Programming in Scala

PrePrint™ Edition
Excerpt

artima
ARTIMA PRESS
MOUNTAIN VIEW, CALIFORNIA

Thank you for downloading this sample chapter from the PrePrint™ Edition, Version 2, of *Programming in Scala*.

A PrePrint™ is a work-in-progress, a book that has not yet been fully written, reviewed, edited, or formatted. We are publishing this book as a PrePrint™ for two main reasons. First, even though this book is not quite finished, the information contained in its pages can already provide value to many readers. Second, we hope to get reports of errata and suggestions for improvement from those readers while we still have time incorporate them into the first printing.

As a PrePrint™ customer, you'll be able to download new PrePrint™ versions from Artima as the book evolves, as well as the final PDF of the book once finished. You'll have access to the book's content prior to its print publication, and can participate in its creation by submitting feedback. If you purchase the PrePrint™, you'll be able to submit feedback by clicking on the *Suggest* link at the bottom of each page. (In this excerpt, however, you'll find no *Suggest* links.)

Thanks. We hope you find this sample chapter useful and enjoyable.

Bill Venners
President, Artima, Inc.

Chapter 18

Abstract Members and Properties

A member of a class or trait is *abstract* if the member does not have a complete definition in the class. Abstract members are supposed to be implemented in subclasses of the class in which they are defined. This idea is found in many object-oriented languages. For instance, Java lets you declare abstract methods. Scala also lets you declare such methods, as you have seen in chapter **??**. But it goes beyond that and implements the idea in its full generality—besides methods you can also define abstract fields and even abstract types.

An example is the following trait `Abstract` which defines an abstract type `T`, an abstract method `transform`, an abstract value `initial`, and an abstract variable `current`.

```
trait Abstract {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}
```

A concrete implementation of `Abs` needs to fill in definitions for each of its abstract members. Here is an example implementation that provides these definitions.

```
class Concrete extends Abstract {
  type T = String
```

```
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

The implementation gives a concrete meaning to the type name `T` by defining it as an alias of type `String`. The `transform` operation concatenates a given string with itself, and the `initial` and `current` values are both set to `"hi"`.

This example gives you a rough first idea of what kinds of abstract members exist in Scala. The next sections will present the details and explain what the new forms of abstract members are good for.

18.1 Abstract `val`s

An abstract `val` definition has a form like

```
val initial: String
```

It gives a name and type for a `val`, but not its value. This value has to be provided by a concrete value definition in a subclass. For instance, class `Concrete` implemented the value using

```
val initial = "hi"
```

You use an abstract value definition in a class when you do not know the correct value in the class, but you do know that the variable will have an unchangeable value in each instance of the class.

An abstract value definition resembles an abstract parameterless method definition such as

```
def initial: String
```

Client code refers to both the value and the method in exactly the same way, i.e. `obj.initial`. However, if `initial` is an abstract value, the client is guaranteed that `obj.initial` will yield the same value everytime it is referenced. If `initial` is an abstract method, that guarantee would not hold, because in that case `initial` could be implemented by a concrete method that returned a different value everytime it was called.

In other words, an abstract value constrains its legal implementation: Any implementation must be a `val` definition; it may not be a `var` or `def` definition. Abstract method definitions, on the other hand, may be implemented by both concrete method definitions and concrete value definitions. So given a class,

```
abstract class A {
  val v: String   // ‘v' for value
  def m: String   // ‘m' for method
}
```

the following class would be a legal implementation

```
class C1 extends A {
  val v: String
  val m: String // OK to override a ‘def' with a ‘val'
}
```

But the following class would be in error:

```
class C2 extends A {
  def v: String // ERROR: cannot override a ‘val' with a ‘def'
  def m: String
}
```

18.2 Abstract vars

Like an abstract `val`, an abstract `var` defines just a name and a type, but not an initial value. For instance, here is a class `AbstractTime` which defines two abstract variables named `hour` and `minute`.

