Chapter 13. Visibility Rules

Scala goes well beyond Java's four-tiered visibility rules: *public*, *protected*, *private*, and default package visibility. Visibility rules in object-oriented languages are designed to expose only the essential public abstraction of a type and encapsulate implementation information, hiding it from view.

This chapter covers the details in depth and to be honest, it's dry stuff. Superficially, they are just like Java's rules with the exception that *public* visibility is the default in Scala. Scala adds more sophisticated scoping rules, but you don't see them used a lot in every-day Scala code. So, consider skimming the first few sections of this chapter and the concluding section, Final Thoughts on Visibility. Save the rest of the chapter for when you need to know the particulars, especially when you start writing your own libraries.

Public Visibility: The Default

Unlike Java, but more like many other object-oriented languages, Scala uses *public* visibility by default. It's common to declare type members private, when you want to limit their visibility to the type only, or to declare them protected to limit visbility to subclasses. But Scala gives you more options. This chapter covers the details.

You'll want to use public visibility for anything that users of your objects should see and use. Keep in mind that the set of publicly visible members form the abstraction exposed by the type, along with the type's name itself.

Tip

The art of good object-oriented design includes defining minimal, clear, and cohesive public abstractions.

The conventional wisdom in object-oriented design is that fields should be private or protected. If access is required, it should happen through methods, but not everything should be accessible by default.

There are two reasons for this convention. The first is to prevent users from making modifications to mutable fields outside your control. However, using immutable values eliminates this concern. The second reason is that a particular field might be part of the implementation and not part of the public abstraction that you want to expose.

When access makes sense, the virtue of the *Uniform Access Principle* (see The Uniform Access Principle) is that we can give the user the semantics of public field access, but use either a method or actual direct access to the field, whichever is appropriate for the task. The user doesn't need to know which implementation is used. We can even change the implementation without forcing code changes on the user, although recompilation will be necessary.

There are two kinds of "users" of a type: derived types, and code that works with instances of the type. Derived types usually need more access to the members of their parent types than users of instances do.

Scala's visibility rules are similar to Java's, but tend to be both more consistently applied and more flexible. For example, in Java, if an inner class has a `private` member, the enclosing class can see it. In Scala, the enclosing class can't see a `private` member, but Scala provides another way to declare it visible to the enclosing class.

Visibility Keywords

As in Java and C#, the keywords that modify visibility, such as `private` and `protected`, appear at the beginning of declarations. You'll find them before the `class` or `trait` keywords for types, before the `val` or `var` for fields, and before the `def` for methods.

Note

You can also use an access modifier keyword on the primary constructor of a class. Put it after the type name and type parameters, if any, and before the argument list, as in this example:

```
class Restricted[+A] private (name: String)
{...}
```

.

Why do this? It forces users to call a factory method instead of instantiating types directly.

Table 13-1 summarizes the visibility scopes.

Table 13-1. Visibility scopes

Name	Keyword	Description
public	*none*	Public members and types are visible everywhere, across all boundaries.
protected	`protected`	Protected members are visible to the defining type, to derived types, and to nested types. Protected types are visible only within the same package and subpackages.
private	`private`	Private members are visible only within the defining type and nested types. Private types are visible only within the same package.
scoped protected	`protected[scope]`	Visibility is limited to `scope`, which can be a package, type, or `this` (meaning the same instance, when applied to members, or the enclosing package, when applied to types). See the following text for details.
scoped private	`private[scope]`	Synonymous with scoped protected visibility, except under inheritance (discussed in the following text).

Let's explore these visibility options in more detail. To keep things simple, we'll use fields for member examples. Method and type declarations behave the same way.

Note

Unfortunately, you can't apply any of the visibility modifiers to packages. Therefore, a package is always public, even when it contains no publicly visible types.

