Table of Contents for Programming Scala, 2nd Edition

Chapter 9. Traits

In Java, a class can implement an arbitrary number of *interfaces*. This model is very useful for declaring that a class implements multiple abstractions. Unfortunately, it has one major drawback. For many interfaces, much of the functionality can be implemented with boilerplate code that will be valid for all classes that use the interface.

Often, an interface has members that are unrelated ("orthogonal") to the rest of the members in the implementating class. The term *mixin* is used for such focused and reusable state and behavior. Ideally, we would maintain this behavior independent of any concrete types that use it.

Before version 8, Java provided no built-in mechanism for defining and using such reusable code. Instead, Java programmers must use ad hoc conventions to reuse implementation code for a given interface. In the worst case, the developer just copies and pastes the same code into every class that needs it. A better, but imperfect solution has been to implement a separate class for the behavior, keep an instance of it in the original class, and then delegate method calls to the support class. This approach works, but it adds lots of needless overhead and error-prone boilerplate.

Interfaces in Java 8

Java 8 changes this picture. Interfaces can now *define* methods, called *defender methods* or just *default methods*. A class can still provide its own implementation, but if it doesn't the defender method is used. So, Java 8 interfaces behave more like Scala's traits.

However, one difference is that Java 8 interfaces are still limited to defining static fields, whereas Scala traits can define instance-level fields. This means that Java 8 interfaces can't manage any part of an instance's state. A class that implements the interface will have to supply fields for state. This also means that the defender methods have no state to access in their implementations, limiting what they can do.

In the discussion that follows, consider how you would implement the traits and classes presented using Java 8 interfaces and classes. What would port over easily to Java 8 and what would not?

Traits as Mixins

To consider the role of traits in modularizing Scala code, let's begin with the following code for a button in a graphical user interface (GUI) toolkit, which uses callbacks to notify clients when clicks occur:

```
//
src/main/scala/progscala2/traits/ui/ButtonCallbacks.scala
package progscala2.traits.ui

class ButtonWithCallbacks(val label: String,
    val callbacks: List[() => Unit] = Nil) extends Widget {

  def click(): Unit = {
    updateUI()
    callbacks.foreach(f => f())
  }

                                      /* logic to change GUI appearance
  protected def updateUI(): Unit = { */                                    }
}

object ButtonWithCallbacks {

  def apply(label: String, callback: () => Unit) =
    new ButtonWithCallbacks(label, List(callback))

  def apply(label: String) =
    new ButtonWithCallbacks(label, Nil)
}
```

`Widget` is a “marker” trait we will expand later:

```
// src/main/scala/progscala2/traits/ui/Widget.scala
package progscala2.traits.ui

abstract class Widget
```

A list of callback functions of type `() => Unit` (pure side effects) are called when the button is clicked.

This class has two responsibilities: updating the visual appearance (which we’ve elided) and handling callback behavior, including the management of a list of callbacks and calling them whenever the button is clicked.

We should strive for *separation of concerns* in our types, as embodied in the *Single Responsibility Principle*, which says that every type should “do one thing” and not mix multiple responsibilities.

We would like to separate the button-specific logic from the callback logic, such that each part becomes simpler, more modular, easier to test and modify, and more reusable. The callback logic is a good candidate for a *mixin*.

Let’s use traits to separate the callback handling from the button logic. We’ll generalize our approach a little bit. Callbacks are really a special case of the *Observer Design Pattern*. So, let’s create two traits that declare and partially implement the `Subject` and `Observer` logic from this pattern, then use them to handle callback behavior. To simplify things, we’ll start with a single callback that counts the number of button clicks:

```
// src/main/scala/progscala2/traits/observer/Observer.scala
package progscala2.traits.observer

trait Observer[-State] {
// ❶
  def receiveUpdate(state: State): Unit
}

trait Subject[State] {
// ❷
  private var observers: List[Observer[State]] = Nil                    //
❸

  def addObserver(observer:Observer[State]): Unit =                     //
❹
    observers ::= observer
// ❺

  def notifyObservers(state: State): Unit =
// ❻
    observers foreach (_.receiveUpdate(state))
}
```

❶ The trait for clients who want to be notified of state changes. They must implement the `receiveUpdate` message.

❷ The trait for subjects who will send notifications to observers.

❸ A list of observers to notify. It's mutable, so it's not thread-safe!

❹ A method to add observers.

