Table of Contents for Programming Scala, 2nd Edition

Chapter 11. The Scala Object System, Part II

We finish our discussion of Scala's object system by examining the rules for overriding members of classes and traits, including a discussion of the details of the *linearization* algorithm Scala uses for resolving member definitions and overrides in types that mix in traits and extend other types.

Overriding Members of Classes and Traits

Classes and traits can declare *abstract* members: *fields*, *methods*, and *types*. These members must be defined by a derived class or trait before an instance can be created. Most object-oriented languages support abstract methods, and some also support abstract fields and types.

Note

When overriding a concrete member, Scala requires the `override` keyword. It is optional when a subtype defines ("overrides") an abstract member. Conversely, it is an error to use `override` when you aren't actually overriding a member.

Requiring the `override` keyword has several benefits:

- It catches misspelled members that were intended to be overrides. The compiler will throw an error that the member doesn't override anything.
- It catches a subtle bug that can occur if a new member is added to a base class where the member's name collides with a preexisting member in a derived class, one which is unknown to the base class developer. That is, the derived-class member was never intended to override a base-class member. Because the derived-class member won't have the `override` keyword, the compiler will throw an error when the new base-class member is introduced.
- Having to add the keyword reminds you to consider what members should or should not be overridden.

Java has an optional `@Override` annotation for methods. It helps catch spelling errors, but it can't help with the subtle errors described in the second bullet item, because using the annotation is optional.

You can optionally use the `override` keyword when implementing an abstract member. Should you? Let's consider some arguments for and against.

Arguments in favor of always using the `override` keyword, in addition to those cited previously:

- It reminds the reader that a member defined in a parent class is being implemented (or overridden).
- If a parent class removes an abstract member that a child class defines, an error is reported.

Arguments against using the `override` keyword:

- Catching typos isn't actually necessary. A misdefined "override" means an undefined member still exists, so the derived class (or its concrete descendant classes) will fail to compile.
- If during the evolution of the code base, a developer maintaining the parent class abstract member decides to make it concrete, the change will go unnoticed when compiling the child class. Should the child class now call

the `super` version of the method? The compiler will silently accept what is now a complete override of the new parent-class implementation.

Avoid Overriding Concrete Members

In other words, the arguments are somewhat subtle as to whether or not it's a good practice to use `override` for abstract members. This raises a larger question about `override`: should you *ever* override a concrete method? The correct answer is *almost never*.

The relationship between parent and child types is a *contract* and care is required to ensure that a child class does not break the implemented behaviors specified by the parent type.

When overriding concrete members, it is very easy to break this contract. Should an override of the `foo` method call `super.foo`? If so, then when should it be called within the child's implementation? Of course, the answers are different in each case.

A far more robust *contract* is the *Template Method Pattern* described in the famous "Gang of Four" Design Patterns book. In this pattern, the parent class provides a concrete implementation of a method, defining the outline of the required behavior. The method calls `protected`, *abstract* methods at the points where polymorphic behavior is needed. Then, subclasses *only* implement the `protected`, abstract members.

Here is an example, a sketch of a payroll calculator for a company based in the US:

```
// src/main/scala/progscala2/objectsystem/overrides/payroll-template-
method.sc

case class Address(city: String, state: String, zip: String)
case class Employee(name: String, salary: Double, address: Address)

abstract class Payroll {
  def netPay(employee: Employee): Double = {                          // ❶
    val fedTaxes   = calcFedTaxes(employee.salary)
    val stateTaxes = calcStateTaxes(employee.salary, employee.address)
    employee.salary - fedTaxes -stateTaxes
  }

  def calcFedTaxes(salary: Double): Double                            // ❷
  def calcStateTaxes(salary: Double, address: Address): Double        // ❸
}

object Payroll2014 extends Payroll {
  val stateRate = Map(
    "XX" -> 0.05,
    "YY" -> 0.03,
    "ZZ" -> 0.0)

  def calcFedTaxes(salary: Double): Double = salary * 0.25            // ❹
  def calcStateTaxes(salary: Double, address: Address): Double = {
    // Assume the address.state is valid; it's found in the
    map!
    salary * stateRate(address.state)
  }
}

                    "Tom
val tom  = Employee(Jones"     , 100000.0, Address("MyTown", "XX", "12345"))
val jane = Employee("Jane Doe",  110000.0, Address("BigCity", "YY", "67890"))

                             // Result:
Payroll2014.netPay(tom)      70000.0
                             // Result:
Payroll2014.netPay(jane)     79200.0
```

❶

The `netPay` method uses the *Template Method Pattern*. It defines the protocol for calculating payroll, and delegates to abstract methods for details that change year over year, etc.