Public Visibility

Any declaration without a visibility keyword is "public," meaning it is visible everywhere. There is no `public` keyword in Scala. This is in contrast to Java, which defaults to public visibility only within the enclosing package (i.e., "package private"). Other object-oriented languages, like Ruby, also default to public visibility:

```
//
src/main/scala/progscala2/visibility/public.scala

package scopeA {
  class PublicClass1 {
    val publicField = 1

    class Nested {
      val nestedField = 1
    }

    val nested = new Nested
  }

  class PublicClass2 extends PublicClass1 {
    val field2  = publicField + 1
    val nField2 = new Nested().nestedField
  }
}

package scopeB {
  class PublicClass1B extends scopeA.PublicClass1

  class UsingClass(val publicClass: scopeA.PublicClass1) {
    def method = "UsingClass:" +
      " field:
      "          + publicClass.publicField +
      " nested field:
      "                 + publicClass.nested.nestedField
  }
}
```

Everything is public in these packages and classes. Note that `scopeB.UsingClass` can access `scopeA.PublicClass1` and its members, including the instance of `Nested` and its public field.

Protected Visibility

Protected visibility is for the benefit of implementers of derived types, who need a little more access to the details of their parent types. Any member declared with the `protected` keyword is visible only to the defining type, including other instances of the same type and any derived types. When applied to a type, `protected` limits visibility to the enclosing package.

Java, in contrast, makes protected members visible throughout the enclosing package. Scala handles this case with scoped private and protected access:

```
// src/main/scala/progscala2/visibility/protected.scalaX
// WON'T
COMPILE

package scopeA {
  class ProtectedClass1(protected val protectedField1: Int) {
    protected val protectedField2 = 1

    def equalFields(other: ProtectedClass1) =
      (protectedField1 == other.protectedField1) &&
      (protectedField1 == other.protectedField1) &&
      (nested == other.nested)

    class Nested {
      protected val nestedField = 1
    }

    protected val nested = new Nested
  }

  class ProtectedClass2 extends ProtectedClass1(1) {
    val field1 = protectedField1
    val field2 = protectedField2
    val nField = new Nested().nestedField  // ERROR
  }

  class ProtectedClass3 {
    val protectedClass1 = new ProtectedClass1(1)
    val protectedField1 = protectedClass1.protectedField1 // ERROR
    val protectedField2 = protectedClass1.protectedField2 // ERROR
    val protectedNField = protectedClass1.nested.nestedField
// ERROR
  }

  protected class ProtectedClass4

  class ProtectedClass5 extends ProtectedClass4
  protected class ProtectedClass6 extends ProtectedClass4
}

package scopeB {
  class ProtectedClass4B extends scopeA.ProtectedClass4 // ERROR
}
```

When you compile this file with `scalac`, you get five errors like the following, corresponding to the lines with the `// ERROR` comment:

```
.../visibility/protected.scalaX:23: error: value nestedField in class
Nested cannot be accessed in ProtectedClass2.this.Nested
 Access to protected value nestedField not permitted because
 enclosing class ProtectedClass2 in package scopeA is not a subclass
of
 class Nested in class ProtectedClass1 where target is defined
    val nField = new Nested().nestedField  // ERROR
                              ^
...
5 errors found
```

`ProtectedClass2` can access protected members of `ProtectedClass1`, because it derives from it. However, it can't access the protected `nestedField` in `protectedClass1.nested`. Also, `ProtectedClass3` can't access protected members of the `ProtectedClass1` instance it uses.

Finally, because `ProtectedClass4` is declared `protected`, it is not visible in the `scopeB` package.

Private Visibility

Private visibility completely hides implementation details, even from the implementers of derived classes. Any member declared with the `private` keyword is visible only to the defining type, including other instances of the same type. When applied to a type, `private` limits visibility to the enclosing package:

```
// src/main/scala/progscala2/visibility/private.scalaX
// WON'T
COMPILE

package scopeA {
  class PrivateClass1(private val privateField1: Int) {
    private val privateField2 = 1

    def equalFields(other: PrivateClass1) =
      (privateField1 == other.privateField1) &&
      (privateField2 == other.privateField2) &&
      (nested == other.nested)

    class Nested {
      private val nestedField = 1
    }

    private val nested = new Nested
  }

  class PrivateClass2 extends PrivateClass1(1) {
    val field1 = privateField1  // ERROR
    val field2 = privateField2  // ERROR
    val nField = new Nested().nestedField // ERROR
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1(1)
    val privateField1 = privateClass1.privateField1 // ERROR
    val privateField2 = privateClass1.privateField2 // ERROR
    val privateNField = privateClass1.nested.nestedField
// ERROR
  }

  private class PrivateClass4

  class PrivateClass5 extends PrivateClass4  // ERROR
  protected class PrivateClass6 extends PrivateClass4 // ERROR
  private class PrivateClass7 extends PrivateClass4
}

package scopeB {
  class PrivateClass4B extends scopeA.PrivateClass4  // ERROR
}
```

Compiling this file produces nine errors for the lines marked as errors.

