Chapter 20

Abstract Members

A member of a class or trait is abstract if the member does not have a complete definition in the class. Abstract members are intended to be implemented in subclasses of the class in which they are declared. 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 saw in Section 10.2. But Scala goes beyond that and implements the idea in its full generality: Besides methods, you can declare abstract fields and even abstract types as members of classes and traits.

In this chapter we'll describe all four kinds of abstract member: vals, vars, methods, and types. Along the way we'll discuss pre-initialized fields, lazy vals, path-dependent types, and enumerations.

20.1 A QUICK TOUR OF ABSTRACT MEMBERS

The following trait declares one of each kind of abstract member: an abstract type (T), method (transform), val (initial), and var (current):

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

A concrete implementation of Abstract 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 andcurrent values are both set to "hi".

This example gives you a rough first idea of what kinds of abstract members exist in Scala. The remainder of the chapter will present the details and explain what the new forms of abstract members, as well as type members in general, are good for.

20.2 TYPE MEMBERS

As you can see from the example in the previous section, the term *abstract type* in Scala means a type declared (with the "type" keyword) to be a member of a class or trait, without specifying a

definition. Classes themselves may be abstract, and traits are by definition abstract, but neither of these are what are referred to as *abstract types* in Scala. An abstract type in Scala is always a member of some class or trait, such as type T in trait Abstract.

You can think of a non-abstract (or "concrete") type member, such as type T in class Concrete, as a way to define a new name, or *alias*, for a type. In class Concrete, for example, the typeString is given the alias T. As a result, anywhere T appears in the definition of class Concrete, it means String. This includes the parameter and result types of transform, initial, and current, which mention T when they are declared in supertrait Abstract. Thus, when class Concreteimplements these methods, those Ts are interpreted to mean String.

One reason to use a type member is to define a short, descriptive alias for a type whose real name is more verbose, or less obvious in meaning, than the alias. Such type members can help clarify the code of a class or trait. The other main use of type members is to declare abstract types that must be defined in subclasses. This use, which was demonstrated in the previous section, will be described in detail later in this chapter.

20.3 ABSTRACT VALS

An abstract val declaration 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 val definition in a subclass. For instance, class Concrete implemented the val using:

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

You use an abstract val declaration 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 val declaration resembles an abstract parameterless method declaration such as:

```
def initial: String
```

Client code would refer to both the val and the method in exactly the same way (*i.e.*,obj.initial). However, if initial is an abstract val, the client is guaranteed that obj.initial will yield the same value every time it is referenced. If initial were an abstract method, that guarantee would not hold because, in that case, initial could be implemented by a concrete method that returns a different value every time it's called.

In other words, an abstract val constrains its legal implementation: Any implementation must be a val definition; it may not be a var or a def. Abstract method declarations, on the other hand, may be implemented by both concrete method definitions and concrete valdefinitions. Given the abstract class Fruit shown in Listing 20.1, class Apple would be a legal subclass implementation, but class BadApple would not.

```
abstract class Fruit {
```

```
  val v: String // `v' for value
  def m: String // `m' for method
}

abstract class Apple extends Fruit {
  val v: String
  val m: String // OK to override a `def' with a `val'
}

abstract class BadApple extends Fruit {
  def v: String // ERROR: cannot override a `val' with a `def'
  def m: String
}
```

Listing 20.1 - Overriding abstract vals and parameterless methods.

20.4 ABSTRACT VARS

Like an abstract val, an abstract var declares just a name and a type, but not an initial value. For instance, Listing 20.2 shows a trait AbstractTime, which declares two abstract variables named hour and minute:

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

Listing 20.2 - Declaring abstract vars.

