# Table of Contents for Programming Scala, 2nd Edition

# Chapter 14. Scala's Type System, Part I

Scala is a statically typed language. Its type system is arguably the most sophisticated in any programming language, in part because it combines comprehensive ideas from functional programming and object-oriented programming. The type system tries to be logically comprehensive, complete, and consistent. It fixes several limitations in Java's type system.

Ideally, a type system would be expressive enough that you could prevent your program from ever "inhabiting" an invalid state. It would let you enforce these constraints at compile time, so runtime failures would never occur. In practice, we're far from that goal, but Scala's type system pushes the boundaries toward this long-term goal.

However, the type system can be intimidating at first. It is the most controversial feature of the language. When people claim that Scala is "complex," they usually have the type system in mind.

Fortunately, type inference hides many of the details. Mastery of the type system is not required to use Scala effectively, although you'll eventually need to be familiar with most constructs.

We've already learned a lot about the type system. This chapter ties these threads together and covers the remaining, widely used type system features that every new Scala developer should learn. The next chapter covers more advanced features that aren't as important to learn immediately if you're new to Scala. When we get to Chapter 24, we'll discuss type information in the context of reflection (runtime introspection) and macros.

We'll also discuss similarities with Java's type system, because it may be a familiar point of reference for you. Understanding the differences is also useful for interoperability with Java libraries.

In fact, some of the complexity in Scala's type system arises from features that represent idiosyncrasies in Java's type system. Scala needs to support these features for interoperability.

Now let's begin by revisiting familiar ground, *parameterized types*.

## Parameterized Types

We have encountered parameterized types in several places already. In Abstract Types Versus Parameterized Types, we compared them to abstract types. In Parameterized Types: Variance Under Inheritance, we explored *variance* under subtyping.

In this section, we'll recap some of the details, then add additional information that you should know.

### Variance Annotations

First, let's recall how *variance annotations* work. A declaration like `List[+A]` means that `List` is parameterized by a single type, represented by `A`. The `+` is a variance annotation and in this case it says that `List` is *covariant* in the type parameter, meaning that `List[String]` is considered a subtype of `List[AnyRef]`, because `String` is a subtype of `AnyRef`.

Similarly, the `-` variance annotation indicates that the type is *contravariant* in the type parameter. One example is the types for the `N` arguments passed to `FunctionN` values. Consider `Function2`, which has the type signature

`Function2[-T1, -T2, +R]`. We saw in Functions Under the Hood why the types for the function arguments must be contravariant.

### Type Constructors

Sometimes you'll see the term *type constructor* used for a parameterized type. This reflects how the parameterized type is used to create *specific* types, in much the same way that an *instance constructor* for a class is used to construct instances of the class.

For example, `List` is the type constructor for `List[String]` and `List[Int]`, which are different types. In fact, you could say that all classes are type constructors. Those without type parameters are effectively "parameterized types" with zero type parameter arguments.

### Type Parameter Names

Consider using descriptive names for your type parameters. A complaint of new Scala developers is the terse names used for type parameters, like `A` and `B`, in the implementations and Scaladocs for methods like `List.+:`. On the other hand, you quickly learn how to interpret these symbols, which follow some simple rules:

1. Use single-letter or double-letter names like `A`, `B`, `T1`, `T2`, etc. for very generic type parameters, such as container elements. Note that the actual element types have no close connection to the containers. Lists work the same whether they are holding strings, numbers, other lists, etc. This decoupling makes "generic programming" possible.
2. Use more descriptive names for types that are closely associated with the underlying container. Perhaps `That` in the `List.+:` signature doesn't express an obvious meaning when you first encounter it, but it's sufficient for the job once you understand the collection design idioms that we discussed in Design Idioms in the Collections Library.

## Type Bounds

When defining a parameterized type or method, it may be necessary to specify *bounds* on the type parameter. For example, a container might assume that certain methods exist on all types used for the type parameter.

### Upper Type Bounds

Upper type bounds specify that a type must be a subtype of another type. For a motivating example, we saw in Scala's Built-in Implicits that `Predef` defines implicit conversions to wrap instances of `Array` (that is, a Java array) in a `collection.mutable.ArrayOps` instance, where the latter provides the sequence operations we know and love.