❷

The method for calculating federal taxes.

❸

The method for calculating state taxes.

❹

Concrete implementations of the abstract methods defined in the parent class.

Note that `override` is not used anywhere.

These days, when I see the `override` keyword in code, I see it as a potential *design smell*. Someone is overriding concrete behavior and subtle bugs might be lurking.

I can think of two exceptions to this rule. The first occurs when the parent-class implementation of a method does nothing useful. The `toString`, `equals`, and `hashCode` methods are examples. Unfortunately, overrides of the vacuous, default implementations of these methods are so ubiquitous that the practice has made us too comfortable with overriding concrete methods.

The second exception are those (rare?) occasions when you need to mix in non-overlapping behaviors. For example, you might override some critical method to add logging calls. In the child class override, when you invoke the logging calls versus the parent-class method, using `super` won't affect the external behavior (the contract) of the method, *as long as you correctly invoke the parent class method!*

Tip

Don't override concrete members when you can avoid it, except for trivial cases like `toString`. Don't use the `override` keyword unless you're actually overriding a concrete member.

Attempting to Override final Declarations

If a declaration includes the `final` keyword, overriding the declaration is prohibited. In the following example, the `fixedMethod` is declared `final` in the parent class. Attempting to compile the example will result in a compilation error:

```
// src/main/scala/progscala2/objectsystem/overrides/final-
member.scalaX
package progscala2.objectsystem.overrides

class NotFixed {
  final def fixedMethod = "fixed"
}

class Changeable2 extends NotFixed {
                            "not          // COMPILATION
  override def fixedMethod = fixed"       ERROR
}
```

This constraint applies to classes and traits as well as members. In this example, the class `Fixed` is declared `final`, so an attempt to derive a new type from it will also fail to compile:

```
// src/main/scala/progscala2/objectsystem/overrides/final-
class.scalaX
package progscala2.objectsystem.overrides

final class Fixed {
                    "Fixed did
  def doSomething = something!"
}

                                    // COMPILATION
class Changeable1 extends Fixed     ERROR
```

Note

Some of the types in the Scala library are final, including JDK classes like `String` and all the "value" types derived from `AnyVal` (see The Scala Type Hierarchy).

Overriding Abstract and Concrete Methods

For declarations that aren't final, let's examine the rules and behaviors for overriding, starting with methods.

Let's extend the `Widget` trait we introduced in Traits as Mixins with an abstract method `draw`, to support "rendering" the widget to a display, web page, etc. We'll also override a concrete method familiar to any Java programmer, `toString()`, using an ad hoc format.

Note

Drawing is actually a *cross-cutting concern*. The state of a `Widget` is one thing; how it is rendered on different platforms—"fat" clients, web pages, mobile devices, etc.—is a separate issue. So, drawing is a very good candidate for a trait, especially if you want your GUI abstractions to be portable. However, to keep things simple, we will handle drawing in the `Widget` hierarchy itself.

Here is the revised `Widget` with `draw` and `toString` methods. We'll now make it an `abstract class`, only because it is a logical parent for all UI widgets, like buttons. However, we could continue to define it as a trait:

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

abstract class Widget {
  def draw(): Unit
  override def toString() = "(widget)"
}
```

The `draw` method is abstract because it has no body. Therefore, `Widget` has to be declared `abstract`. Each concrete subclass of `Widget` will have to implement `draw` or rely on a parent class that implements it, of course. We don't need to return anything from `draw`, so its return value is `Unit`, although some sort of "success" status could be returned.

The `toString()` method is straightforward. Because `AnyRef` defines `toString`, the `override` keyword is required for `Widget.toString`.

Here is the revised `Button` class, with `draw` and `toString` methods:

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

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

  // Simple hack for demonstration
  purposes:
                              "Drawing:
  def draw(): Unit = println(s$this"          )

  // From Clickable:
                                          "$this clicked; updating
  protected def updateUI(): Unit = println(sUI"                          )

                             "(button: label=$label,
  override def toString() = s${super.toString()})"
}
```

It also mixes in the `Clickable` trait we introduced in Stackable Traits. We'll exploit it shortly.

