Chapter 12

Traits

Traits are a fundamental unit of code reuse in Scala. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits. This chapter shows you how traits work and shows two of the most common ways they are useful: widening thin interfaces to rich ones, and defining stackable modifications. It also shows how to use the Ordered trait and compares traits to the multiple inheritance of other languages.

12.1 HOW TRAITS WORK

A trait definition looks just like a class definition except that it uses the keyword trait. An example is shown in Listing 12.1:

```
trait Philosophical {
  def philosophize() = {
    println("I consume memory, therefore I am!")
  }
}
```

Listing 12.1 - The definition of trait Philosophical.

This trait is named Philosophical. It does not declare a superclass, so like a class, it has the default superclass of AnyRef. It defines one method, named philosophize, which is concrete. It's a simple trait, just enough to show how traits work.

Once a trait is defined, it can be *mixed in* to a class using either the extends or with keywords.Scala programmers "mix in" traits rather than inherit from them, because mixing in a trait has important differences from the multiple inheritance found in many other languages. This issue is discussed in Section 12.6. For example, Listing 12.2 shows a class that mixes in thePhilosophical trait using extends:

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

Listing 12.2 - Mixing in a trait using extends.

You can use the extends keyword to mix in a trait; in that case you implicitly inherit the trait's superclass. For instance, in Listing 12.2, class Frog subclasses AnyRef (the superclass ofPhilosophical) and mixes in Philosophical. Methods inherited from a trait can be used just like methods inherited from a superclass. Here's an example:

```
scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize()
```

```
I consume memory, therefore I am!
```

A trait also defines a type. Here's an example in which Philosophical is used as a type:

```
scala> val phil: Philosophical = frog
phil: Philosophical = green

scala> phil.philosophize()
I consume memory, therefore I am!
```

The type of phil is Philosophical, a trait. Thus, variable phil could have been initialized with any object whose class mixes in Philosophical.

If you wish to mix a trait into a class that explicitly extends a superclass, you use extends to indicate the superclass and with to mix in the trait. Listing 12.3 shows an example. If you want to mix in multiple traits, you add more with clauses. For example, given a trait HasLegs, you could mix both Philosophical and HasLegs into Frog as shown in Listing 12.4.

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

Listing 12.3 - Mixing in a trait using with.

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

Listing 12.4 - Mixing in multiple traits.

In the examples you've seen so far, class Frog has inherited an implementation of philosophizefrom trait Philosophical. Alternatively, Frog could override philosophize. The syntax looks the same as overriding a method declared in a superclass. Here's an example:

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
  override def philosophize() = {
    println("It ain't easy being " + toString + "!")
  }
}
```

Because this new definition of Frog still mixes in trait Philosophical, you can still use it from a variable of that type. But because Frog overrides Philosophical's implementation of philosophize, you'll get a new behavior when you call it:

```
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green
```

```
scala> phrog.philosophize()
It ain't easy being green!
```

At this point you might philosophize that traits are like Java interfaces with concrete methods, but they can actually do much more. Traits can, for example, declare fields and maintain state. In fact, you can do anything in a trait definition that you can do in a class definition, and the syntax looks exactly the same, with only two exceptions.

First, a trait cannot have any "class" parameters (*i.e.*, parameters passed to the primary constructor of a class). In other words, although you could define a class like this:

```
class Point(x: Int, y: Int)
```

The following attempt to define a trait would not compile:

```
trait NoPoint(x: Int, y: Int) // Does not compile
```

You'll find out in Section 20.5 how to work around this restriction.

The other difference between classes and traits is that whereas in classes, super calls are statically bound, in traits, they are dynamically bound. If you write "super.toString" in a class, you know exactly which method implementation will be invoked. When you write the same thing in a trait, however, the method implementation to invoke for the super call is undefined when you define the trait. Rather, the implementation to invoke will be determined anew each time the trait is mixed into a concrete class. This curious behavior of super is key to allowing traits to work as *stackable modifications*, which will be described in Section 12.5. The rules for resolving super calls will be given in Section 12.6.

12.2 THIN VERSUS RICH INTERFACES

One major use of traits is to automatically add methods to a class in terms of methods the class already has. That is, traits can enrich a *thin* interface, making it into a *rich* interface.

Thin versus rich interfaces represents a commonly faced trade-off in object-oriented design. The trade-off is between the implementers and the clients of an interface. A rich interface has many methods, which make it convenient for the caller. Clients can pick a method that exactly matches the functionality they need. A thin interface, on the other hand, has fewer methods, and thus is easier on the implementers. Clients calling into a thin interface, however, have to write more code. Given the smaller selection of methods to call, they may have to choose a less than perfect match for their needs and write extra code to use it.

Java's interfaces are more often thin than rich. For example, interface CharSequence, which was introduced in Java 1.4, is a thin interface common to all string-like classes that hold a sequence of characters. Here's its definition when seen as a Scala trait:

```
trait CharSequence {
  def charAt(index: Int): Char
  def length: Int
  def subSequence(start: Int, end: Int): CharSequence
```

```
  def toString(): String
}
```

Although most of the dozens of methods in class String would apply to any CharSequence, Java'sCharSequence interface declares only four methods. Had CharSequence instead included the fullString interface, it would have placed a large burden on implementers of CharSequence. Every programmer that implemented CharSequence in Java would have had to define dozens more methods. Because Scala traits can contain concrete methods, they make rich interfaces far more convenient.

Adding a concrete method to a trait tilts the thin-rich trade-off heavily towards rich interfaces. Unlike in Java, adding a concrete method to a Scala trait is a one-time effort. You only need to implement the method once, in the trait itself, instead of needing to reimplement it for every class that mixes in the trait. Thus, rich interfaces are less work to provide in Scala than in a language without traits.

To enrich an interface using traits, simply define a trait with a small number of abstract methods—the thin part of the trait's interface—and a potentially large number of concrete methods, all implemented in terms of the abstract methods. Then you can mix the enrichment trait into a class, implement the thin portion of the interface, and end up with a class that has all of the rich interface available.

12.3 EXAMPLE: RECTANGULAR OBJECTS

Graphics libraries often have many different classes that represent something rectangular. Some examples are windows, bitmap images, and regions selected with a mouse. To make these rectangular objects convenient to use, it is nice if the library provides geometric queries, such as width, height, left, right, topLeft, and so on. However, many such methods exist that would be nice to have, so it can be a large burden on library writers to provide all of them for all rectangular objects in a Java library. If such a library were written in Scala, by contrast, the library writer could use traits to easily supply all of these convenience methods on all the classes they like.

To see how, first imagine what the code would look like without traits. There would be some basic geometric classes like Point and Rectangle:

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val bottomRight: Point) {
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // and many more geometric methods...
}
```

This Rectangle class takes two points in its primary constructor: the coordinates of the top-left and bottom-right corners. It then implements many convenience methods, such as left, right, and width, by performing simple calculations on these two points.

Another class a graphics library might have is a 2-D graphical widget:

```
abstract class Component {
  def topLeft: Point
```

```
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // and many more geometric methods...
}
```

Notice that the definitions of left, right, and width are exactly the same in the two classes. They will also be the same, aside from minor variations, in any other classes for rectangular objects.

This repetition can be eliminated with an enrichment trait. The trait will have two abstract methods: one that returns the top-left coordinate of the object, and another that returns the bottom-right coordinate. It can then supply concrete implementations of all the other geometric queries. Listing 12.5 shows what it will look like:

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // and many more geometric methods...
}
```

Listing 12.5 - Defining an enrichment trait.

Class Component can mix in this trait to get all the geometric methods provided by Rectangular:

```
abstract class Component extends Rectangular {
  // other methods...
}
```

Similarly, Rectangle itself can mix in the trait:

```
class Rectangle(val topLeft: Point, val bottomRight: Point)
    extends Rectangular {

  // other methods...
}
```

Given these definitions, you can create a Rectangle and call geometric methods such as widthand left on it:

```
scala> val rect = new Rectangle(new Point(1, 1),
           new Point(10, 10))
rect: Rectangle = Rectangle@5f5da68c

scala> rect.left
res2: Int = 1

scala> rect.right
res3: Int = 10

scala> rect.width
```

```
res4: Int = 9
```

12.4 THE ORDERED TRAIT

Comparison is another domain where a rich interface is convenient. Whenever you compare two objects that are ordered, it is convenient if you use a single method call to ask about the precise comparison you want. If you want "is less than," you would like to call <, and if you want "is less than or equal," you would like to call <=. With a thin comparison interface, you might just have the < method, and you would sometimes have to write things like "(x < y) || (x == y)". A rich interface would provide you with all of the usual comparison operators, thus allowing you to directly write things like "x <= y".

Before looking at Ordered, imagine what you might do without it. Suppose you took the Rationalclass from Chapter 6 and added comparison operations to it. You would end up with something like this:[1]

```
class Rational(n: Int, d: Int) {
  // ...
  def < (that: Rational) =
    this.numer * that.denom < that.numer * this.denom
  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
}
```

This class defines four comparison operators (<, >, <=, and >=), and it's a classic demonstration of the costs of defining a rich interface. First, notice that three of the comparison operators are defined in terms of the first one. For example, > is defined as the reverse of <, and <= is defined as literally "less than or equal." Next, notice that all three of these methods would be the same for any other class that is comparable. There is nothing special about rational numbers regarding <=. In a comparison context, <= is *always* used to mean "less than or equals." Overall, there is quite a lot of boilerplate code in this class which would be the same in any other class that implements comparison operations.

This problem is so common that Scala provides a trait to help with it. The trait is calledOrdered. To use it, you replace all of the individual comparison methods with a single comparemethod. The Ordered trait then defines <, >, <=, and >= for you in terms of this one method. Thus, trait Ordered allows you to enrich a class with comparison methods by implementing only one method, compare.

Here is how it looks if you define comparison operations on Rational by using the Ordered trait:

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
  // ...
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
}
```

There are just two things to do. First, this version of Rational mixes in the Ordered trait. Unlike the traits you have seen so far, Ordered requires you to specify a *type parameter* when you mix it in. Type parameters are not discussed in detail until Chapter 19, but for now all you need to know is that when

you mix in Ordered, you must actually mix in Ordered[*C*], where *C* is the class whose elements you compare. In this case, Rational mixes in Ordered[Rational].

The second thing you need to do is define a compare method for comparing two objects. This method should compare the receiver, this, with the object passed as an argument to the method. It should return an integer that is zero if the objects are the same, negative if receiver is less than the argument, and positive if the receiver is greater than the argument.

In this case, the comparison method of Rational uses a formula based on converting the fractions to a common denominator and then subtracting the resulting numerators. Given this mixin and the definition of compare, class Rational now has all four comparison methods:

```
scala> val half = new Rational(1, 2)
half: Rational = 1/2

scala> val third = new Rational(1, 3)
third: Rational = 1/3

scala> half < third
res5: Boolean = false

scala> half > third
res6: Boolean = true
```

Any time you implement a class that is ordered by some comparison, you should consider mixing in the Ordered trait. If you do, you will provide the class's users with a rich set of comparison methods.

Beware that the Ordered trait does not define equals for you, because it is unable to do so. The problem is that implementing equals in terms of compare requires checking the type of the passed object, and because of type erasure, Ordered itself cannot do this test. Thus, you need to define equals yourself, even if you inherit Ordered. You'll find out how to go about this inChapter 30.

The complete Ordered trait, minus comments and compatibility cruft, is as follows:

```
trait Ordered[T] {
  def compare(that: T): Int

  def <(that: T): Boolean = (this compare that) < 0
  def >(that: T): Boolean = (this compare that) > 0
  def <=(that: T): Boolean = (this compare that) <= 0
  def >=(that: T): Boolean = (this compare that) >= 0
}
```

Do not worry much about the T's and [T]'s. T here is a type parameter, as described in detail inChapter 19. For understanding the Ordered trait, just think of it as "the same type as the receiver". You can then see that this trait defines one abstract method, compare, which is expected to compare the receiver (this) against another object of the same type as the receiver (that). Given this one method, Ordered can then provide concrete definitions for <, >, <=, and >=.

12.5 TRAITS AS STACKABLE MODIFICATIONS

You have now seen one major use of traits: turning a thin interface into a rich one. Now we'll turn to a second major use: providing stackable modifications to classes. Traits let you*modify* the methods of a class, and they do so in a way that allows you to *stack* those modifications with each other.