```
trait AbstractTime {
  var hour: Int
  var minute: Int
}
```

What should be the meaning of an abstract `var` like `hour` or `minute`? You have seen in Chapter 17 that `var`s that are members of classes come equipped

with getter and setter methods. This holds for concrete as well as abstract variables. If you define an abstract `var x`, you implicitly define a getter method `x` and a setter method `x_=`. In fact, an abstract `var` is just a shorthand for a pair of getter and setter methods. There's no reassignable field to be defined – that will come in subclasses which define the concrete implementation of the abstract `var`. For instance, the definition of `AbstractTime` above is exactly equivalent to the following definition.

```
trait AbstractTime {
  def hour: Int          // getter for 'hour'
  def hour_=(x: Int)     // setter for 'hour'
  def minute: Int        // getter for 'minute'
  def minute_=(x: Int)   // setter for 'minute'
}
```

18.3 Abstract types

In the beginning of this chapter you saw an abstract type declaration

```
type T
```

The rest of this chapter discusses what such an abstract type declaration means and what it's good for. Like all other abstract declarations this is a placeholder for something that will be defined concretely in subclasses. In this case, it is a type that will be defined further down the class hierarchy. So `T` above refers to a type that is at yet unknown at the point where it is defined. Different subclasses can provide different realizations of `T`.

Here is a well-known example where abstract types show up naturally. Suppose you are given the task to model eating habits of animals. You might start with a class `Food` and a class `Animal` with an `eat` method:

```
class Food {}
abstract class Animal {
  def eat(food: Food)
}
```

You would then specialize these two classes to a class of `Cows` which eat `Grass`:

```
class Grass extends Food {}
class Cow extends Animal {
  override def eat(food: Grass) {}
}
```

However, if you tried to compile the new classes you'd get compilation errors:

```
error: class Cow needs to be abstract, since method eat is not
defined
class Cow {
      ^

error: method eat overrides nothing
  override def eat(food: Grass)
               ^
```

What happened is that the `eat` method in class `Cow` does not override the `eat` method in class `Animal` because its parameter type is different—it's `Grass` in class `Cow` vs. `Food` in class `Animal`.

Some people have argued that the type system is unnecessarily strict in refusing these classes. They have said that it should be OK to specialize a parameter of a method in a subclass. However, if the classes were allowed as written, you could get yourself in unsafe situations very quickly. For instance, the following would pass the type checker.

```
class Fish extends Food
val cow: Animal = new Cow
cow eat (new Fish)
```

The program would compile, because `Cows` are `Animals` and `Animals` do have an `eat` method that accepts any kind of `Food`, including `Fish`. But surely it would do a cow no good to eat a fish!

What you need to do instead is apply some more precise modelling. `Animals` do eat `Food` but what kind of `Food` depends on the `Animal`. This can be neatly expressed with an abstract type:

```
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
```

```
}
```

With the new class definition, an Animal can eat only food that's suitable. What food is suitable cannot be determined on the level of the `Animal` class. That's why `SuitableFood` is modelled as an abstract type. The type has an upper bound `Food`, which is expressed by the `<: Food` clause. This means that any concrete instantiation of `SuitableFood` in a subclass must be a subclass of `Food`. For example, you would not be able to instantiate `SuitableFood` with class `IOException`.

With `Animal` defined, you can now progress to cows:

```
class Cow extends Animal {
  type SuitableFood = Grass
  def eat(food: Grass) {}
}
```

Class `Cow` fixes its `SuitableFood` to be `Grass` and also defines a concrete `eat` method for this kind of food. These new class definitions compile without errors. If you tried to run the "cows-that-eat-fish" counterexample with the new class definitions you'd get the following:

```
scala> class Fish extends Food
defined class Fish
scala> val cow: Animal = new Cow
cow: Animal = Cow@1fb069

scala> cow eat (new Fish)
<console>:7: error: type mismatch;
 found   : Fish
 required: cow.SuitableFood
       cow eat (new Fish)
               ^
```

Path-dependent types

Have a look at the last error message: What's interesting about it is the type required by the `eat` method: `cow.SuitableFood`. This type consists of an object reference (`cow`) which is followed by a type field `SuitableFood` of

the object. So this shows that objects in Scala can have types as members. `cow.SuitableFood` means "the type `SuitableFood` which is a member of the `cow` object," or otherwise said, the type of food that's suitable for `cow`. A type like `cow.SuitableFood` is called a *path dependent type*. The word "path" here means a reference to an object. It could be a single name or a longer access path such as in `swiss.cow.SuitableFood`.

As the term "path-dependent type" says, the type depends on the path: in general, different paths give rise to different types. For instance, if you had two animals `cat` and `dog`, their `SuitableFood`s would not be the same: `cat.SuitableFood` is incompatible with `dog.SuitableFood`. The case is different for `Cow`s however. Because their `SuitableFood` type is defined to be an alias for class `Grass`, the `SuitableFood` types of two cows are in fact the same.

A path-dependent type resembles a reference to an inner class in Java, but there is a crucial difference: A path-dependent type names an outer *object*, whereas a reference to an inner class names an outer *class*. References to inner classes as in Java can also be expressed in Scala, but they are written differently. Assume two nested classes `Outer` and `Inner`:

```
class Outer {
  class Inner
}
```

In Scala, the inner class is addressed using the expression `Outer # Inner` instead of `Outer.Inner` in Java. The `.` syntax is reserved for objects. For instance, assume two objects:

```
val o1, o2 = new Outer
```

Then `o1.Inner` and `o2.Inner` would be two path-dependent types (and they would be different types). Both of these types would conform to the more general type `Outer # Inner` which represents the `Inner` class with an arbitrary outer object of type `Outer`.

18.4 Case study: Currencies

The rest of this chapter presents a case-study which explains how abstract types can be used in Scala. The task is to design a class `Currency`. A

typical instance of `Currency` would represent an amount of money in Dollars or Euros or Yen, or in some other currency. It should be possible to do some arithmetic on currencies. For instance, you should be able to add two amounts of the same currency. Or you should be able to multiply currency amount by a factor representing an interest rate.

These thoughts lead to the following first design for a currency class:

```
// A first (faulty) design of the Currency class
abstract class Currency {
  val amount: Long
  def designation: String
  override def toString = amount+" "+designation
  def +(that: Currency): Currency = ...
  def *(x: Double): Currency = ...
}
```

The `amount` of a currency is the number of currency units it represents. This is a field of type `Long` so that very large amounts of money such as the market capitalization of Google or Microsoft can be represented. It's left abstract here, waiting to be defined when one talks about concrete amounts of money. The `designation` of a currency is a string that identifies it. The `toString` method of class `Currency` indicates an amount and a designation. It would yield results such as

```
79 USD
11000 Yen
99 Euro
```

Finally, there are methods '`+`' for adding currencies and '`*`' for multiplying a currency with a a floating point number. A concrete currency value could be created by instantiating `amount` and `designation` with concrete values. For instance:

```
new Currency {
  val amount = 99.95
  def designation = "USD"
}
```

This design would be OK if all we wanted to model was a single currency such as only Dollars or only Euros. But it fails once we need to deal with several currencies. Assume you model Dollars and Euros as two subclasses of class currency.