We could make it a case class, but as we'll see, we're going to subclass it for other button types and we want to avoid the previously discussed issues with case class inheritance.

`Button` implements the abstract method `draw`. The `override` keyword is optional here. `Button` also overrides `toString`, which requires the `override` keyword. Note that `super.toString` is called.

Tip

Should you use the `override` keyword when implementing an abstract method? I don't think so. Suppose in the future that the maintainer of `Widget` decides to provide a default implementation of `draw`, perhaps to log all calls to it. Now implementers should truly overwrite `draw` *and* call `Widget.draw`. If you've been "overriding" `draw` all this time, the compiler will silently accept that you are now *really* overriding a concrete method and you may never know of the change. However, if you leave off the `override` keyword, your code will fail to compile when the abstract `draw` method suddenly has an implementation. You'll know of the change.

The `super` keyword is analogous to `this`, but it binds to the parent type, which is the aggregation of the parent class and any mixed-in traits. The search for `super.toString` will find the "closest" parent type `toString`, as determined by the linearization process we'll discuss later in this chapter (see Linearization of an Object's Hierarchy). In this case, because `Clickable` doesn't define `toString`, `Widget.toString` will be called. We are reusing `Clickable` from Stackable Traits.

Tip

Overriding a concrete method should be done rarely, because it is error-prone. Should you invoke the parent method? If so, when? Do you call it before doing anything else, or afterward? While the writer of the parent method *might* document the overriding constraints for the method, it's difficult to ensure that the writer of a derived class will honor those constraints. A much more robust approach is the *Template Method Pattern*.

Here is a simple script that exercises `Button`:

```
// src/main/scala/progscala2/objectsystem/ui/button.sc
import progscala2.objectsystem.ui.Button

val b = new Button("Submit")
// Result: b: oop.ui.Button = (button: label=Submit,
(widget))

b.draw()
// Result: Drawing: (button: label=Submit,
(widget))
```

Overriding Abstract and Concrete Fields

Let's discuss overriding fields in traits and classes separately, because traits have some particular issues.

Overriding fields in traits

Consider this contrived example that uses an undefined field before it is properly initialized:

```
// src/main/scala/progscala2/objectsystem/overrides/trait-invalid-init-
val.sc
// ERROR: "value" read before
initialized.

trait AbstractT2 {
          "In
  println(AbstractT2:"    )
  val value: Int
  val inverse = 1.0/value      // ❶
          "AbstractT2: value =         ", inverse =
  println("                    +value+"             +inverse)
}

val obj = new AbstractT2 {
          "In
  println(obj:"    )
  val value = 10
}
        "obj.value =             ", inverse =
println("            +obj.value+"            +obj.inverse)
```

❶ What is `value` when `inverse` is initialized?

Although it appears that we are creating an instance of the trait with `new AbstractT2 …`, we are actually using an anonymous class that implicitly extends the trait. Note the output that is produced if we run the script with the `scala` command (`$` is the shell prompt):

```
$ scala src/main/scala/progscala2/objectsystem/overrides/trait-bad-init-
val.sc
In AbstractT2:
AbstractT2: value = 0, inverse = Infinity
In obj:
obj.value = 10, inverse = Infinity
```

You get the results (with a few additional lines of output) if you use the REPL command `:load src/main/scala/progscala2/objectsystem/overrides/trait-bad-init-val.sc` or you paste the code into the REPL.

As you might expect, the `inverse` is calculated too early. Note that a divide-by-zero exception isn't thrown, but the compiler recognizes the value is infinite.

One way to detect this problem is to use the `scalac` compiler option `-Xcheckinit`, which will throw an exception when a field is referenced before it is initialized.