Now, `PrivateClass2` can’t access private members of its parent class `PrivateClass1`. They are completely invisible to the subclass, as indicated by the error messages. Nor can it access a private field in a `Nested` class.

Just as for the case of `protected` access, `PrivateClass3` can’t access private members of the `PrivateClass1` instance it is using. Note, however, that the `equalFields` method can access private members of the `other` instance.

The declarations of `PrivateClass5` and `PrivateClass6` fail because, if allowed, they would enable `PrivateClass4` to “escape its defining scope.” However, the declaration of `PrivateClass7` succeeds because it is also declared to be private. Curiously, our previous example was able to declare a public class that subclassed a protected class without a similar error.

Finally, just as for `protected` type declarations, the `private` types can’t be subclassed outside the same package.

Scoped Private and Protected Visibility

Scala goes beyond most languages with an additional way of fine-tuning the scope of visibility; *scoped* `private` and `protected` visibility declarations. Note that using `private` or `protected` in a scoped declaration is interchangeable, because they behave identically, except under inheritance when applied to members.

Tip

Although they behave nearly the same, it is a little more common to see `private[X]` rather than `protected[X]` used in Scala libraries. It’s interesting to note that in the first version of this book, we noted that the Scala 2.7.X library used `private[X]` roughly five times more often than `protected[X]`. In Scala 2.11, the ratio is much closer, 5/3.

Let’s begin by considering the only differences in behavior between scoped private and scoped protected—how they behave under inheritance when members have these scopes:

```
// src/main/scala/progscala2/visibility/scope-inheritance.scalaX
// WON'T
COMPILE

package scopeA {
  class Class1 {
    private[scopeA]   val scopeA_privateField = 1
    protected[scopeA] val scopeA_protectedField = 2
    private[Class1]   val class1_privateField = 3
    protected[Class1] val class1_protectedField = 4
    private[this]     val this_privateField = 5
    protected[this]   val this_protectedField = 6
  }

  class Class2 extends Class1 {
    val field1 = scopeA_privateField
    val field2 = scopeA_protectedField
    val field3 = class1_privateField     // ERROR
    val field4 = class1_protectedField
    val field5 = this_privateField       // ERROR
    val field6 = this_protectedField
  }
}

package scopeB {
  class Class2B extends scopeA.Class1 {
    val field1 = scopeA_privateField     // ERROR
    val field2 = scopeA_protectedField
    val field3 = class1_privateField     // ERROR
    val field4 = class1_protectedField
    val field5 = this_privateField       // ERROR
    val field6 = this_protectedField
  }
}
```

This file produces five compilation errors.

The first two errors, inside `Class2`, show us that a derived class inside the same package can’t reference a member that is scoped private to the parent class or `this`, but it can reference a private member scoped to the package (or type) that encloses both `Class1` and `Class2`.

In contrast, for a derived class outside the same package, it has no access to any of the scoped private members of `Class1`.

However, all the scoped protected members are visible in both derived classes.

We’ll use scoped private declarations for the rest of our examples and discussion, because use of scoped private is a little more common in the Scala library than scoped protected, when the previous inheritance scenarios aren’t a factor.