```
abstract class Dollar extends Currency {
  def designation = "USD"
}
abstract class Euro extends Currency {
  def designation = "Euro"
}
```

At first glance this looks reasonable. But it would let you add Dollars to Euros. The result of such an addition would be of type `Currency`. But it would be a funny currency that was made up of a mix of Euros and Dollars. What you want instead is a more specialized version of '+': When implemented in class `Dollar`, it should take `Dollar` arguments and yield a `Dollar` amount; when implemented in class `Euro`, it should take `Euro` arguments and yield `Euro` results. So the type of the addition method changes depending in which class you are in. However, you would like to write the addition method just once, not each time a new currency is defined.

In Scala, there's a simple technique to deal with situations like this: If something is not known at the point where a class is defined, make it abstract in the class. This applies to both values and types. In the case of currencies, the exact argument and result type of the addition method are not known, so it is a good candidate for an abstract type. This would lead to the following sketch of class `AbstractCurrency`:

```
// A second (still imperfect) design of the Currency class
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency
  val amount: Long
  def designation: String
  override def toString = amount+" "+designation
  def +(that: Currency): Currency = ...
  def *(x: Double): Currency = ...
}
```

The only difference to the situation before is that the class is now called `AbstractCurrency`, and that it contains an abstract type `Currency`, which represents the real currency in question. A concrete subclass of `AbstractCurrency` would need to fix the `Currency` type to refer to the concrete subclass itself, thereby "tying the knot". For instance, here is a new version of `Dollar` which extends `AbstractCurrency`.

```
class Dollar extends AbstractCurrency {
  type Currency = Dollar
  def designation = "USD"
}
```

This design is workable, but it is still not perfect. One problem is hidden by triple dots which indicate the missing method definitions of '`+`' and '`*`'in class `AbstractCurrency`. In particular, how should addition be implemented in this class? It's easy enough to calculate the correct amount of the new currency as `this.amount + that.amount`, but how to convert the amount into a currency of the right type? You might try something like

```
def +(that: Currency): Currency = new Currency {
  val amount = this.amount + that.amount
}
```

However, this would not compile:

```
error: class type required
  def +(that: Currency): Currency = new Currency {
                                        ^
```

One of the restrictions of Scala's treatment of abstract types is that you cannot create an instance of an abstract type, or have an abstract type as a supertype of another class.[1] So the compiler refused the example code above.

However, you can work around this restriction using a *factory method*. Instead of creating an instance of an abstract type directly, define an abstract method which does it. Then, wherever the abstract type is fixed to be some concrete type, you also need to give a concrete implementation of the factory method. For class `AbstractCurrency`, this would look as follows:

[1] There's some promising recent research on *virtual classes*, which would allow this, but this is not currently supported in Scala.

```
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency // abstract type
  def make(amount: Long): Currency  // factory method
  ...                               // rest of class
}
```

A design like this could be made to work, but it looks rather suspicious. Why place the factory method *inside* class `AbstractCurrency`? This looks like a code smell, for at least two reasons. First, if you have some amount of currency (say: one Dollar), you also hold in your hand the ability to make more of the same currency, using code such as:

```
myDollar.make(100)  // here are a hundred more!
```

In the age of color copying this might be a tempting scenario, but hopefully not one which you would be able to do for very long without being caught. The second problem with this code is that you can make more `Currency` objects if you already have a reference to a `Currency` object, but how do you get the first object of a given `Currency`? You'd need another creation method, which does essentially the same job as `make`. So you have a case of code duplication, which is a sure sign of a code smell.

The solution, of course, is to move the abstract type and the factory method outside class `AbstractCurrency`. You need to create another class which contains the `AbstractCurrency` class, the `Currency` type, and the `make` factory method. Let's call this object a `CurrencyZone`:

```
abstract class CurrencyZone {
  type Currency <: AbstractCurrency
  def make(x: Long): Currency
  abstract class AbstractCurrency {
    val amount: Long
    def designation: String
    override def toString = amount+" "+designation
    def +(that: Currency): Currency =
      make(this.amount + that.amount)
    def *(x: Double): Currency =
      make((this.amount * x).toLong)
  }
```

```
}
```

An example of a concrete `CurrencyZone` is the US. You could define this as follows:

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(x: Long) = new Dollar { val amount = x }
}
```

Here, `US` is an object that extends `CurrencyZone`. It defines a class `Dollar` which is a subclass of `AbstractCurrency`. So the type of money in this zone is `US.Dollar`. The `US` object also fixes the type `Currency` to be an alias for `Dollar`, and it gives an implementation of the `make` factory method to return a dollar amount.