Scala provides two solutions to this problem. The first is *lazy values*, which we discussed in lazy val:

```
// src/main/scala/progscala2/objectsystem/overrides/trait-lazy-init-
val.sc

trait AbstractT2 {
          "In
  println(AbstractT2:"    )
  val value: Int
  lazy val inverse = 1.0/value      // ❶
  // println("AbstractT2: value = "+value+", inverse =
  "+inverse)
}

val obj = new AbstractT2 {
          "In
  println(obj:"    )
  val value = 10
}
        "obj.value =             ", inverse =
println("             +obj.value+"             +obj.inverse)
```

❶

Added the keyword `lazy` *and* commented out the `println` statement.

Now `inverse` is initialized to a valid value:

```
In AbstractT2:
In obj:
obj.value = 10, inverse =
0.1
```

However, `lazy` only helps if the `println` statement is not used. If you remove the `//` and run it, you'll get

`Infinity` again, because `lazy` only defers evaluation until the value is used. The `println` statement forces evaluation too soon.

Tip

If a `val` is `lazy`, make sure all uses of the `val` are also as `lazy` as possible.

The second solution, which is less commonly used, is *pre-initialized fields*. Consider this refined implementation:

```
// src/main/scala/progscala2/objectsystem/overrides/trait-pre-init-
val.sc

trait AbstractT2 {
          "In
  println(AbstractT2:"    )
  val value: Int
  val inverse = 1.0/value
          "AbstractT2: value =         ", inverse =
  println("                     +value+"             +inverse)
}

val obj = new {
  // println("In obj:")      // ❶
  val value = 10
} with AbstractT2

        "obj.value =             ", inverse =
println("             +obj.value+"             +obj.inverse)
```

❶

> Only type definitions and concrete field definitions are allowed in pre-initialization blocks. It would be a compilation error to use the `println` statement here, for example.

We instantiate an anonymous inner class, initializing the `value` field in the block, before the `with AbstractT2` clause. This guarantees that `value` is initialized before the body of `AbstractT2` is executed, as shown when you run the script:

```
In AbstractT2:
AbstractT2: value = 10, inverse =
0.1
obj.value = 10, inverse = 0.1
```

Even within the body of the trait, `inverse` is properly initialized.

Now let's consider the `VetoableClicks` trait we used in Stackable Traits. It defines a `val` named `maxAllowed` and initializes it to `1`. We would like the ability to override the value in a class that mixes in this trait. Here it is again:

```
//
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`.

It should be straightforward to implement an instance that mixes in this trait and overrides `maxAllowed` if we want. However, there are initialization issues we should review first.

To see those issues, let’s first return to `VetoableClicks` and use it with `Button`. To see what happens, we’ll also need to mix in the `ObservableClicks` trait that we also discussed in Stackable Traits:

```
// 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 in Stackable Traits.

Here is a test script:

```
// src/main/scala/progscala2/objectsystem/ui/vetoable-clicks.sc
import progscala2.objectsystem.ui.Button
import progscala2.traits.ui2.{Clickable, ObservableClicks, VetoableClicks}
import progscala2.traits.observer._

val observableButton =
// ❶
  new Button("Okay") with ObservableClicks with VetoableClicks {
    override val maxAllowed: Int = 2
// ❷
  }

assert(observableButton.maxAllowed == 2,                                    //
❸
   "maxAllowed =
  s${observableButton.maxAllowed}"                )

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

val clickCountObserver = new ClickCountObserver                             //
❺
observableButton.addObserver(clickCountObserver)

val n = 5
for (i <- 1 to n) observableButton.click()                                  //
❻

assert(clickCountObserver.count == 2,
// ❼
   "count = ${clickCountObserver.count}. Should be !=
  s$n"                                                  )
```

❶

Construct an observable button by mixing in the required traits.

❷

The main point of this exercise is to override a `val`. Note that `override` and the full declaration of `maxAllowed` is required.

❸

Verify that we successfully changed `maxAllowed`.

❹

Define an observer to track the number of clicks that reach it.

❺

Instantiate an observer instance and “register” it with the button *Subject*.

❻

Click the button five times.

❼

Verify that the observer only saw two clicks; the other three were vetoed.

