Table of Contents for Programming Scala, 2nd Edition

Most of the time, Scala's static typing is a virtue. It adds safety constraints that are useful for ensuring correctness at runtime and easier comprehension when browsing code. These benefits are especially useful in large-scale systems.

Occasionally, you might miss the benefits of dynamic typing, however, such as allowing method calls that don't exist at compile time! The popular *Ruby on Rails* web framework uses this technique very effectively in its `ActiveRecord` API. Let's see how we might implement the same technique in Scala.

A Motivating Example: ActiveRecord in Ruby on Rails

`ActiveRecord` is the original object-relational mapping (ORM) library integrated with Rails. Most of the details don't concern us here, but one of the useful features it offers is a DSL for composing queries that consist of chained method calls on a domain object.

However, the "methods" aren't actually defined. Instead, invocations are routed to Ruby's catch-all for undefined methods, `method_missing`. Normally, this method throws an exception, but it can be overridden in classes to do something else. `ActiveRecord` does this to interpret the "missing method" as a directive for constructing a SQL query.

Suppose we have a simple database table of states in the United States (for some dialect of SQL):

```
CREATE TABLE states (
                        -- Name of the
  name        TEXT,     state.
                        -- Name of the capital
  capital     TEXT,     city.
  statehood   INTEGER
-- Year the state was admitted to the
union.
);
```

With `ActiveRecord` you can construct queries as follows, where the Ruby domain object `State` is the analog of the table `states`:

```
# Find all states named
"Alaska"
State.find_by_name("Alaska")
# Find all states named "Alaska" that entered the union in
1959
State.find_by_name_and_statehood("Alaska", 1959)
...
```

For a table with lots of columns, defining all permutations of the `find_by_*` methods would be unworkable. However, the *protocol* defined by the naming convention is easy to automate, so no explicit definitions are required. `ActiveRecord` automates all the boilerplate needed to parse the name, generate the corresponding SQL query, and construct in-memory objects for the results.

Note that `ActiveRecord` implements an *embedded* or *internal* DSL, where the language is an idiomatic dialect of the host language Ruby, rather than an alternative language that requires its own grammar and parser.

Dynamic Invocation in Scala with the Dynamic Trait

It might be useful to implement a similar DSL in Scala, but normally Scala expects all such methods to be defined explicitly. Fortunately, Scala version 2.9 added the `scala.Dynamic` trait to support the dynamic resolution behavior we just described.

The `Dynamic` trait is a marker trait; it has no method definitions. Instead, the compiler sees this trait and follows a protocol for handling uses of it. The protocol is summarized in the trait's Scaladoc page, using the following example for some instance `foo` of a class `Foo` that extends `Dynamic`:

```
foo.method("blah")       ~~> foo.applyDynamic("method")("blah")
foo.method(x = "blah")   ~~> foo.applyDynamicNamed("method")(("x", "blah"))
foo.method(x = 1, 2)     ~~> foo.applyDynamicNamed("method")(("x", 1), ("", 2
))
foo.field                ~~> foo.selectDynamic("field")
foo.varia = 10           ~~> foo.updateDynamic("varia")(10)
foo.arr(10) = 13         ~~> foo.selectDynamic("arr").update(10, 13)
foo.arr(10)              ~~> foo.applyDynamic("arr")(10)
```

`Foo` must implement any of the `*Dynamic*` methods that might be called. The `applyDynamic` method is used for calls that don't use named parameters. If the user names any of the parameters, `applyDynamicNamed` is called. Note that the first argument list has a single argument for the method name invoked. The second argument list has the actual arguments passed to the method.

You can declare these second argument lists to allow a variable number of arguments if you want or you can declare a specific set of typed arguments. It all depends on how you expect users to call the methods.

The methods `selectDynamic` and `updateDynamic` are for reading and writing fields that aren't arrays. The second to last example shows the special form used for writing array elements. For reading array elements, the invocation is indistinguishable from a method call with a single argument. So, for this case, `applyDynamic` has to be used.

Let's create a simple query DSL in Scala using `Dynamic`. Actually, our example is closer to a query DSL in .NET languages called LINQ (language-integrated query). LINQ enables SQL-like queries to be embedded into .NET programs and used with collections, database tables, etc. LINQ is one inspiration for *Slick*, a Scala *functional-relational mapping* (FRM) library.

We'll implement just a few possible operators, so we'll call it CLINQ, for *cheap language-integrated query*. We'll define a case class with that name and yes, it's meant to sound silly.