This is a workable design. There are only some refinements to be added. The first refinement concerns subunits. So far, every currency was measured in a single unit: Dollars, Euros, or Yen. However, most currencies have subunits: for instance, in the US, it's dollars and cents. The most straightforward way to model cents is to have the `amount` field in `US.Currency` represent cents instead of Dollars. To convert back to Dollars, it's useful to introduce a field `CurrencyUnit` in class `CurrencyZone` which contains the amount of currency of one standard unit in that currency. Class `CurrencyZone` gets thus augmented like this:

```
class CurrencyZone {
  ...
  val CurrencyUnit: Currency
}
```

The `US` object then defines the quantities `Cent`, `Dollar`, and `CurrencyUnit` as follows:

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency
  type Currency = Dollar
```

```
  def make(x: Long) = new Dollar {
    val amount = x
    def designation = "USD"
  }
  val Cent = make(1)
  val Dollar = make(100)
  val CurrencyUnit = Dollar
}
```

This definition is just like the previous definition of the `US` object, except that it adds three new fields: The field `Cent` represents an amount of 1 `US.Currency`. It's an object analogous to a copper coin of one cent. The field `Dollar` represents an amount of 100 `US.Currency`. So the `US` object now defines the name `Dollar` in two ways: The *type* Dollar represents the generic name of the `Currency` valid in the `US` currency zone. By contrast the *variable* Dollar represents a single US Dollar, analogous to a greenback bill. The third field definition of `CurrencyUnit` specifies that the standard currency unit in the `US` zone is the `Dollar`.

The `toString` method in class `Currency` also needs to be adapted to take subunits into account. For instance, the sum of ten dollars and twenty three cents should print as a decimal number: 10.23 USD. To achieve this, you implement `Currency`s `toString` method as follows:

```
override def toString =
  amount format "%."+decimals(CurrencyUnit.amount)+"f"
```

Here, `format` is a method which Scala adds to the standard `String` class. It returns a string which is formatted according to a format string which is given as the method's right-hand operand. Format strings are as for Java's `String.format` method. For instance, the format string `%.2f` formats a number with two decimal digits. The format string above is assembled by calling the `decimals` method on `CurrencyUnit.amount`. This method returns the number of decimal digits of a decimal power minus one. For instance, `decimals(10)` is `1`, `decimals(100)` is `2`, and so on. The `decimals` method is implemented by a simple recursion:

```
private def decimals(l: Long): Int =
  if (l == 1) 0 else 1 + decimals(l / 10)
```

Here are some other currency zones:

```
  object European extends CurrencyZone {
    abstract class Euro extends AbstractCurrency
    type Currency = Euro
    def make(x: Long) = new Euro {
      val amount = x
    }
    def designation = "EUR"
  }
  val Euro = make(100)
  val Cent = make(1)
  val CurrencyUnit = Euro
}
object Japan extends CurrencyZone {
  abstract class Yen extends AbstractCurrency
  type Currency = Yen
  def make(x: Long) = new Yen {
    val amount = x
    def designation = "JPY"
  }
  val Yen = make(1)
  val CurrencyUnit = Yen
}
```

As another refinement you can add a currency conversion feature to the model. As a first step, write a `Converter` object which contains applicable exchange rates between currencies. For instance:

```
  object Converter {
    var exchangeRate = Map(
      "USD" -> Map("USD" -> 1.0   , "EUR" -> 0.7596,
                   "JPY" -> 1.211 , "CHF" -> 1.223),
      "EUR" -> Map("USD" -> 1.316 , "EUR" -> 1.0   ,
                   "JPY" -> 1.594 , "CHF" -> 1.623),
      "JPY" -> Map("USD" -> 0.8257, "EUR" -> 0.6272,
                   "JPY" -> 1.0   , "CHF" -> 1.018),
      "CHF" -> Map("USD" -> 0.8108, "EUR" -> 0.6160,
```