Recall that the mixin order of traits determines priority order, a subject that we’ll finish exploring later in this chapter, in Linearization of an Object’s Hierarchy.

Try switching the order of `ObservableClicks` and `VetoableClicks` in the line after the label ❶. What do you expect to happen now? You should see the final assertion test fail with a count of five instead of two. Why? Because `ObservableClicks` will now see each click *before* `VetoableClicks` sees it. In other words, `VetoableClicks` is now effectively doing nothing.

So, we see that we can override immutable field definitions. What if you want more dynamic control over `maxAllowed`, where it might vary during program execution? You can declare the field to be a mutable variable with `var`, then the declaration of `observableButton` changes to the following:

```
val observableButton =
  new Button("Okay") with ObservableClicks with VetoableClicks {
    maxAllowed = 2
  }
```

The previous `override` keyword with the full signature is no longer necessary.

I should mention that for logical consistency, you would need to decide what changing `maxAllowed` should mean for the state of the observer. If `maxAllowed` is decreased and the observer has already counted a larger number of clicks, should you add a mechanism to decrease the observer’s count?

Now we can discuss the initialization issues we mentioned earlier. Because the body of the trait is executed before the body of the class using it, reassigning the field value happens *after* the initial assignment in the trait’s body. Recall our previous pathological example of `inverse` using `value` before it was set. For `VetoableObserver`, suppose it initialized some sort of private array to save up to `maxAllowed` updates. The final assignment to `maxAllowed` would leave the object in an inconsistent state! You would need to avoid this problem manually, such as deferring the storage allocation until it’s needed for the first updates, well after the initialization process has completed. Declaring `maxAllowed` as a `val` doesn’t eliminate this problem, although it does signal to users of the type that `VetoableClicks` makes assumptions about the state of the instance, namely that this part of the state won’t change. Still, if you are the maintainer of `VetoableClicks`, you’ll have to remember that users might override the value of `maxAllowed`, whether or not it is declared immutable!

Tip

Avoid `var` fields when possible (in classes as well as traits). Consider public `var` fields especially risky.

However, `val`s don’t offer complete protection. A `val` in a trait can also be overridden during initialization of subclass instances, although it will remain immutable afterwards.

Overriding fields in classes

For members declared in classes, the behavior is essentially the same as for traits. For completeness, here is an example with both a `val` override and a `var` reassignment in a derived class:

```
// src/main/scala/progscala2/objectsystem/overrides/class-
field.sc

class C1 {
  val name = "C1"
  var count = 0
}

class ClassWithC1 extends C1 {
  override val name = "ClassWithC1"
  count = 1
}

val c = new ClassWithC1()
println(c.name)
println(c.count)
```

The `override` keyword is required for the *concrete* `val` field `name`, but not for the `var` field `count`. This is because we are changing the initialization of a constant (`val`), which is a "special" operation.

If you run this script, the output is the following:

```
ClassWithC1
1
```

Both fields are overridden in the derived class, as expected. Here is the same example modified so that both the `val` and the `var` are abstract in the base class:

```
// src/main/scala/progscala2/objectsystem/overrides/class-abs-
field.sc

abstract class AbstractC1 {
  val name: String
  var count: Int
}

class ClassWithAbstractC1 extends AbstractC1 {
  val name = "ClassWithAbstractC1"
  var count = 1
}

val c = new ClassWithAbstractC1()
println(c.name)
println(c.count)
```

The `override` keyword is not required for `name` in `ClassWithAbstractC1`, because the original declaration is abstract. The output of this script is the following:

```
ClassWithAbstractC1
1
```

It's important to emphasize that `name` and `count` are *abstract* fields, not concrete fields with default values. A similar-looking declaration of `name` in a Java class, `String name`, would declare a *concrete* field with the *default* value, `null` in this case. Java doesn't support abstract fields, only methods.

Overriding Abstract Types

We introduced abstract type declarations in Abstract Types Versus Parameterized Types, which Java doesn't support. Recall the `BulkReader` example from that section:

```
abstract class BulkReader {
  type In
  val source: In
  def read: String  // Read source and return a String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}
...
```

The example shows how to declare an abstract type and how to define a concrete value in derived classes. `BulkReader` declares `type In` without initializing it. The concrete derived class `StringBulkReader` provides a concrete value using `type In = String`.