❺ This expression means "prefix `observer` to `observers` and assign the result to `observers`."

❻ A method to notify observers of state changes.

Often, the most convenient choice for the `State` type parameter is just the type of the class mixing in `Subject`. Hence, when the `notifyObservers` method is called, the instance just passes itself, i.e., `this`.

Note that because `Observer` doesn't define the method it declares nor does it declare any other members, it is exactly equivalent to a pre-Java 8 interface at the byte code level. No abstract keyword is required, because the missing method body makes it clear. However, abstract classes, those with unimplemented method bodies, must be declared with the `abstract` keyword. For example:

```
trait PureAbstractTrait {
  def abstractMember(str: String): Int
}

abstract class AbstractClass {
  def concreteMember(str: String): Int = str.length
  def abstractMember(str: String): Int
}
```

Note

Traits with abstract members don't have to be declared abstract by adding the `abstract` keyword before the `trait` keyword. However, classes that are abstract, where one or more member is undefined, must be declared abstract.

To expand on the fifth note, because `observers` is mutable, the following two expressions are equivalent:

```
observers ::= observer
observers = observer :: observers
```

Now, let's define a simplified `Button` class:

```
// src/main/scala/progscala2/traits/ui/Button.scala
package progscala2.traits.ui

class Button(val label: String) extends Widget {

  def click(): Unit = updateUI()

                            /* logic to change GUI appearance
  def updateUI(): Unit = { */                                    }
}
```

`Button` is considerably simpler. It has only one concern, handling clicks.

Let's put our `Subject` trait to use. Here is `ObservableButton`, which subclasses `Button` and mixes in `Subject`:

```
// src/main/scala/progscala2/traits/ui/ObservableButton.scala
package progscala2.traits.ui
import progscala2.traits.observer._

class ObservableButton(name: String)
// ❶
    extends Button(name) with Subject[Button] {
// ❷

  override def click(): Unit = {
// ❸
    super.click()
// ❹
    notifyObservers(this)
// ❺
  }
}
```

❶

A subclass of `Button` that mixes in observability.

❷

Extends `Button` and mixes in `Subject` and uses `Button` as the `Subject` type parameter, named `State` in the declaration of `Subject`.

❸

In order to notify observers, we have to override the `click` method. If there were other state-changing methods, all would require overrides.

❹

First, call the parent class `click` to perform the normal GUI update logic.

❺

Notify the observers, passing `this` as the `State`. In this case, there isn't any state other than the "event" that a button click occurred.

Let's try it with the following script:

```
// src/main/scala/progscala2/traits/ui/button-count-observer.sc
import progscala2.traits.ui._
import progscala2.traits.observer._

class ButtonCountObserver extends Observer[Button] {
  var count = 0
  def receiveUpdate(state: Button): Unit = count += 1
}

                                  "Click
val button = new ObservableButton(Me!"       )
val bco1   = new ButtonCountObserver
val bco2   = new ButtonCountObserver

button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

assert(bco1.count == 5)
assert(bco2.count == 5)
```

The script declares an observer type, `ButtonCountObserver`, that counts clicks. Then it creates two instances and an `ObservableButton`, and it registers the two observers with the button. It clicks the button five times and then uses the `assert` method in `Predef` to verify the counts for each observer equals five.

Suppose we only need one instance of an `ObservableButton`. We don't need to declare a class that mixes in the traits we want. Instead, we can declare a `Button` and mix in the `Subject` trait "on the fly":

```
// src/main/scala/progscala2/traits/ui/button-count-observer2.sc
import progscala2.traits.ui._
import progscala2.traits.observer._

                        "Click
val button = new Button(Me!"       ) with Subject[Button] {

  override def click(): Unit = {
    super.click()
    notifyObservers(this)
  }
}

class ButtonCountObserver extends Observer[Button] {
  var count = 0
  def receiveUpdate(state: Button): Unit = count += 1
}

val bco1   = new ButtonCountObserver
val bco2   = new ButtonCountObserver

button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

assert(bco1.count == 5)
assert(bco2.count == 5)
println("Success!")
```

We declared a `Button() with Subject[Button]` *instance*, without declaring a class first. This is similar to instantiating an anonymous class that implements an interface in Java, but Scala provides more flexibility.

Note

When declaring a class that only mixes in traits and doesn't extend another class, you must use the `extends` keyword anyway for the first trait listed and the `with` keyword for the rest of the traits. However, when instantiating a class and mixing in traits with the declaration, use the `with` keyword with all the traits.