As an example, consider stacking modifications to a queue of integers. The queue will have two operations: put, which places integers in the queue, and get, which takes them back out. Queues are first-in, first-out, so get should return the integers in the same order they were put in the queue.

Given a class that implements such a queue, you could define traits to perform modifications such as these:

- Doubling: double all integers that are put in the queue
- Incrementing: increment all integers that are put in the queue
- Filtering: filter out negative integers from a queue

These three traits represent *modifications*, because they modify the behavior of an underlying queue class rather than defining a full queue class themselves. The three are also*stackable*. You can select any of the three you like, mix them into a class, and obtain a new class that has all of the modifications you chose.

An abstract IntQueue class is shown in Listing 12.6. IntQueue has a put method that adds new integers to the queue, and a get method that removes and returns them. A basic implementation of IntQueue that uses an ArrayBuffer is shown in Listing 12.7.

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
```

Listing 12.6 - Abstract class IntQueue.

```
import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) = { buf += x }
}
```

Listing 12.7 - A BasicIntQueue implemented with an ArrayBuffer.

Class BasicIntQueue has a private field holding an array buffer. The get method removes an entry from one end of the buffer, while the put method adds elements to the other end. Here's how this implementation looks when you use it:

```
scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@23164256

scala> queue.put(10)
```

```
scala> queue.put(20)

scala> queue.get()
res9: Int = 10

scala> queue.get()
res10: Int = 20
```

So far so good. Now take a look at using traits to modify this behavior. Listing 12.8 shows a trait that doubles integers as they are put in the queue. The Doubling trait has two funny things going on. The first is that it declares a superclass, IntQueue. This declaration means that the trait can only be mixed into a class that also extends IntQueue. Thus, you can mix Doubling intoBasicIntQueue, but not into Rational.

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) = { super.put(2 * x) }
}
```

Listing 12.8 - The Doubling stackable modification trait.

The second funny thing is that the trait has a super call on a method declared abstract. Such calls are illegal for normal classes because they will certainly fail at run time. For a trait, however, such a call can actually succeed. Since super calls in a trait are dynamically bound, the super call in trait Doubling will work so long as the trait is mixed in *after* another trait or class that gives a concrete definition to the method.

This arrangement is frequently needed with traits that implement stackable modifications. To tell the compiler you are doing this on purpose, you must mark such methods as abstractoverride. This combination of modifiers is only allowed for members of traits, not classes, and it means that the trait must be mixed into some class that has a concrete definition of the method in question.

There is a lot going on with such a simple trait, isn't there! Here's how it looks to use the trait:

```
scala> class MyQueue extends BasicIntQueue with Doubling
defined class MyQueue

scala> val queue = new MyQueue
queue: MyQueue = MyQueue@44bbf788

scala> queue.put(10)

scala> queue.get()
res12: Int = 20
```

In the first line in this interpreter session, we define class MyQueue, which extends BasicIntQueueand mixes in Doubling. We then put a 10 in the queue, but because Doubling has been mixed in, the 10 is doubled. When we get an integer from the queue, it is a 20.

Note that MyQueue defines no new code. It simply identifies a class and mixes in a trait. In this situation, you could supply "BasicIntQueue with Doubling" directly to new instead of defining a named class. It would look as shown in Listing 12.9:

```
scala> val queue = new BasicIntQueue with Doubling
queue: BasicIntQueue with Doubling = $anon$1@141f05bf

scala> queue.put(10)

scala> queue.get()
res14: Int = 20
```

Listing 12.9 - Mixing in a trait when instantiating with new.

To see how to stack modifications, we need to define the other two modification traits,Incrementing and Filtering. Implementations of these traits are shown in Listing 12.10:

```
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) = { super.put(x + 1) }
}
trait Filtering extends IntQueue {
  abstract override def put(x: Int) = {
    if (x >= 0) super.put(x)
  }
}
```

Listing 12.10 - Stackable modification traits Incrementing and Filtering.