What is the meaning of abstract vars like hour and minute? You saw in Section 18.2 that vars declared as members of classes come equipped with getter and setter methods. This holds for abstract vars as well. If you declare an abstract var named hour, for example, you implicitly declare an abstract getter method, hour, and an abstract setter method, hour_=. There's no reassignable field to be defined—that will come in subclasses that define the concrete implementation of the abstract var. For instance, the definition of AbstractTime shown inListing 20.2 is exactly equivalent to the definition shown in Listing 20.3.

```
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'
}
```

Listing 20.3 - How abstract vars are expanded into getters and setters.

20.5 INITIALIZING ABSTRACT VALS

Abstract vals sometimes play a role analogous to superclass parameters: they let you provide details in a subclass that are missing in a superclass. This is particularly important for traits, because traits don't have a constructor to which you could pass parameters. So the usual notion of parameterizing a trait works via abstract vals that are implemented in subclasses.

As an example, consider a reformulation of class Rational from Chapter 6, as shown in Listing 6.5 here, as a trait:

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
}
```

The Rational class from Chapter 6 had two parameters: n for the numerator of the rational number, and d for the denominator. The RationalTrait trait given here defines instead two abstract vals: numerArg and denomArg. To instantiate a concrete instance of that trait, you need to implement the abstract val definitions. Here's an example:

```
new RationalTrait {
  val numerArg = 1
  val denomArg = 2
}
```

Here the keyword new appears in front of a trait name, RationalTrait, which is followed by a class body in curly braces. This expression yields an instance of an *anonymous class* that mixes in the trait and is defined by the body. This particular anonymous class instantiation has an effect analogous to the instance creation new Rational(1, 2).

The analogy is not perfect, however. There's a subtle difference concerning the order in which expressions are initialized. When you write:

```
new Rational(expr1, expr2)
```

the two expressions, expr1 and expr2, are evaluated before class Rational is initialized, so the values of expr1 and expr2 are available for the initialization of class Rational.

For traits, the situation is the opposite. When you write:

```
new RationalTrait {
  val numerArg = expr1
  val denomArg = expr2
}
```

the expressions, expr1 and expr2, are evaluated as part of the initialization of the anonymous class, but the anonymous class is initialized *after* the RationalTrait. So the values of numerArgand denomArg are not available during the initialization of RationalTrait (more precisely, a selection of either value would yield the default value for type Int, 0). For the definition ofRationalTrait given previously, this is not a problem, because the trait's initialization does not make use of values numerArg or denomArg. However, it becomes a problem in the variant ofRationalTrait shown in Listing 20.4, which defines normalized numerators and denominators.

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
  require(denomArg != 0)
  private val g = gcd(numerArg, denomArg)
```

```
  val numer = numerArg / g
  val denom = denomArg / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def toString = numer + "/" + denom
}
```

Listing 20.4 - A trait that uses its abstract vals.

If you try to instantiate this trait with some numerator and denominator expressions that are not simple literals, you'll get an exception:

```
scala> val x = 2
x: Int = 2

scala> new RationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:207)
  at RationalTrait$class.$init$(<console>:10)
  ... 28 elided
```

The exception in this example was thrown because denomArg still had its default value of 0 when class RationalTrait was initialized, which caused the require invocation to fail.

This example demonstrates that initialization order is not the same for class parameters and abstract fields. A class parameter argument is evaluated before it is passed to the class constructor (unless the parameter is by-name). An implementing val definition in a subclass, by contrast, is evaluated only after the superclass has been initialized.

Now that you understand why abstract vals behave differently from parameters, it would be good to know what can be done about this. Is it possible to define a RationalTrait that can be initialized robustly, without fearing errors due to uninitialized fields? In fact, Scala offers two alternative solutions to this problem, pre-initialized fields and lazy vals. They are presented in the remainder of this section.

Pre-initialized fields

The first solution, pre-initialized fields, lets you initialize a field of a subclass before the superclass is called. To do this, simply place the field definition in braces before the superclass constructor call. As an example, Listing 20.5 shows another attempt to create an instance of RationalTrait. As you see from this example, the initialization section comes before the mention of the supertrait RationalTrait. Both are separated by a with.

```
scala> new {
         val numerArg = 1 * x
         val denomArg = 2 * x
       } with RationalTrait
res1: RationalTrait = 1/2
```

Listing 20.5 - Pre-initialized fields in an anonymous class expression.