In Good Object-Oriented Design: A Digression, I recommended stringent rules for subclassing. Have we broken those "rules"? Let's look at each one:

An abstract base class or trait is subclassed one level by concrete classes, including case classes
: We didn't use an abstract base class in this case.

Concrete classes are never subclassed, except for two cases: (1) classes that mix in other behaviors in traits
: Although `Button` and `ObservableButton`, its child, are both concrete, the latter only mixes in traits.

Concrete classes are never subclassed, except for two cases: (2) test-only versions to promote automated unit testing
: Not applicable.

When subclassing seems like the right approach, consider partitioning behaviors into traits and mix in those traits instead
: We did!!

Never split logical state across parent-child type boundaries

The logical, visible state for buttons or other UI widgets is disjoint from the internal mechanism of state-change notification. So, UI state is still encapsulated in `Button`, while the state associated with the *Observer* pattern, i.e., the list of observers, is encapsulated in the `State` trait.

Stackable Traits

There are several, further refinements we can do to improve the reusability of our code and to make it easier to use more than one trait at a time, i.e., to "stack" traits.

First, "clickability" is not limited to buttons in a GUI. We should abstract that logic, too. We could put it in `Widget`, the so-far empty parent type of `Button`, but it's not necessarily true that all GUI widgets will accept clicks. Instead, let's introduce another trait, `Clickable`:

```
//
src/main/scala/progscala2/traits/ui2/Clickable.scala
package progscala2.traits.ui2
// ❶

trait Clickable {
  def click(): Unit = updateUI()
// ❷

  protected def updateUI(): Unit
// ❸
}
```

❶

Use a new package because we're reimplementing types in `traits.ui`.

❷

The public method `click` is concrete. It delegates to `updateUI`.

❸

The `updateUI` method is protected and abstract. Implementing classes provide the appropriate logic.

Although this simple interface doesn't really need to follow the example of `Button`, where `click` delegates to a `protected` method, it was a useful idiom for separation of a public abstraction from an implementation detail, so we follow it here. It's a simple example of the *Template Method Pattern* described in the "Gang of Four" patterns book.

Here is the refactored button, which uses the trait:

```
//
src/main/scala/progscala2/traits/ui2/Button.scala
package progscala2.traits.ui2
import progscala2.traits.ui.Widget

class Button(val label: String) extends Widget with Clickable {

                                      /* logic to change GUI appearance
  protected def updateUI(): Unit = { */                                    }
}
```

Observation should now be tied to `Clickable` and not `Button`, as it was before. When we refactor the code this way, it becomes clear that we don't really care about observing buttons. We really care about observing *events*, such as clicks. Here is a trait that focuses solely on observing `Clickable`:

```
// src/main/scala/progscala2/traits/ui2/ObservableClicks.scala
package progscala2.traits.ui2
import progscala2.traits.observer._

trait ObservableClicks extends Clickable with Subject[Clickable] {
  abstract override def click(): Unit = {        // ❶
    super.click()
    notifyObservers(this)
  }
}
```

❶

Note the `abstract override` keywords, discussed next.

The implementation is very similar to the previous `ObservableButton` example. The important difference is the `abstract` keyword. We had `override` before.

Look closely at this method. It calls `super.click()`, but what is `super` in this case? At this point, it could only appear to be `Clickable`, which *declares* but does not *define* the `click` method, or it could be `Subject`, which doesn't have a `click` method. So, `super` can't be bound to a real instance, at least not yet. This is why `abstract` is required here.

In fact, `super` will be bound when this trait is mixed into a concrete instance that defines the `click` method, such as `Button`. The `abstract` keyword tells the compiler (and the reader) that `click` is not yet fully implemented, even though `ObservableClicks.click` has a body.

Note

The `abstract` keyword is only required on a method in a trait when the method *has* a body, but it invokes another method in `super` that doesn't have a concrete implementation in parents of the trait.