Given these modifications, you can now pick and choose which ones you want for a particular queue. For example, here is a queue that both filters negative numbers and adds one to all numbers that it keeps:

```
scala> val queue = (new BasicIntQueue
           with Incrementing with Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res16: Int = 1

scala> queue.get()
res17: Int = 2
```

The order of mixins is significant.[2] The precise rules are given in the following section, but, roughly speaking, traits further to the right take effect first. When you call a method on a class with mixins, the method in the trait furthest to the right is called first. If that method calls super, it invokes the method in the next trait to its left, and so on. In the previous example, Filtering's put is invoked first, so it removes integers that were negative to begin with. Incrementing's put is invoked second, so it adds one to those integers that remain.

If you reverse the order, first integers will be incremented, and *then* the integers that are still negative will be discarded:

```
scala> val queue = (new BasicIntQueue
           with Filtering with Incrementing)
queue: BasicIntQueue with Filtering with Incrementing...
```

```
scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res19: Int = 0

scala> queue.get()
res20: Int = 1

scala> queue.get()
res21: Int = 2
```

Overall, code written in this style gives you a great deal of flexibility. You can define sixteen different classes by mixing in these three traits in different combinations and orders. That's a lot of flexibility for a small amount of code, so you should keep your eyes open for opportunities to arrange code as stackable modifications.

12.6 WHY NOT MULTIPLE INHERITANCE?

Traits are a way to inherit from multiple class-like constructs, but they differ in important ways from the multiple inheritance present in many languages. One difference is especially important: the interpretation of super. With multiple inheritance, the method called by a supercall can be determined right where the call appears. With traits, the method called is determined by a *linearization* of the classes and traits that are mixed into a class. This is the difference that enables the stacking of modifications described in the previous section.

Before looking at linearization, take a moment to consider how to stack modifications in a language with traditional multiple inheritance. Imagine the following code, but this time interpreted as multiple inheritance instead of trait mixin:

```
// Multiple inheritance thought experiment
val q = new BasicIntQueue with Incrementing with Doubling
q.put(42)  // which put would be called?
```

The first question is: Which put method would get invoked by this call? Perhaps the rule would be that the last superclass wins, in which case Doubling would get called. Doubling would double its argument and call super.put, and that would be it. No incrementing would happen! Likewise, if the rule were that the first superclass wins, the resulting queue would increment integers but not double them. Thus neither ordering would work.

You might also entertain the possibility of allowing programmers to identify exactly which superclass method they want when they say super. For example, imagine the following Scala-like code, in which super appears to be explicitly invoked on both Incrementing and Doubling:

```
// Multiple inheritance thought experiment
trait MyQueue extends BasicIntQueue
    with Incrementing with Doubling {

  def put(x: Int) = {
    Incrementing.super.put(x) // (Not real Scala)
    Doubling.super.put(x)
  }
```

```
}
```

This approach would give us new problems (with the verbosity of this attempt being the least of its problems). What would happen is that the base class's put method would get called*twice*—once with an incremented value and once with a doubled value, but neither time with an incremented, doubled value.

There is simply no good solution to this problem using multiple inheritance. You would have to back up in your design and factor the code differently. By contrast, the traits solution in Scala is straightforward. You simply mix in Incrementing and Doubling, and Scala's special treatment of super in traits makes it all work out. Something is clearly different here from traditional multiple inheritance, but what? As hinted previously, the answer is linearization. When you instantiate a class with new, Scala takes the class, and all of its inherited classes and traits, and puts them in a single, *linear* order. Then, whenever you call super inside one of those classes, the invoked method is the next one up the chain. If all of the methods but the last call super, the net result is stackable behavior.

The precise order of the linearization is described in the language specification. It is a little bit complicated, but the main thing you need to know is that, in any linearization, a class is always linearized in front of *all* its superclasses and mixed in traits. Thus, when you write a method that calls super, that method is definitely modifying the behavior of the superclasses and mixed in traits, not the other way around.

Note

The remainder of this section describes the details of linearization. You can safely skip the rest of this section if you are not interested in understanding those details right now.

The main properties of Scala's linearization are illustrated by the following example: Say you have a class Cat, which inherits from a superclass Animal and two supertraits Furry andFourLegged. FourLegged extends in turn another trait HasLegs:

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Class Cat's inheritance hierarchy and linearization are shown in Figure 12.1. Inheritance is indicated using traditional UML notation:[3] arrows with white, triangular arrowheads indicate inheritance, with the arrowhead pointing to the supertype. The arrows with darkened, non-triangular arrowheads depict linearization. The darkened arrowheads point in the direction in which super calls will be resolved.