First, let’s start with the most restrictive visibility, `private[this]`, because it affects type members:

```
// src/main/scala/progscala2/visibility/private-this.scalaX
// WON'T
COMPILE

package scopeA {
  class PrivateClass1(private[this] val privateField1: Int) {
    private[this] val privateField2 = 1

    def equalFields(other: PrivateClass1) =
      (privateField1 == other.privateField1) && // ERROR
      (privateField2 == other.privateField2) && // ERROR
      (nested == other.nested) // ERROR

    class Nested {
      private[this] val nestedField = 1
    }

    private[this] val nested = new Nested
  }

  class PrivateClass2 extends PrivateClass1(1) {
    val field1 = privateField1  // ERROR
    val field2 = privateField2  // ERROR
    val nField = new Nested().nestedField  // ERROR
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1(1)
    val privateField1 = privateClass1.privateField1  // ERROR
    val privateField2 = privateClass1.privateField2  // ERROR
    val privateNField = privateClass1.nested.nestedField
// ERROR
  }
}
```

Nine errors are reported by the compiler.

Lines 10 and 11 also won't parse. Because they are part of the expression that started on line 9, the compiler stopped after the first error.

The `private[this]` members are only visible to the same instance. An instance of the same class can't see `private[this]` members of another instance, so the `equalFields` method won't parse.

Otherwise, the visibility of class members is the same as `private` without a scope specifier.

When declaring a type with `private[this]`, use of `this` effectively binds to the enclosing package, as shown here:

```
// src/main/scala/progscala2/visibility/private-this-pkg.scalaX
// WON'T
COMPILE

package scopeA {
  private[this] class PrivateClass1

  package scopeA2 {
    private[this] class PrivateClass2
  }

  class PrivateClass3 extends PrivateClass1  // ERROR
  protected class PrivateClass4 extends PrivateClass1 // ERROR
  private class PrivateClass5 extends PrivateClass1
  private[this] class PrivateClass6 extends PrivateClass1

  private[this] class PrivateClass7 extends scopeA2.PrivateClass2 // ERROR
}

package scopeB {
  class PrivateClass1B extends scopeA.PrivateClass1 // ERROR
}
```

This produces four errors.

In the same package, attempting to declare a `public` or `protected` subclass fails. Only `private` and `private[this]` subclasses are allowed. Also, `PrivateClass2` is scoped to `scopeA2`, so you can't declare it outside `scopeA2`. Similarly, an attempt to declare a class in unrelated `scopeB` using `PrivateClass1` also fails.

Hence, when applied to types, `private[this]` is equivalent to Java's `package private` visibility.

Next, let's examine type-level visibility, `private[T]`, where `T` is a type:

```
// src/main/scala/progscala2/visibility/private-type.scalaX
// WON'T
COMPILE

package scopeA {
  class PrivateClass1(private[PrivateClass1] val privateField1: Int) {
    private[PrivateClass1] val privateField2 = 1

    def equalFields(other: PrivateClass1) =
      (privateField1 == other.privateField1) &&
      (privateField2 == other.privateField2) &&
      (nested  == other.nested)

    class Nested {
      private[Nested] val nestedField = 1
    }

    private[PrivateClass1] val nested = new Nested
    val nestedNested = nested.nestedField   // ERROR
  }

  class PrivateClass2 extends PrivateClass1(1) {
    val field1 = privateField1  // ERROR
    val field2 = privateField2  // ERROR
    val nField = new Nested().nestedField  // ERROR
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1(1)
    val privateField1 = privateClass1.privateField1  // ERROR
    val privateField2 = privateClass1.privateField2  // ERROR
    val privateNField = privateClass1.nested.nestedField // ERROR
  }
}
```

There are seven access errors in this file.

A `private[PrivateClass1]` member is visible to other instances, so the `equalFields` method now parses. Hence, `private[T]` is not as restrictive as `private[this]`. Note that `PrivateClass1` can't see `Nested.nestedField` because that field is declared `private[Nested]`.

Tip

When members of `T` are declared `private[T]` the behavior is equivalent to `private`. It is not equivalent to `private[this]`, which is more restrictive.

What if we change the scope of `Nested.nestedField` to be `private[PrivateClass1]`? Let's see how `private[T]` affects nested types:

```
// src/main/scala/progscala2/visibility/private-type-nested.scalaX
// WON'T
COMPILE

package scopeA {
  class PrivateClass1 {
    classNested {
      private[PrivateClass1] val nestedField = 1
    }

    private[PrivateClass1] val nested = new Nested
    val nestedNested = nested.nestedField
  }

  classPrivateClass2 extends PrivateClass1 {
    val nField = new Nested().nestedField   // ERROR
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1
    val privateNField = privateClass1.nested.nestedField // ERROR
  }
}
```

Two compilation errors occur.