We'll assume we want to query in-memory data structures, specifically a sequence of maps (key-value pairs) with a SQL-inspired DSL. The implementation is compiled with the code examples, so let's first try the script that both demonstrates the syntax we want and verifies that the implementation works:

```
// src/main/scala/progscala2/dynamic/clinq-example.sc

scala> import progscala2.dynamic.CLINQ
import progscala2.dynamic.CLINQ

scala> def makeMap(name: String, capital: String, statehood: Int) =
     |
  Map("name" -> name, "capital" -> capital, "statehood" -> statehood
)

// "Records" for Five of the states in the
U.S.A.
scala> val states = CLINQ(
         List
     | (
           makeMap("Alaska",     "Juneau",      1959
     | ),
           makeMap("California", "Sacramento",  1850
     | ),
           makeMap("Illinois",   "Springfield", 1818
     | ),
           makeMap("Virginia",   "Richmond",    1788
     | ),
           makeMap("Washington", "Olympia",     1889
     | )))
states: dynamic.CLINQ[Any] =
Map(name -> Alaska, capital -> Juneau, statehood -> 1959)
Map(name -> California, capital -> Sacramento, statehood -> 1850)
Map(name -> Illinois, capital -> Springfield, statehood -> 1818)
Map(name -> Virginia, capital -> Richmond, statehood -> 1788)
Map(name -> Washington, capital -> Olympia, statehood -> 1889)
```

We import the `dynamic.CLINQ` case class that we'll study in a moment. Then we create an instance with a sequence of maps, where each map is a "record."

In contrast to the `ActiveRecord` example, we'll use the `n_and_m` to simply project out the fields we want, like a SQL `SELECT` statement, where `all` will correspond to `*` (some of the output elided):

```
scala> states.name
res0: dynamic.CLINQ[Any] =
Map(name -> Alaska)
Map(name -> California)
Map(name -> Illinois)
Map(name -> Virginia)
Map(name -> Washington)

scala> states.capital
res1: dynamic.CLINQ[Any] =
Map(capital -> Juneau)
Map(capital -> Sacramento)
...

scala> states.statehood
res2: dynamic.CLINQ[Any] =
Map(statehood -> 1959)
Map(statehood -> 1850)
...

scala> states.name_and_capital
res3: dynamic.CLINQ[Any] =
Map(name -> Alaska, capital -> Juneau)
Map(name -> California, capital -> Sacramento)
...

scala> states.name_and_statehood
res4: dynamic.CLINQ[Any] =
Map(name -> Alaska, statehood -> 1959)
Map(name -> California, statehood -> 1850)
...

scala> states.capital_and_statehood
res5: dynamic.CLINQ[Any] =
Map(capital -> Juneau, statehood -> 1959)
Map(capital -> Sacramento, statehood -> 1850)
...

scala> states.all
res6: dynamic.CLINQ[Any] =
Map(name -> Alaska, capital -> Juneau, statehood -> 1959)
Map(name -> California, capital -> Sacramento, statehood -> 1850)
...
```

Finally, how about some `WHERE` clauses?

```
scala> states.all.where("name").NE("Alaska")
res7: dynamic.CLINQ[Any] =
Map(name -> California, capital -> Sacramento, statehood -> 1850)
Map(name -> Illinois, capital -> Springfield, statehood -> 1818)
Map(name -> Virginia, capital -> Richmond, statehood -> 1788)
Map(name -> Washington, capital -> Olympia, statehood -> 1889)

scala> states.all.where("statehood").EQ(1889)
res8: dynamic.CLINQ[Any] =
Map(name -> Washington, capital -> Olympia, statehood -> 1889)

scala> states.name_and_statehood.where("statehood").NE(1850)
res9: dynamic.CLINQ[Any] =
Map(name -> Alaska, statehood -> 1959)
Map(name -> Illinois, statehood -> 1818)
Map(name -> Virginia, statehood -> 1788)
Map(name -> Washington, statehood -> 1889)
```