Unlike fields and methods, it is not possible to override a concrete `type` definition.

When Accessor Methods and Fields Are Indistinguishable: The Uniform Access Principle

Let's take one more look at `maxAllowed` in `VetoableClick` and discuss an interesting implication of mixing inheritance and the *Uniform Access Principle*, which we learned about in The Uniform Access Principle.

Here is a new version of `VetoableClick`, called `VetoableClickUAP` ("UAP" for uniform access principle) in a script that uses it:

```
// src/main/scala/progscala2/objectsystem/ui/vetoable-clicks-uap.sc
import progscala2.objectsystem.ui.Button
import progscala2.traits.ui2.{Clickable, ObservableClicks, VetoableClicks}
import progscala2.traits.observer._

trait VetoableClicksUAP extends Clickable {

  def maxAllowed: Int = 1                          // ❶

  private var count = 0

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

val observableButton =
  new Button("Okay") with ObservableClicks with VetoableClicksUAP {
    override val maxAllowed: Int = 2               // ❷
  }

assert(observableButton.maxAllowed == 2,
   "maxAllowed =
  s${observableButton.maxAllowed}"                )

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

val clickCountObserver = new ClickCountObserver
observableButton.addObserver(clickCountObserver)

val n = 5
for (i <- 1 to n) observableButton.click()

assert(clickCountObserver.count == 2,
   "count = ${clickCountObserver.count}. Should be !=
  s$n"                                                )
```

❶ `maxAllowed` is now a method that returns the default value of `1`.

❷ Instead of overriding the method, use a value (`val`) definition.

The `override` keyword is required because the original method is defined. If the method is abstract in the trait, the `override` keyword is not required.

The output is the same as before, but we exploited the Uniform Access Principle to override the method definition

with a value. Why is this allowed?

Using a method declaration supports the freedom to return a different value each time we call it, as long as the implementation does the necessary work. However, the declaration is consistent with the case where one and only one value is ever returned. Of course, this is preferred in functional programming, anyway. Ideally, a no-argument method should always return the same value in a pure functional world.

So, when we replace the method call with a value, we are exploiting *referential transparency* and not violating any rules about how the "method" implementation should behave.

For this reason, it's a common practice in Scala libraries for traits to declare no-argument methods instead of field values. Think of these methods as *property readers*, if you like. That gives implementers of types that use the trait complete freedom to provide an implementation for the method, perhaps to defer expensive initialization until necessary, or to simply use a value for implementation.

Overriding a method with a value in a subclass can also be handy when interoperating with Java code. For example, you can override a getter as a `val` by placing it in the constructor.

Consider the following example, in which a Scala `Person` class implements a hypothetical `PersonInterface` from some legacy Java library:

```
class Person(val getName: String) extends PersonInterface
```

If you only need to override a few accessors in the Java code, this technique makes quick work of them.

What about overriding a parameterless method with a `var`, or overriding a `val` or `var` with a method? These are not permitted because they can't match the behaviors of the things they are overriding.

If you attempt to use a `var` to override a parameterless method, you get an error that the writer method, `override def name: name_=`, is not overriding anything. For example, if an abstract method, `String`, is declared in a trait and an implementing subclass attempts to use `override val name = "foo"`, this would be equivalent to overriding *two* methods, the original and `def name_=(…)`, but there is no such method.

If you could override a `val` with a method, there would be no way for Scala to guarantee that the method always returns the same value, consistent with `val` semantics.

Linearization of an Object's Hierarchy

Because of single inheritance, if we ignored mixed-in traits, the inheritance hierarchy would be linear, one ancestor after another. When traits are considered, each of which may be derived from other traits and classes, the inheritance hierarchy forms a directed, acyclic graph.

The term *linearization* refers to the algorithm used to "flatten" this graph for the purposes of resolving method lookup priorities, constructor invocation order, binding of `super`, etc.