Pre-initialized fields are not restricted to anonymous classes; they can also be used in objects or named subclasses. Two examples are shown in Listings 20.6 and 20.7. As you can see from these examples, the pre-initialization section comes in each case after the extends keyword of the defined object or class. Class RationalClass, shown in Listing 20.7, exemplifies a general schema of how class parameters can be made available for the initialization of a supertrait.

```
object twoThirds extends {
  val numerArg = 2
  val denomArg = 3
} with RationalTrait
```

Listing 20.6 - Pre-initialized fields in an object definition.

```
class RationalClass(n: Int, d: Int) extends {
  val numerArg = n
  val denomArg = d
} with RationalTrait {
  def + (that: RationalClass) = new RationalClass(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
}
```

Listing 20.7 - Pre-initialized fields in a class definition.

Because pre-initialized fields are initialized before the superclass constructor is called, their initializers cannot refer to the object that's being constructed. Consequently, if such an initializer refers to this, the reference goes to the object containing the class or object that's being constructed, not the constructed object itself.

Here's an example:

```
scala> new {
       val numerArg = 1
       val denomArg = this.numerArg * 2
     } with RationalTrait
<console>:11: error: value numerArg is not a member of object
 $iw
               val denomArg = this.numerArg * 2
                                   ^
```

The example did not compile because the reference this.numerArg was looking for a numerArgfield in the object containing the new (which in this case was the synthetic object named $iw, into which the interpreter puts user input lines). Once more, pre-initialized fields behave in this respect like class constructor arguments.

Lazy vals

You can use pre-initialized fields to simulate precisely the initialization behavior of class constructor arguments. Sometimes, however, you might prefer to let the system itself sort out how things should be initialized. This can be achieved by making your val definitions *lazy*. If you prefix a val definition with

a lazy modifier, the initializing expression on the right-hand side will only be evaluated the first time the val is used.

For an example, define an object Demo with a val as follows:

```
scala> object Demo {
         val x = { println("initializing x"); "done" }
       }
defined object Demo
```

Now, first refer to Demo, then to Demo.x:

```
scala> Demo
initializing x
res3: Demo.type = Demo$@2129a843

scala> Demo.x
res4: String = done
```

As you can see, the moment you use Demo, its x field becomes initialized. The initialization of xforms part of the initialization of Demo. The situation changes, however, if you define the x field to be lazy:

```
scala> object Demo {
         lazy val x = { println("initializing x"); "done" }
       }
defined object Demo

scala> Demo
res5: Demo.type = Demo$@5b1769c

scala> Demo.x
initializing x
res6: String = done
```

Now, initializing Demo does not involve initializing x. The initialization of x will be deferred until the first time x is used. This is similar to the situation where x is defined as a parameterless method, using a def. However, unlike a def, a lazy val is never evaluated more than once. In fact, after the first evaluation of a lazy val the result of the evaluation is stored, to be reused when the same val is used subsequently.

Looking at this example, it seems that objects like Demo themselves behave like lazy vals, in that they are also initialized on demand, the first time they are used. This is correct. In fact an object definition can be seen as a shorthand for the definition of a lazy val with an anonymous class that describes the object's contents.

