**this.type for chaining method calls**

Scala has powerful mechanisms borrowed from other languages, but often one small unique feature is not discussed: **this.type**. In this article I'll explain it shortly and show you a demonstration.  
  
this.type is a singleton type, which means the type represents a certain object, not class of objects. It helps to solve the problem when you need to chain method calls of a class hierarchy, i.e. "value.method1.method2", where method1 is in super class A and method2 is in deriving class B.

class A {def method1: A = this }  
 class B extends A (def method2: B = this}  
 val b = new B

Now we have defined the hierarchy, and created an instance of B named 'b'. Let's see about the chaining of method calls: The following works fine, because method2 returns type B which has method1:

b.method2.method1

However, this won't

b.method1.method2

because method1's result type is A, and so compiler thinks that the object hasn't got the method2.  
  
We could solve this by overriding the method1 in B, which calls the super classes' method1 to do whatever work it does. The result type is covariantly set as B:

class B extends A{  
 override def method1: B = {super.method1; this };  
 ...  
 }

But this is extra work for every subclass of A that needs method chaining!  
  
Here's the other way we can do it in Scala: Using this.type we can tell compiler that the resulting object from method1 is exactly the same as b, so compiler can infer that 'yeah, the resulting object from method1 has method2 too'.

class A { def method1: this.type = this }  
 class B extends A { def method2: this.type = this }  
 // in method2 this.type is not a necessity  
 // unless there's subclass of B.

Now

val b = new B  
 b.method1.method2

works with both alternatives.  
  
For more specific information you can see: [**Scalable Component Abstractions**](http://lamp.epfl.ch/~odersky/papers/ScalableComponent.pdf) which is a very good read otherwise, too.  
  
Scala has also linear mixin composition of which you can read also on the same link. I'm not going into details, but idea is to have partial implementations as traits which we mixin together. We use them in this article, because we want to create a library, but we are not sure how we'd like to extend it later and want to take only the parts we need. Additionally we want the chaining work with all the upcoming traits uniformly, not just with the methods in the superclass. This is achieved with traits and this.type.  
  
The heart is CommandCenter, which queues up given computations, which are just normal Scala functions, of type Unit => Any.

trait CommandCenter{  
 protected var queue = List[Unit => Any]()  
 def <+(computation: => Any): this.type = {  
 queue :::= List[Unit=>Any]({x => computation})  
 this  
 }  
 }

Now we can make objects of CommandCenter that can chain adding of computations, e.g.

val cmd = new CommandCenter{}  
 cmd <+ Console.println("hi") <+ {network.send("bye")}

Certainly we'd like to execute those computations later, but we don't want to pollute the same trait, because it's different responsibility. Here's first a helper class, which just forces the evalution of computations that are in the list.

def executeAll(list: List[Unit => Any])   
 = list foreach {\_()}

Here's the trait for actual execution:

trait ExecuteCommands extends CommandCenter{  
 def execute: Unit = executeAll(queue)  
 }

Now we'd like to extend this thing so that we can make a group of computations and give them names by which we can later ask them to be evaluated.

trait GroupedCommands extends CommandCenter {  
 import scala.collection.mutable.HashMap  
 protected val groups = HashMap[String, List[Unit=>Any]]()  
 def >>(name: String) = {  
 groups += ((name, queue))  
 queue = List()  
 }  
 }

'>>' method ends the construction of computations; it takes all the computations in the queue and names it as a group, and clears the original queue so that new computations can be binded together.  
  
We have also different trait for execution group computations, but we skip that (you can see it in the complete source code).  
  
Now we can create a test case. First we have a construct method that only creates the computations. We restrict ourself from knowing more than grouping build operations i.e. GroupedCommands trait.

def construct(command: GroupedCommands) = {  
 var (x, y) = (1.0, 1.0)  
 def printx = Console.println("x: " + x)  
 command <+ printx <+ { x \*= 2 } <+ printx >> "doubleX"  
 command <+ printx <+ { x /= 2 } <+ printx >> "halfX"  
 }

Of course <+ is kinda useless, we could just make a one big computation with {comp1; comp2; ... compN}, but this is only an example, though by partioning them to be differentiable, one could use for example time delay between evaluations. Here's the execute part:

def execute(command: GroupExecution) = {  
 command execute("doubleX")  
 command execute("doubleX")  
 command execute("halfX")  
 }

And finally the test that gives observable results:

val command = new CommandCenter   
 with GroupedCommands with GroupExecution  
 construct(command)  
 execute(command)

It prints:

x: 1.0, x: 2.0, x: 2.0, 4.0, x: 4.0, x: 2.0

Here's the complete [**source code**](http://users.utu.fi/hvkhut/scalada/thistype.scala).  
  
So now you know of this.type. Spread the word, for rarely anyone mentions it.