a distributed description can't be recursive unless it has at least one recursive piece, i.e., a method definition (in a particular class definition) that calls itself, i.e., contains a call to itself in its body. But a function can be recursive without having this property. It suffices for method A to call method B, which calls method A. Of course, it is undecidable whether there is a set of arguments for which a method actually ever *does* call itself.