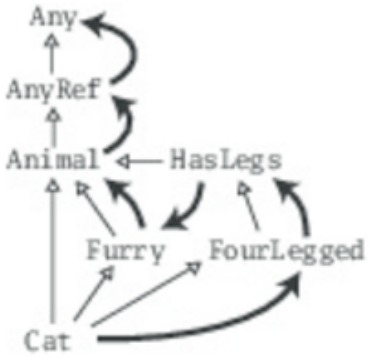

Figure 12.1 - Inheritance hierarchy and linearization of class Cat.

The linearization of Cat is computed from back to front as follows. The last part of the linearization of Cat is the linearization of its superclass, Animal. This linearization is copied over without any changes. (The linearization of each of these types is shown in Table 12.1here.) Because Animal doesn't explicitly extend a superclass or mix in any supertraits, it by default extends AnyRef, which extends Any. Animal's linearization, therefore, looks like:

Animal → AnyRef → Any

The second to last part is the linearization of the first mixin, trait Furry, but all classes that are already in the linearization of Animal are left out now, so that each class appears only once inCat's linearization. The result is:

Furry → Animal → AnyRef → Any

This is preceded by the linearization of FourLegged, where again any classes that have already been copied in the linearizations of the superclass or the first mixin are left out:

FourLegged → HasLegs → Furry → Animal → AnyRef → Any

Finally, the first class in the linearization of Cat is Cat itself:

Cat → FourLegged → HasLegs → Furry → Animal → AnyRef → Any

When any of these classes and traits invokes a method via super, the implementation invoked will be the first implementation to its right in the linearization.

Table 12.1 - Linearization of types in Cat's hierarchy

Type	Linearization
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any

FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

12.7 TO TRAIT OR NOT TO TRAIT?

Whenever you implement a reusable collection of behavior, you will have to decide whether you want to use a trait or an abstract class. There is no firm rule, but this section contains a few guidelines to consider.

If the behavior will not be reused, then make it a concrete class. It is not reusable behavior after all.

If it might be reused in multiple, unrelated classes, make it a trait. Only traits can be mixed into different parts of the class hierarchy.

If you want to inherit from it in Java code, use an abstract class. Since traits with code do not have a close Java analog, it tends to be awkward to inherit from a trait in a Java class. Inheriting from a Scala class, meanwhile, is exactly like inheriting from a Java class. As one exception, a Scala trait with only abstract members translates directly to a Java interface, so you should feel free to define such traits even if you expect Java code to inherit from it. SeeChapter 31 for more information on working with Java and Scala together.

If you plan to distribute it in compiled form, and you expect outside groups to write classes inheriting from it, you might lean towards using an abstract class. The issue is that when a trait gains or loses a member, any classes that inherit from it must be recompiled, even if they have not changed. If outside clients will only call into the behavior, instead of inheriting from it, then using a trait is fine.

If you still do not know, after considering the above, then start by making it as a trait. You can always change it later, and in general using a trait keeps more options open.

12.8 CONCLUSION

This chapter has shown you how traits work and how to use them in several common idioms. You saw that traits are similar to multiple inheritance. But because traits interpret super using linearization, they both avoid some of the difficulties of traditional multiple inheritance and allow you to stack behaviors. You also saw the Ordered trait and learned how to write your own enrichment traits.

Now that you have seen all of these facets, it is worth stepping back and taking another look at traits as a whole. Traits do not merely support the idioms described in this chapter; they are a fundamental unit of code that is reusable through inheritance. As such, many experienced Scala programmers start with traits when they are at the early stages of implementation. Each trait can hold less than an entire concept, a mere fragment of a concept. As the design solidifies, the fragments can be combined into more complete concepts through trait mixin.

Footnotes for Chapter 12:

[1] This example is based on the Rational class shown in Listing 6.5 here, with equals, hashCode, and modifications to ensure a positive denom added.

[2] Once a trait is mixed into a class, you can alternatively call it a *mixin*.

[3] Rumbaugh, *et. al.*, *The Unified Modeling Language Reference Manual*. [Rum04]