Using lazy vals, you could reformulate RationalTrait as shown in Listing 20.8. In the new trait definition, all concrete fields are defined lazy. Another change with respect to the previous definition of RationalTrait, shown in Listing 20.4, is that the require clause was moved from the body of the trait to the initializer of the private field, g, which computes the greatest common divisor of numerArg and denomArg. With these changes, there's nothing that remains to be done

when LazyRationalTrait is initialized; all initialization code is now part of the right-hand side of a lazy val. Thus, it is safe to initialize the abstract fields of LazyRationalTrait after the class is defined.

```
trait LazyRationalTrait {
  val numerArg: Int
  val denomArg: Int
  lazy val numer = numerArg / g
  lazy val denom = denomArg / g
  override def toString = numer + "/" + denom
  private lazy val g = {
    require(denomArg != 0)
    gcd(numerArg, denomArg)
  }
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Listing 20.8 - Initializing a trait with lazy vals.

Here's an example:

```
scala> val x = 2
x: Int = 2

scala> new LazyRationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
res7: LazyRationalTrait = 1/2
```

No pre-initialization is needed. It's instructive to trace the sequence of initializations that lead to the string 1/2 to be printed in the code above:

1. A fresh instance of LazyRationalTrait gets created and the initialization code ofLazyRationalTrait is run. This initialization code is empty; none of the fields ofLazyRationalTrait is initialized yet.
2. Next, the primary constructor of the anonymous subclass defined by the new expression is executed. This involves the initialization of numerArg with 2 and denomArg with 4.
3. Next, the toString method is invoked on the constructed object by the interpreter, so that the resulting value can be printed.
4. Next, the numer field is accessed for the first time by the toString method in traitLazyRationalTrait, so its initializer is evaluated.
5. The initializer of numer accesses the private field, g, so g is evaluated next. This evaluation accesses numerArg and denomArg, which were defined in Step 2.
6. Next, the toString method accesses the value of denom, which causes denom's evaluation. The evaluation of denom accesses the values of denomArg and g. The initializer of the g field is not re-evaluated, because it was already evaluated in Step 5.
7. Finally, the result string "1/2" is constructed and printed.

Note that the definition of g comes textually after the definitions of numer and denom in classLazyRationalTrait. Nevertheless, because all three values are lazy, g gets initialized before the initialization of numer and denom is completed.

This shows an important property of lazy vals: The textual order of their definitions does not matter because values get initialized on demand. Thus, lazy vals can free you as a programmer from having to think hard how to arrange val definitions to ensure that everything is defined when it is needed.

However, this advantage holds only as long as the initialization of lazy vals neither produces side effects nor depends on them. In the presence of side effects, initialization order starts to matter. And then it can be quite difficult to trace in what order initialization code is run, as the previous example has demonstrated. So lazy vals are an ideal complement to functional objects, where the order of initializations does not matter, as long as everything gets initialized eventually. They are less well suited for code that's predominantly imperative.

LAZY FUNCTIONAL LANGUAGES

Scala is by no means the first language to have exploited the perfect match of lazy definitions and functional code. In fact, there is a category of "lazy functional programming languages" in which every value and parameter is initialized lazily. The best known member of this class of languages is Haskell [SPJ02].

20.6 ABSTRACT TYPES

In the beginning of this chapter, you saw, "type T", an abstract type declaration. The rest of this chapter discusses what such an abstract type declaration means and what it's good for. Like all other abstract declarations, an abstract type declaration 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 as yet unknown at the point where it is declared. 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 of modeling the eating habits of animals. You might start with a class Food and a classAnimal with an eat method:

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

You might then attempt to specialize these two classes to a class of Cows that eat Grass:

```
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) = {} // This won't compile
}
```

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

```
BuggyAnimals.scala:7: error: class Cow needs to be
abstract, since method eat in class Animal of type
    (Food)Unit is not defined
class Cow extends Animal {
      ^
BuggyAnimals.scala:8: error: method eat overrides nothing
  override def eat(food: Grass) = {}
               ^
```

What happened is that the eat method in class Cow did not override the eat method in classAnimal 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 script would pass the type checker:

```
class Food
abstract class Animal {
  def eat(food: Food)
}
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) = {} // This won't compile,
}                                    // but if it did,...
class Fish extends Food
val bessy: Animal = new Cow
bessy eat (new Fish)     // ...you could feed fish to cows.
```

The program would compile if the restriction were eased, 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 modeling. Animals do eat Food, but what kind of Food each Animal eats depends on the Animal. This can be neatly expressed with an abstract type, as shown in Listing 20.9:

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

Listing 20.9 - Modeling suitable food with an abstract type.

With the new class definition, an Animal can eat only food that's suitable. What food is suitable cannot be determined at the level of the Animal class. That's why SuitableFood is modeled 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 of Animal) must be a subclass of Food. For example, you would not be able to instantiate SuitableFood with classIOException.

With Animal defined, you can now progress to cows, as shown in Listing 20.10. Class Cow fixes its SuitableFood to be Grass and also defines a concrete eat method for this kind of food.

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

Listing 20.10 - Implementing an abstract type in a subclass.

These new class definitions compile without errors. If you tried to run the "cows-that-eat-fish" counterexample with the new class definitions, you would get the following compiler error:

```
scala> class Fish extends Food
defined class Fish

scala> val bessy: Animal = new Cow
bessy: Animal = Cow@1515d8a6

scala> bessy eat (new Fish)
<console>:14: error: type mismatch;
 found   : Fish
 required: bessy.SuitableFood
             bessy eat (new Fish)
                        ^
```

20.7 PATH-DEPENDENT TYPES

Have a look at the last error message again. What's interesting about it is the type required by the eat method: bessy.SuitableFood. This type consists of an object reference, bessy, followed by a type field, SuitableFood, of the object. So this shows that objects in Scala can have types as members. The meaning of bessy.SuitableFood is "the type SuitableFood that is a member of the object referenced from bessy" or, alternatively, the type of food that's suitable for bessy.

A type like bessy.SuitableFood is called a path-dependent type. The word "path" here means a reference an object. It could be a single name, such as bessy, or a longer access path, such asfarm.barn.bessy, where each of farm, barn, and bessy are variables (or singleton object names) that refer to objects.

As the term "path-dependent type" implies, the type depends on the path; in general, different paths give rise to different types. For instance, say you defined classes DogFood andDog, like this:

```
class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  override def eat(food: DogFood) = {}
}
```

If you attempted to feed a dog with food fit for a cow, your code would not compile:

```
scala> val bessy = new Cow
bessy: Cow = Cow@713e7e09
```

```
scala> val lassie = new Dog
lassie: Dog = Dog@6eaf2c57

scala> lassie eat (new bessy.SuitableFood)
<console>:16: error: type mismatch;
 found   : Grass
 required: DogFood
              lassie eat (new bessy.SuitableFood)
                          ^
```

The problem here is that the type of the SuitableFood object passed to the eat method,bessy.SuitableFood, is incompatible with the parameter type of eat, lassie.SuitableFood.

The case would be different for two Dogs. Because Dog's SuitableFood type is defined to be an alias for class DogFood, the SuitableFood types of two Dogs are in fact the same. As a result, the Doginstance named lassie could actually eat the suitable food of a different Dog instance (which we'll name bootsie):

```
scala> val bootsie = new Dog
bootsie: Dog = Dog@13a7c48c

scala> lassie eat (new bootsie.SuitableFood)
```

A path-dependent type resembles the syntax for an inner class type in Java, but there is a crucial difference: a path-dependent type names an outer object, whereas an inner class type names an outer class. Java-style inner class types can also be expressed in Scala, but they are written differently. Consider these two classes, Outer and Inner:

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

In Scala, the inner class is addressed using the expression Outer#Inner instead of Java'sOuter.Inner. The `.' syntax is reserved for objects. For example, imagine you instantiate two objects of type Outer, like this:

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

Here o1.Inner and o2.Inner are two path-dependent types (and they are different types). Both of these types conform to (are subtypes of) the more general type Outer#Inner, which represents the Inner class with an *arbitrary* outer object of type Outer. By contrast, typeo1.Inner refers to the Inner class with a *specific* outer object (the one referenced from o1). Likewise, type o2.Inner refers to the Inner class with a different, specific outer object (the one referenced from o2).

In Scala, as in Java, inner class instances hold a reference to an enclosing outer class instance. This allows an inner class, for example, to access members of its outer class. Thus you can't instantiate an inner class without in some way specifying an outer class instance. One way to do this is to instantiate

the inner class inside the body of the outer class. In this case, the current outer class instance (referenced from this) will be used.

Another way is to use a path-dependent type. For example, because the type, o1.Inner, names a specific outer object, you can instantiate it:

```
scala> new o1.Inner
res11: o1.Inner = Outer$Inner@1ae1e03f
```

The resulting inner object will contain a reference to its outer object, the object referenced from o1. By contrast, because the type Outer#Inner does not name any specific instance of Outer, you can't create an instance of it:

```
scala> new Outer#Inner
<console>:9: error: Outer is not a legal prefix for a
constructor
              new Outer#Inner
                        ^
```

20.8 REFINEMENT TYPES

When a class inherits from another, the first class is said to be a *nominal* subtype of the other one. It's a *nominal* subtype because each type has a *name*, and the names are explicitly declared to have a subtyping relationship. Scala additionally supports *structural* subtyping,where you get a subtyping relationship simply because two types have compatible members. To get structural subtyping in Scala, use Scala's *refinement types*.

Nominal subtyping is usually more convenient, so you should try nominal types first with any new design. A name is a single short identifier and thus is more concise than an explicit listing of member types. Further, structural subtyping is often more flexible than you want. A widget can draw(), and a Western cowboy can draw(), but they aren't really substitutable. You'd typically prefer to get a compilation error if you tried to substitute a cowboy for a widget.

Nonetheless, structural subtyping has its own advantages. One is that sometimes there really is no more to a type than its members. For example, suppose you want to define a Pastureclass that can contain animals that eat grass. One option would be to define a traitAnimalThatEatsGrass and mix it into every class where it applies. It would be verbose, however. Class Cow has already declared that it's an animal and that it eats grass, and now it would have to declare that it is also an animal-that-eats-grass.

Instead of defining AnimalThatEatsGrass, you can use a refinement type. Simply write the base type, Animal, followed by a sequence of members listed in curly braces. The members in the curly braces further specify—or refine, if you will—the types of members from the base class.

Here is how you write the type, "animal that eats grass":

```
Animal { type SuitableFood = Grass }
```

Given this type, you can now write the pasture class like this:

```
class Pasture {
  var animals: List[Animal { type SuitableFood = Grass }] = Nil
  // ...
}
```

20.9 ENUMERATIONS

An interesting application of path-dependent types is found in Scala's support for enumerations. Some other languages, including Java and C#, have enumerations as a built-in language construct to define new types. Scala does not need special syntax for enumerations. Instead, there's a class in its standard library, scala.Enumeration.

To create a new enumeration, you define an object that extends this class, as in the following example, which defines a new enumeration of Colors:

```
object Color extends Enumeration {
  val Red = Value
  val Green = Value
  val Blue = Value
}
```

Scala lets you also shorten several successive val or var definitions with the same right-hand side. Equivalently to the above you could write:

```
object Color extends Enumeration {
  val Red, Green, Blue = Value
}
```

This object definition provides three values: Color.Red, Color.Green, and Color.Blue. You could also import everything in Color with:

```
import Color._
```

and then just use Red, Green, and Blue. But what is the type of these values?

Enumeration defines an inner class named Value, and the same-named parameterless Valuemethod returns a fresh instance of that class. In other words, a value such as Color.Red is of type Color.Value; Color.Value is the type of all enumeration values defined in object Color. It's a path-dependent type, with Color being the path and Value being the dependent type. What's significant about this is that it is a completely new type, different from all other types.

In particular, if you define another enumeration, such as:

```
object Direction extends Enumeration {
  val North, East, South, West = Value
}
```

then Direction.Value would be different from Color.Value because the path parts of the two types differ.

Scala's Enumeration class also offers many other features found in the enumeration designs of other languages. You can associate names with enumeration values by using a different overloaded variant of the Value method:

```
object Direction extends Enumeration {
  val North = Value("North")
  val East = Value("East")
  val South = Value("South")
  val West = Value("West")
}
```

You can iterate over the values of an enumeration via the set returned by the enumeration'svalues method:

```
scala> for (d <- Direction.values) print(d + " ")
North East South West
```

Values of an enumeration are numbered from 0, and you can find out the number of an enumeration value by its id method:

```
scala> Direction.East.id
res14: Int = 1
```

It's also possible to go the other way, from a non-negative integer number to the value that has this number as id in an enumeration:

```
scala> Direction(1)
res15: Direction.Value = East
```

This should be enough to get you started with enumerations. You can find more information in the Scaladoc comments of class scala.Enumeration.

20.10 CASE STUDY: CURRENCIES

The rest of this chapter presents a case study that 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, euros, yen, or 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 a 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 typeLong so that very large amounts of money, such as the market capitalization of Google or Apple, can be represented. It's left abstract here, waiting to be defined when a subclass 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 floating-point number. You can create a concrete currency value by supplying concrete amountand designation values, like this:

```
new Currency {
  val amount = 79L
  def designation = "USD"
}
```

This design would be OK if all we wanted to model was a single currency, like only dollars or only euros. But it fails if we need to deal with several currencies. Assume that 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 the +method. When implemented in class Dollar, it should take Dollar arguments and yield a Dollarresult; when implemented in class Euro, it should take Euro arguments and yield a Euro result. So the type of the addition method would change depending on which class you are in. Nonetheless, 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 differences from the previous situation are that the class is now calledAbstractCurrency, and that it contains an abstract type Currency, which represents the real currency in question. Each concrete subclass of AbstractCurrency would need to fix the Currencytype to refer to the concrete subclass itself, thereby "tying the knot."

For instance, here is a new version of class Dollar, which now extends class AbstractCurrency:

```
abstract 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 the ellipses that 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 would you 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 can neither create an instance of an abstract type nor have an abstract type as a supertype of another class.[1] So the compiler would refuse the example code here that attempted to instantiate Currency.

However, you can work around this restriction using a factory method. Instead of creating an instance of an abstract type directly, declare an abstract method that 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:

```
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 dubious 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 Currencyobject. 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 classAbstractCurrency. You need to create another class that contains the AbstractCurrency class, theCurrency type, and the make factory method.

We'll call this 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 concrete CurrencyZone is the US, which could be defined as:

```
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 ofAbstractCurrency. So the type of money in this zone is US.Dollar. The US object also fixes the typeCurrency 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 a few 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.Currencyrepresent cents instead of dollars. To convert

back to dollars, it's useful to introduce a fieldCurrencyUnit into class CurrencyZone, which contains the amount of one standard unit in that currency:

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

As shown in Listing 20.11, The US object could define the quantities Cent, Dollar, andCurrencyUnit. 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 one-cent coin. The field Dollar represents an amount of 100 US.Currency. So theUS object now defines the name Dollar in two ways. The type Dollar (defined by the abstract inner class named Dollar) represents the generic name of the Currency valid in the US currency zone. By contrast, the value Dollar (referenced from the val field named Dollar) represents a single US dollar, analogous to a one-dollar bill. The third field definition of CurrencyUnitspecifies that the standard currency unit in the US zone is the Dollar (*i.e.*, the value Dollar, referenced from the field, not the type 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 could implement Currency's toString method as follows:

```
override def toString =
  ((amount.toDouble / CurrencyUnit.amount.toDouble)
   formatted ("%." + decimals(CurrencyUnit.amount) + "f")
   + " " + designation)
```

Here, formatted is a method that Scala makes available on several classes, including Double. [2]The formatted method returns the string that results from formatting the original string on which formatted was invoked according to a format string passed as the formatted method's right-hand operand. The syntax of format strings passed to formatted is the same as that of Java's String.format method.