Let's use this trait with `Button` and its concrete `click` method:

```
// src/main/scala/progscala2/traits/ui2/click-count-observer.sc
import progscala2.traits.ui2._
import progscala2.traits.observer._

// No override of "click" in Button
required.
                        "Click
val button = new Button(Me!"       ) with ObservableClicks

class ClickCountObserver extends Observer[Clickable] {
  var count = 0
  def receiveUpdate(state: Clickable): Unit = count += 1
}

val bco1 = new ClickCountObserver
val bco2 = new ClickCountObserver

button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

                         "bco1.count ${bco1.count} !=
assert(bco1.count == 5, s5"                              )
                         "bco2.count ${bco2.count} !=
assert(bco2.count == 5, s5"                              )
println("Success!")
```

Note that we can now declare a `Button` instance and mix in `ObservableClicks` without having to override the `click` method. Not only that, even when we don't want to observe clicks, we can construct GUI objects that support clicking by mixing in `Clickable`.

This fine-grained composition through mixin traits is quite powerful, but it can be overused:

- Lots of traits slow down compile times, because the compiler has to do more work to synthesize the output byte code.
- Library users can find a long list of traits intimidating when looking at code or Scaladocs.

Let's finish our example by adding a second trait. The JavaBeans Specification has the idea of "vetoable" events, where listeners for changes to a JavaBean can veto the change. Let's implement something similar with a trait that vetoes more than a set number of clicks. You could imagine implementing something similar to prevent users from accidentally clicking a button more than once that triggers a financial transaction. Here is our `VetoableClick` trait:

```
//
src/main/scala/progscala2/traits/ui2/VetoableClicks.scala
package progscala2.traits.ui2
import progscala2.traits.observer._

trait VetoableClicks extends Clickable {
// ❶
  // Default number of allowed
  clicks.
  val maxAllowed = 1
// ❷
  private var count = 0

  abstract override def click() = {
    if (count < maxAllowed) {
// ❸
      count += 1
      super.click()
    }
  }
}
```

❶

Also extends `Clickable`.

❷

The maximum number of allowed clicks. (A “reset” feature would be useful.)

❸

Once the number of clicks exceeds the allowed value (counting from zero), no further clicks are sent to `super`.

Note that `maxAllowed` is declared a `val` and the comment says it is the “default” value, which implies it can be changed. How can that be if it’s a `val`? The answer is that we can override the value in a class or trait that mixes in this trait. In this use of both traits, we reset the value to `2`:

```
// src/main/scala/progscala2/traits/ui2/vetoable-click-count-observer.sc
import progscala2.traits.ui2._
import progscala2.traits.observer._

// No override of "click" in Button
required.
val button =
                "Click
    new Button(Me!"       ) with ObservableClicks with VetoableClicks {
  override val maxAllowed = 2                                          // ❶
}

class ClickCountObserver extends Observer[Clickable] {                 // ❷
  var count = 0
  def receiveUpdate(state: Clickable): Unit = count += 1
}

val bco1 = new ClickCountObserver
val bco2 = new ClickCountObserver

button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

                         "bco1.count ${bco1.count} !=
assert(bco1.count == 2, s2"                                )       // ❸
                         "bco2.count ${bco2.count} !=
assert(bco2.count == 2, s2"                                )
println("Success!")
```

❶ Override the value of `maxAllowed` to `2`.

❷ Use the same `ClickObserver` as before.

❸ Note that the expected count is now `2`, even though the actual number of button clicks is `5`.

Try this experiment. Switch the order of the traits in the declaration of `button` to this:

```
                        "Click
val button = new Button(Me!"       ) with VetoableClicks with ObservableClicks
```

What happens when you run this code now?

The assertions should now fail, claiming that the count observed is actually `5`, not the expected `2`.

We have three versions of `click` wrapped like an onion. The question is which version of `click` gets called first when we mix in `VetoableClicks` and `ObservableClicks`? The answer is the declaration order determines the

order, from *right* to *left*.

Here is pseudocode for the effective call structure for each example:

```
// new Button("Click Me!") with VetoableClicks with
ObservableClicks

def ObservableClicks.click() = {
                                    // super.click =>
  if (count < maxAllowed) {         VetoableClicks.click
    count += 1
    {
                                    // super.click =>
      updateUI()                    Clickable.click
    }
  }
  notifyObservers(this)
}

// new Button("Click Me!") with ObservableClicks with
VetoableClicks

def VetoableClicks.click() = {
  if (count < maxAllowed) {
    count += 1
    {
// super.click =>
ObservableClicks.click
      {
                                    // super.click =>
        updateUI()                  Clickable.click
      }
      notifyObservers(this)
    }
  }
}
```

In the first case, when `ObservableClicks` is last in the declaration order, the actual `Clickable.click` method is still vetoed, but the observers are notified for all clicks. You could interpret this is a bug if observers are expecting notification only when clicks are not vetoed.