Informally, we saw in Stackable Traits that when an instance has more than one trait, they bind right to left, as declared. Consider the following example that demonstrates this straightforward linearization:

```
//
src/main/scala/progscala2/objectsystem/linearization/linearization1.sc

class C1 {
                "C1
  def m = print("    )
}

trait T1 extends C1 {
                           "T1
  override def m = { print("    ); super.m }
}

trait T2 extends C1 {
                           "T2
  override def m = { print("    ); super.m }
}

trait T3 extends C1 {
                           "T3
  override def m = { print("    ); super.m }
}

class C2 extends T1 with T2 with T3 {
                           "C2
  override def m = { print("    ); super.m }
}

val c2 = new C2
c2.m
```

Running this script yields the following output:

```
C2 T3 T2 T1
C1
```

So, the `m` methods in the traits are called in the right-to-left order of the declaration of the traits. We'll also see why C1 is at the end of the list in a moment.

Next, let's see what the invocation sequence of the constructors looks like:

```
//
src/main/scala/progscala2/objectsystem/linearization/linearization2.sc

class C1 {
        "C1
  print("    )
}

trait T1 extends C1 {
        "T1
  print("    )
}

trait T2 extends C1 {
        "T2
  print("    )
}

trait T3 extends C1 {
        "T3
  print("    )
}

class C2 extends T1 with T2 with T3 {
          "C2
  println("    )
}

val c2 = new C2
```

Running this script yields this output:

```
C1 T1 T2 T3
C2
```

So, the construction sequence is the reverse. This invocation order makes sense when you consider that the parent types need to be constructed before the derived types, because a derived type often uses fields and methods in the parent types during its construction process.

The output of the first linearization script is actually missing two types at the end. The full linearization for reference types actually ends with `AnyRef` and `Any`. So the linearization for `C2` is actually the following:

```
C2 T3 T2 T1 C1 AnyRef
Any
```

Prior to Scala 2.10, there was also a marker trait, `ScalaObject`, inserted in the hierarchy before `AnyRef`. Our output doesn't show `AnyRef` and `Any` because they don't have the `print` statements we used, of course.

In contrast, the value types, which subclass `AnyVal`, are all declared `final abstract`. The compiler manages instantiation of them. Because we can't subclass them, their linearizations are simple and straightforward.

What about the new value classes? Let's use a modified version of the `USPhoneNumber` we saw previously, where we have added the same `m` method we used earlier. The rules for value classes don't allow us to add the same `print` statements in the type bodies:

```
// src/main/scala/progscala2/basicoop/ValueClassPhoneNumber.sc

class USPhoneNumber(val s: String) extends AnyVal {

  override def toString = {
    val digs = digits(s)
    val areaCode  = digs.substring(0,3)
    val exchange  = digs.substring(3,6)
                                          // "subscriber
    val subnumber = digs.substring(6,10)  number"
     "($areaCode) $exchange-
    s$subnumber"
  }

  private def digits(str: String): String = str.replaceAll("""\D""", "")
}

val number = new USPhoneNumber("987-654-3210")
// Result: number: USPhoneNumber = (987) 654-
3210
```

It prints the following when `m` is called:

```
USPhoneNumber Formatter Digitizer
M
```

The output is consistent to what we saw for the `C*` class hierarchy. However, notice that the `M` trait is mixed into several other traits. Why does `M` show up last in the output, meaning its `m` method is the last in the lookup order? Let's examine linearization more closely.

We'll use our `C*` classes. All the classes and traits define the method `m`. The one in `C2` is called first, because the instance is of that type. `C2.m` calls `super.m`, which resolves to `T3.m`. The search appears to be *breadth-first*, rather than *depth-first*. If it were depth-first, it would invoke `C1.m` after `T3.m`. Afterward, `T3.m`, `T2.m`, then `T1.m`, and finally `C1.m` are invoked. `C1` is the parent of the three traits. From which of the traits did we traverse to `C1`? Actually, it is breadth-first, with "delayed" evaluation, as we will see. Let's modify our first example and see more explicitly how we got to `C1`:

```
//
src/main/scala/progscala2/objectsystem/linearization/linearization3.sc

class C1 {
  def m(previous: String) = print(s"C1($previous)")
}

trait T1 extends C1 {
  override def m(p: String) = { super.m(s"T1($p)") }
}

trait T2 extends C1 {
  override def m(p: String) = { super.m(s"T2($p)") }
}

trait T3 extends C1 {
  override def m(p: String) = { super.m(s"T3($p)") }
}

class C2 extends T1 with T2 with T3 {
  override def m(p: String) = { super.m(s"C2($p)") }
}

val c2 = new C2
c2.m("")
```

Now we pass the name of the caller of `super.m` as a parameter, then `C1` prints out who called it. Running this script yields the following output:

```
C1(T1(T2(T3(C2()))))
```

Here is the actual algorithm for calculating the linearization. A more formal definition is given in *The Scala Language Specification*.