`CLINQ` knows nothing about the keys in the maps, but the `Dynamic` trait allows us to support methods constructed from them. Here is `CLINQ`:

```
// src/main/scala/progscala2/dynamic/CLINQ.scala
package progscala2.dynamic
import scala.language.dynamics                                    // ❶

case class CLINQ[T](records: Seq[Map[String,T]]) extends Dynamic {

  def selectDynamic(name: String): CLINQ[T] =                     // ❷
    if (name == "all" || records.length == 0) this                // ❸
    else {
      val fields = name.split("_and_")                            // ❹
      val seed = Seq.empty[Map[String,T]]
      val newRecords = (records foldLeft seed) {
        (results, record) =>
          val projection = record filter {                        // ❺
            case (key, value) => fields contains key
          }
          // Drop records with no
          projection.
          if (projection.size > 0) results :+ projection
          else results
      }
      CLINQ(newRecords)                                           // ❻
    }

  def applyDynamic(name: String)(field: String): Where = name match {
    case "where" => new Where(field)                              // ❼
    case _ => throw CLINQ.BadOperation(field, "where.""""Expected")
  }

  protected class Where(field: String) extends Dynamic {          // ❽
    def filter(value: T)(op: (T,T) => Boolean): CLINQ[T] = {      // ❾
```

```
      val newRecords = records filter {
        _ exists {
          case (k, v) => field == k && op(value, v)
        }
      }
      CLINQ(newRecords)
    }

    def applyDynamic(op: String)(value: T): CLINQ[T] = op match {
      case "EQ" => filter(value)(_ == _)                              // ❿
      case "NE" => filter(value)(_ != _)                              // ⓫
                                                 """Expected "EQ" or
      case _ => throw CLINQ.BadOperation(field, "NE".""" )
    }
  }

  override def toString: String = records mkString "\n"              // ⓬
}

object CLINQ {                                                       //
❸
  case class BadOperation(name: String, msg: String) extends RuntimeException(
     "Unrecognized operation $name.
    s$msg"                                )
}
```

❶

`Dynamic` is an optional language feature, so we import it to enable it.

❷

We'll use `selectDynamic` for the *projections* of fields.

❸

Return all the fields for the "keyword" `all` or for no records.

❹

Two or more fields are joined by `_and_`, so split the name into an array of field names.

❺

Filter the maps to return just the named fields.

❻

Construct a new `CLINQ` to return.

❼

Use `applyDynamic` for operators that follow projections. We will only implement `where` for the equivalent of SQL `WHERE` clauses. A new `Where` instance is returned, which also extends `Dynamic`. Note that the *same* set of `records` will be in scope for this instance, so we don't need to construct the new object with them! If another SQL-like keyword is used, it is an error.

❽

The `Where` class used to filter the `records` for particular values of the field named `field`.

❾

A helper method that filters the in-scope `records` for those maps that have a key-value pair with the name specified by `field` and a corresponding value `v` such that `op(value, v)` returns `true`.

❿

If `EQ` is the operator, call `filter` to return only records where the value for the given `field` is equal to the user-specified value.

⓫

Support the not equals case. Note that supporting greater than, less than, etc. would require more careful handling of the types, because not all possible value types support such expressions.

⓬

Create strings for the records that are easier to read.

⓭

Define the `BadOperation` exception in the companion object.

`CLINQ` is definitely "cheap" in several ways. It doesn't implement other useful operations from SQL, like the equivalent of `GROUP BY`. Nor does it implement other `WHERE`-clause operators like greater than, less than, etc. They are actually tricky to support, but not impossible, because not all possible value types support them.

DSL Considerations

The `Dynamic` trait is one of Scala's many tools for implementing *embedded* or *internal* DSLs. We'll explore them in depth in the next chapter. For now, note a few things.

First, the implementation is not easy to understand, which means it's hard to maintain, debug, and extend. It's very tempting to use a "cool" tool like this and live to regret the effort you've taken on. So, use `Dynamic`, as well as any DSL feature, judiciously.

Second, a related challenge that plagues all DSLs is the need to provide meaningful, helpful error messages to users. Try experimenting with the examples we used in the previous section and you'll easily write something the compiler can't parse and the error messages won't be very helpful. (Hint: try using infix notation, where some periods and parentheses are removed.)

Third, a good DSL should prevent the user from writing something that's logically invalid. This simple example doesn't really have that problem, but it becomes a challenge for more advanced DSLs.

Recap and What's Next

We explored Scala's "hook" for writing code with dynamically defined methods and values, which are familiar to users of dynamically typed languages like Ruby. We used it to implement a query DSL that "magically" offered methods based on data values!

However, we also summarized some of the challenges of writing DSLs with features like this. Fortunately, we have many tools at our disposal for writing DSLs, as we'll explore in the next chapter.