In the second case, when `VetoableClicks` is last in the declaration order, the veto logic wraps all other `click` methods, so `updateUI` *and* `notifyObservers` are both called only when the click isn't vetoed.

A algorithm called *linearization* is used to resolve the priority of traits and classes in the inheritance tree for a concrete class. These two versions of `button` are straightforward. The traits have priority order from right to left and the body of `button`, such as the version that overrides `maxAllowed`, is evaluated last.

We'll cover the full details of how linearization works for more complex declarations in Linearization of an Object's Hierarchy.

Constructing Traits

While the body of a trait functions as its primary constructor, traits don't support an argument list for the primary

constructor, nor can you define auxiliary constructors.

We saw in the last section's examples that traits can extend other traits. They can also extend classes. However, there is no mechanism for a trait to pass arguments to the parent class constructor, even literal values. Therefore, traits can only extend classes that have a no-argument primary or auxiliary constructor.

However, like classes, the body of a trait is executed every time an instance is created that uses the trait, as demonstrated by the following script:

```
// src/main/scala/progscala2/traits/trait-
construction.sc

trait T1 {
           "  in T1: x =
  println(s$x"             )
  val x=1
           "  in T1: x =
  println(s$x"             )
}

trait T2 {
           "  in T2: y =
  println(s$y"             )
  val y="T2"
           "  in T2: y =
  println(s$y"             )
}

class Base12 {
           "  in Base12: b =
  println(s$b"                 )
  val b="Base12"
           "  in Base12: b =
  println(s$b"                 )
}

class C12 extends Base12 with T1 with T2 {
           "  in C12: c =
  println(s$c"              )
  val c="C12"
           "  in C12: c =
  println(s$c"              )
}

println(s"Creating C12:")
new C12
         "After Creating
println(sC12"               )
```

Running this script yields the following output:

```
Creating C12:
  in Base12: b = null
  in Base12: b =
Base12
  in T1: x = 0
  in T1: x = 1
  in T2: y = null
  in T2: y = T2
  in C12: c = null
  in C12: c = C12
After Creating C12
```

Here we have a base class `Base12` for `C12`. It is evaluated first, then the traits `T1` and `T2` (i.e., *left* to *right* in declaration order), and finally the body of `C12`.

The order of the `println` statements for `T1` and `T2` seem to be reversed compared to our `Clickable` example. Actually, this is consistent. Here we are tracking initialization order, which goes left to right.

When we declared `button` with traits in the order `with VetoableClicks with ObservableClicks`, we first defined `click` as the method `VetoableClicks.click`. It called `super.click`, which wasn't resolved yet, at that moment. Then `ObservableClicks` was evaluated. It overrode `VetoableClicks.click` with its own version of `click`, but because it also called `super.click`, it will invoke `VetoableClicks.click`. Finally, because `ObservableClicks` extends `Clickable`, it provides a concrete `click` that `VetoableClicks.click` needs to call when it calls `super.click`.

So, while you can't pass construction parameters to traits, you can initialize fields, as well as methods and types, in the body of the trait. Subsequent traits or classes in the linearization can override these definitions. If a member is left abstract in the trait or abstract parent class where it is declared, a subsequent trait or class must define it. Concrete classes can't have any abstract members.

Tip

Avoid concrete fields in traits that can't be initialized to suitable default values. Use abstract fields instead, or convert the trait to a class with a constructor. Of course, stateless traits don't have any issues with initialization.

Class or Trait?

When considering whether a "concept" should be a trait or a class, keep in mind that traits as mixins make the most sense for "adjunct" behavior. If you find that a particular trait is used most often as a parent of other classes, so that the child classes *behave as* the parent trait, then consider defining the trait as a class instead, to make this logical relationship more clear. (We said *behaves as*, rather than *is a*, because the former is the more precise definition of inheritance, based on the *Liskov Substitution Principle*.)

It's a general principle of good object-oriented design that an instance should always be in a known valid state, starting from the moment the construction process finishes.

Recap and What's Next

In this chapter, we learned how to use traits to encapsulate and share cross-cutting concerns between classes. We covered when and how to use traits, how to "stack" multiple traits, and the rules for initializing values within traits.

In the next few chapters, we explore Scala’s object system and class hierarchy, with particular attention to the collections.