There are several of these conversions defined. Most are for the specific `AnyVal` types, like `Long`, but one handles conversions of `Array[AnyRef]` instances:

```
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): ArrayOps[T] =
  new ArrayOps.ofRef[T](xs)
implicit def longArrayOps(xs: Array[Long]): ArrayOps[Long] =
  new ArrayOps.ofLong(xs)
    // Methods for the other AnyVal
... types.
```

The type parameter `A <: AnyRef` means “any type `A` that is a *subtype* of `AnyRef`.” Recall that a type is always a subtype and a supertype of itself, so `A` could also equal `AnyRef`. Hence, the `<:` operator indicates that the type to the left must be derived from the type to the right, or that they must be the same type. As we said in Reserved Words, this operator is actually a reserved word in the language.

By restricting the first method to apply only to subtypes of `AnyRef`, there is no ambiguity between this generic conversion method and the more specific conversion methods for `Long`, `Int`, etc.

These bounds are called *upper type bounds*, following the de facto convention that type hierarchies are drawn with subtypes below their supertypes. We followed this convention in the diagram shown in The Scala Type Hierarchy.

### Note

Type bounds and variance annotations cover unrelated issues. A type bound specifies constraints on allowed types that can be used for a type parameter in a parameterized type. For example `T <: AnyRef` limits `T` to be a subtype of `AnyRef`. A variance annotation specifies when an instance of a subtype of a parameterized type can be substituted where a supertype instance is expected. For example, because `List[+T]` is covariant in `T`, `List[String]` is a subtype of `List[Any]`.

## Lower Type Bounds

In contrast, a *lower type bound* expresses that one type must be a supertype (or the same type) as another. An example is the `getOrElse` method in `Option`:

```
sealed abstract class Option[+A] extends Product with Serializable {
  ...
  @inline final def getOrElse[B >: A](default: => B): B = {...}
  ...
}
```

If the `Option` instance is `Some[A]`, the value it contains is returned. Otherwise, the by-name parameter `default` is evaluated and returned. It is allowed to be a supertype of `A`. Why is that? In fact, why does Scala *require* that we declare the method this way? Let’s consider an example that illustrates why this requirement is necessary (and poorly understood):

```
// src/main/scala/progscala2/typesystem/bounds/lower-bounds.sc

class Parent(val value: Int) {                      // ❶
  override def toString = s"${this.getClass.getName}($value)"
}
class Child(value: Int) extends Parent(value)

val op1: Option[Parent] = Option(new Child(1))     // ❷      Some(Child(1
))
                                                   // Result:
val p1: Parent = op1.getOrElse(new Parent(10))     Child(1)

val op2: Option[Parent] = Option[Parent](null)     // ❸      None
                                                   // Result:
val p2a: Parent = op2.getOrElse(new Parent(10))    Parent(10)
                                                   // Result:
val p2b: Parent = op2.getOrElse(new Child(100))    Child(100)

val op3: Option[Parent] = Option[Child](null)      // ❹      None
                                                   // Result:
val p3a: Parent = op3.getOrElse(new Parent(20))    Parent(20)
                                                   // Result:
val p3b: Parent = op3.getOrElse(new Child(200))    Child(200)
```

❶

A simple type hierarchy for demonstration purposes.

❷

The reference `op1` only knows it's an `Option[Parent]`, but it actually references a (valid) subclass, `Option[Child]`, because `Option[+T]` is covariant.

❸

`Option[X](null)` always returns `None`. In this case, the reference returned is typed to `Option[Parent]`.

❹

`None` again, but now the reference returned is typed to `Option[Child]`, although it is assigned to an `Option[Parent]` reference.

These two lines near the end illustrate the crucial point:

```
val op3: Option[Parent] = Option[Child](null)
val p3a: Parent = op3.getOrElse(new Parent(20))
```

The `op3` line clearly shows that `Option[Child](null)` (i.e., `None`) is assigned to `Option[Parent]`, but what if instead that value came back from a "black-box" method call, so we couldn't know what it really is? The crucial point in this example is that the calling code only has references to `Option[Parent]`, so it has the reasonable expectation that a `Parent` value can be extracted from an `Option[Parent]`, whether it has a `None`, in which case the default `Parent` argument is returned, or it is a `Some[Parent]` or a `Some[Child]`, in which case the value in the `Some` is returned. All combinations return a `Parent` value, as shown, although sometimes it is actually a `Child` subclass instance.