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(cents: Long) = new Dollar {
    val amount = cents
  }
  val Cent = make(1)
  val Dollar = make(100)
  val CurrencyUnit = Dollar
}
```

Listing 20.11 - The US currency zone.

For instance, the format string %.2f formats a number with two decimal digits. The format string used in the toString shown previously is assembled by calling the decimals method onCurrencyUnit.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(n: Long): Int =
  if (n == 1) 0 else 1 + decimals(n / 10)
```

Listing 20.12 shows some other currency zones. As another refinement, you can add a currency conversion feature to the model. First, you could write a Converter object that contains applicable exchange rates between currencies, as shown in Listing 20.13. Then, you could 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 an arbitrary currency as argument. This is expressed by its formal parameter type, CurrencyZone#AbstractCurrency, which indicates that the argument passed asother must be 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.[3]

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

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

Listing 20.12 - Currency zones for Europe and Japan.

```
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  )
  )
}
```

Listing 20.13 - A converter object with an exchange rates map.

```
abstract class CurrencyZone {

  type Currency <: AbstractCurrency
  def make(x: Long): Currency

  abstract class AbstractCurrency {

    val amount: Long
    def designation: String

    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
    def - (that: Currency): Currency =
      make(this.amount - that.amount)
    def / (that: Double) =
      make((this.amount / that).toLong)
    def / (that: Currency) =
      this.amount.toDouble / that.amount

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

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

    override def toString =
      ((amount.toDouble / CurrencyUnit.amount.toDouble)
       formatted ("%." + decimals(CurrencyUnit.amount) + "f")
       + " " + designation)
  }

  val CurrencyUnit: Currency
}
```

Listing 20.14 - The full code of class CurrencyZone.

The final version of the CurrencyZone class is shown in Listing 20.14. You can test the class in the Scala command shell. We'll assume that the CurrencyZone class and all concrete CurrencyZoneobjects are defined in a package org.stairwaybook.currencies. The first step is to import "org.stairwaybook.currencies._" into the command shell. Then you can do some currency conversions:

```
scala> Japan.Yen from US.Dollar * 100
```

```
res16: Japan.Currency = 12110 JPY

scala> Europe.Euro from res16
res17: Europe.Currency = 75.95 EUR

scala> US.Dollar from res17
res18: US.Currency = 99.95 USD
```

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 + res18
res19: US.Currency = 199.95 USD
```

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

```
scala> US.Dollar + Europe.Euro
<console>:12: error: type mismatch;
 found   : Europe.Euro
 required: US.Currency
    (which expands to)  US.Dollar
              US.Dollar + Europe.Euro
                                ^
```

By preventing the addition of two values with different units (in this case, currencies), the type abstraction has done its job. It prevents us from performing calculations that 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 September 23, 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 currencies 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 ten-month voyage.

20.11 CONCLUSION

Scala offers systematic and very general support for object-oriented abstraction. It enables you to not only abstract over methods, but also over values, variables, and types. This chapter has shown how to take advantage of abstract members. They support a simple yet effective principle for systems structuring: when designing a class, make everything that is not yet known into an abstract member. The type system will then drive the development of your model, just as you saw with the currency case study. It does not matter whether the unknown is a type, method, variable or value. In Scala, all of these can be declared abstract.

Footnotes for Chapter 20:

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

[2] Scala uses rich wrappers, described in Section 5.10, to make formatted available.

[3] By the way, in case you think you're getting a bad deal on Japanese yen, the exchange rates convert currencies based on their CurrencyZone amounts. Thus, 1.211 is the exchange rate between US cents and Japanese yen.