Now `nestedField` is visible to `PrivateClass1`, but it is still invisible outside of `PrivateClass1`. This is how `private` works in Java.

Let’s examine scoping using a package name:

```
// src/main/scala/progscala2/visibility/private-pkg-type.scalaX
// WON'T
COMPILE

package scopeA {
  private[scopeA] class PrivateClass1

  package scopeA2 {
    private [scopeA2] class PrivateClass2
    private [scopeA]  class PrivateClass3
  }

  class PrivateClass4 extends PrivateClass1
  protected class PrivateClass5 extends PrivateClass1
  private class PrivateClass6 extends PrivateClass1
  private[this] class PrivateClass7 extends PrivateClass1

  private[this] class PrivateClass8 extends scopeA2.PrivateClass2 // ERROR
  private[this] class PrivateClass9 extends scopeA2.PrivateClass3
}

package scopeB {
  class PrivateClass1B extends scopeA.PrivateClass1 // ERROR
}
```

Compiling this file also yields two errors.

Note that `PrivateClass2` can’t be subclassed outside of `scopeA2`, but `PrivateClass3` can be subclassed in `scopeA`, because it is declared `private[scopeA]`.

Finally, let’s look at the effect of package-level scoping of type members:

```
// src/main/scala/progscala2/visibility/private-pkg.scalaX
// WON'T
COMPILE

package scopeA {
  class PrivateClass1 {
    private[scopeA] val privateField = 1

    class Nested {
      private[scopeA] val nestedField = 1
    }

    private[scopeA] val nested = new Nested
  }

  class PrivateClass2 extends PrivateClass1 {
    val field  = privateField
    val nField = new Nested().nestedField
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1
    val privateField  = privateClass1.privateField
    val privateNField = privateClass1.nested.nestedField
  }

  package scopeA2 {
    class PrivateClass4 {
      private[scopeA2] val field1 = 1
      private[scopeA]  val field2 = 2
    }
  }

  class PrivateClass5 {
    val privateClass4 = new scopeA2.PrivateClass4
    val field1 = privateClass4.field1  // ERROR
    val field2 = privateClass4.field2
  }
}

package scopeB {
  class PrivateClass1B extends scopeA.PrivateClass1 {
    val field1 = privateField   // ERROR
    val privateClass1 = new scopeA.PrivateClass1
    val field2 = privateClass1.privateField  // ERROR
  }
}
```

This last file has three errors.

The only errors are when we attempt to access members scoped to `scopeA` from the unrelated package `scopeB` and when we attempt to access a member from a nested package `scopeA2` that is scoped to that package.

Tip

When a type or member is declared `private[P]`, where `P` is the enclosing package, it is equivalent to Java's `package private` visibility.

Final Thoughts on Visibility

Scala visibility declarations are very flexible, and they behave consistently. They provide fine-grained control over visibility at all possible scopes, from the instance level (`private[this]`) up to package-level visibility (`private[P]`, for a package `P`). For example, they make it easier to create reusable components with types exposed outside of the component's top-level package, while hiding implementation types and type members within the component's packages.

These fine-grained visibility controls are not widely used outside the standard library, but they should be. When you're writing your own libraries, consider which types and methods should be hidden from clients and apply the appropriate visibility rules to them.

Finally, we observed a potential "gotcha" with hidden members of traits—see the following tip.

Tip

Be careful when choosing names for the members of traits. If two traits have a member of the same name and the traits are used in the same instance, a name collision will occur even if both members are private.

Fortunately, the compiler catches this problem.

Recap and What's Next

Scala's visibility rules offer fine-grained controls that allow us to limit visibility of features in precise ways. It's easy to be lazy and just use the default public visibility. However, good library design includes attention to what features are visible outside the library. Inside the library, limiting visibility between components helps ensure robustness and makes long-term maintenance easier.

Now we turn to a tour of Scala's type system. We already know quite a lot about it, but to really exploit the type system's power, we need a systematic understanding of it.