This explains how we got to `C1` from `T1` in the previous example. `T3` and `T2` also have it in their linearizations, but they come before `T1`, so the `C1` terms they contributed were deleted. Similarly, the `M` trait in the `USPhoneNumber` example ends up at the righthand side of the list for the same reason.

Let's work through the algorithm using a slightly more involved example:

```
//
src/main/scala/progscala2/objectsystem/linearization/linearization4.sc

class C1 {
                "C1
  def m = print("    )
}

trait T1 extends C1 {
                           "T1
  override def m = { print("    ); super.m }
}

trait T2 extends C1 {
                           "T2
  override def m = { print("    ); super.m }
}

trait T3 extends C1 {
                           "T3
  override def m = { print("    ); super.m }
}

class C2A extends T2 {
                           "C2A
  override def m = { print("      ); super.m }
}

class C2 extends C2A with T1 with T2 with T3 {
                           "C2
  override def m = { print("    ); super.m }
}

def calcLinearization(obj: C1, name: String) = {
         "$name:
  print(s"        )
  obj.m
       "AnyRef
  print("        )
  println("Any")
}

                          "C2
calcLinearization(new C2, "    )
println("")
                             "T3
calcLinearization(new T3 {}, "    )
                             "T2
calcLinearization(new T2 {}, "    )
                             "T1
calcLinearization(new T1 {}, "    )
calcLinearization(new C2A, "C2A")
                          "C1
calcLinearization(new C1, "    )
```

The output is the following:

```
C2 : C2 T3 T1 C2A T2 C1 AnyRef
Any

T3 : T3 C1 AnyRef Any
T2 : T2 C1 AnyRef Any
T1 : T1 C1 AnyRef Any
C2A: C2A T2 C1 AnyRef Any
C1 : C1 AnyRef Any
```

To help us along, we calculated the linearizations for the other types, and we also appended `AnyRef` and `Any` to remind ourselves that they should also be there.

So, let's work through the algorithm for `C2` and confirm our results. We'll suppress the `AnyRef` and `Any` for clarity, until the end. See Table 11-1.

Table 11-1. Hand calculation of C2 linearization: C2 extends C2A with T1 with T2 with T3 {…}

#	Linearization	Description
1	`C2`	Add the type of the instance.
2	`C2, T3, C1`	Add the linearization for `T3` (farthest on the right).
3	`C2, T3, C1, T2, C1`	Add the linearization for `T2`.
4	`C2, T3, C1, T2, C1, T1, C1`	Add the linearization for `T1`.
5	`C2, T3, C1, T2, C1, T1, C1, C2A, T2, C1`	Add the linearization for `C2A`.
6	`C2, T3, T2, T1, C2A, T2, C1`	Remove duplicates of `C1`; all but the *last* `C1`.
7	`C2, T3, T1, C2A, T2, C1`	Remove duplicates of `T2`; all but the *last* `T2`.
8	`C2, T3, T1, C2A, T2, C1, AnyRef, Any`	Finish!

What the algorithm does is push any shared types to the right until they come after *all* the types that derive from them.

Try modifying the last script with different hierarchies and see if you can reproduce the results using the algorithm.

Tip

Overly complex type hierarchies can result in method lookup "surprises." If you have to work through this algorithm to figure out what's going on, try to simplify your code.

Recap and What's Next

We explored the fine points of overriding members in derived types, including the ability to override (or implement abstract) no-argument methods with values. Finally, we walked through the details of Scala's linearization algorithm for member lookup resolution.

In the next chapter, we'll learn about Scala's collections library.