```
                    "JPY" -> 0.982 , "CHF" -> 1.0  )
  )
}
```

Then, add a conversion method `from` to class `Currency`, which converts from a given source currency into the current `Currency` object:

```
def from(other: CurrencyZone#AbstractCurrency): Currency =
  make(Math.round(other.amount.toDouble * Converter.exchangeRate
    (other.designation)(this.designation)))
```

The `from` method takes another completely arbitrary currency as argument. That's expressed by its formal parameter type is `CurrencyZone#AbstractCurrency`, which stands for an `AbstractCurrency` type in some arbitrary and unknown `CurrencyZone`. It produces its result by multiplying the amount of the `other` currency with the exchange rate between the other and the current currency.

Here's the code of the final version of the `CurrencyZone` class:

```
abstract class CurrencyZone {
  type Currency <: AbstractCurrency
  protected def make(x: Long): Currency
  val CurrencyUnit: Currency
  private def decimals(l: Long): Long =
    if (l == 1) 0 else 1 + decimals(l / 10)
  abstract class AbstractCurrency {
    val amount: Long
    def designation: String
    def +(that: Currency): Currency =
      make(this.amount + that.amount)
    def -(that: Currency): Currency =
      make(this.amount - that.amount)
    def *(that: Double) =
      make((this.amount * that).toLong)
    def /(that: Double) =
      make((this.amount / that).toLong)
    def /(that: Currency) =
      this.amount.toDouble / that.amount
```

```
    override def toString =
      (amount.toDouble / CurrencyUnit.amount.toDouble) format
        "%."+decimals(CurrencyUnit.amount)+"f"
    def from(other: CurrencyZone#AbstractCurrency): Currency =
      make(Math.round(
        other.amount.toDouble *
        Converter.exchangeRate(other.designation)(this.designation)))
  }
}
```

You can test the class in the Scala command shell. Let's assume that the `CurrencyZone` class and all concrete `CurrencyZone` objects are defined in a package `currencies`. The first step is to import everything in this package into the command shell:

```
scala> import currencies._
```

We can then do some currency conversions:

```
scala> Japan.Yen from US.Dollar * 100
res0: currencies.Japan.Currency = 12110

scala> European.Euro from res0
res1: currencies.European.Currency = 75.95

scala> US.Dollar from res1
res2: currencies.US.Currency = 99.95
```

The fact that we obtain almost the same amount after three conversions implies that these are some pretty good exchange rates!

You can also add up values of the same currency:

```
scala> US.Dollar * 100 + res2
res4: currencies.US.Currency = 199.95
```

On the other hand, you cannot add amounts of different currencies:

```
scala> US.Dollar + European.Euro
<console>:7: error: type mismatch;
 found   : currencies.European.Euro
 required: currencies.US.Currency
```

```
US.Dollar + European.Euro
                          ^
```

So the type abstraction has done its job—it prevents us from performing calculations which are unsound. Failures to convert correctly between different units may seem like trivial bugs, but they have caused many serious systems faults. An example is the crash of the Mars Climate Orbiter spacecraft on Sep 23rd, 1999, which was caused because one engineering team used metric units while another used English units. If units had been coded in the same way as `Currency`s are coded in this chapter, this error would have been detected by a simple compilation run. Instead, it caused the crash of the orbiter after a near 10-month voyage.

Conclusion

Scala offers systematic and very general support for object-oriented abstraction: It enables you to not only abstract over methods, but also to abstract over values, variables, and types. This chapter has shown how to make use of abstract members. You have seen that they support a simple yet effective principle for systems structuring: When designing a class make everything which is not yet known at the level of a class into an abstract member. This principle applies to all sorts of members: methods and variables as well as types.

The chapter has also shown how variables which are members of some class come equipped with setters and getters. You have seen how these getters and setters can be used to implement properties.