Suppose `getOrElse` had this declaration instead:

```
  @inline final def getOrElse(default: => A): A =
{...}
```

In this case, it would not type check to call `op3.getOrElse(new Parent(20))`, because the object that `op3` references is of type `Option[Child]`, so it would expect a `Child` instance to be passed to `getOrElse`.

This is why the compiler won't allow this simpler method signature and instead requires the original signature with the `[B >: A]` bounds. To see this, let's sketch our own option type, call it `Opt`, that uses this method declaration. For simplicity, we'll treat a `null` value as the equivalent of `None`, for which `getOrElse` should return the default value:

```
// src/main/scala/progscala2/typesystem/bounds/lower-bounds2.sc
scala> case class Opt[+A](value: A = null) {
        def getOrElse(default: A) = if (value != null) value else
     |default
     |}
<console>:8: error: covariant type A occurs in contravariant position
  in type A of value default
         def getOrElse(default: A) = if (value != null) value else
default
                             ^
```

So, whenever you see the error message "covariant type A occurs in contravariant position…," it means that you have attempted to define a parameterized type that's covariant in the parameter, but you're also trying to define methods that accept instances of that type parameter, rather than a new supertype parameter, i.e., `B >: A`. This is disallowed for the reasons just outlined.

If this argument sounds vaguely familiar, it should. It's essentially the same behavior we discussed for function types in Functions Under the Hood for the type parameters used for the function arguments. They must also be *contravariant*, e.g., `Function2[-A1, -A2, +R]`, because those argument types occur in contravariant position in the `apply` methods used to implement instances of functions.

## Tip

When attempting to understand why variance annotations and type bounds work the way they do, remember to study what happens with instances of types from the perspective of code that uses them, where that code might have a reference to a parent type, but the actual instance is a child type.

Consider what happens if we change our *covariant* `Opt[+A]` to *invariant*, `Opt[A]`:

```
// src/main/scala/progscala2/typesystem/bounds/lower-bounds2.sc
scala> case class Opt[A](value: A = null) {
     |
  def getOrElse(default: A) = if (value != null) value else
default
     |}

scala> val p4: Parent = Opt(new Child(1)).getOrElse(new Parent(10))
<console>:11: error: type mismatch;
 found   : Parent
 required: Child
       val p4: Parent = Opt(new Child(1)).getOrElse(new Parent(10))
                                                     ^

scala> val p5: Parent = Opt[Parent](null).getOrElse(new Parent(10))
p5: Parent = Parent(10)

scala> val p6: Parent = Opt[Child](null).getOrElse(new Parent(10))
<console>:11: error: type mismatch;
 found   : Parent
 required: Child
       val p6: Parent = Opt[Child](null).getOrElse(new Parent(10))
                                                    ^
```

Only the `p5` case works. We can no longer assign an `Opt[Child]` to an `Opt[Parent]` reference.

It's worth discussing the subtleties of another class of examples where parameterized types that are *covariant* in the type parameter must have *contravariant* behavior in some methods, when we add elements to an immutable collection to construct a new collection.

Consider the `Seq.+:` method for prepending an element to a sequence, creating a new sequence. We've used it before. It's typically used with operator notation, as in the following example:

```
scala> 1 +: Seq(2, 3)
res0: Seq[Int] = List(1, 2, 3)
```

In the Scaladocs, this method has a simplified signature, which assumes we're prepending elements of the same type (`A`), but the method's actual signature is more general. Here are both signatures:

```
def +:(elem: A): Seq[A] = {...}
// ❶
def +:[B >: A, That](elem: B)(
// ❷
  implicit bf: CanBuildFrom[Seq[A], B, That]): That = {...}
```

❶

Simplified signature, which assumes the type parameter `A` stays the same.

❷

Actual signature, which supports prepending elements of an arbitrary new supertype and also includes the

`CanBuildFrom` formalism we've discussed previously.

In the following example, we prepend a `Double` value to a `Seq[Int]`:

```
scala> 0.1 +: res0
<console>:9: warning: a type was inferred to be `AnyVal`; this may
  indicate a programming error.
              0.1 +: res0
                  ^
res1: Seq[AnyVal] = List(0.1, 1, 2, 3)
```

You won't see this warning if you use a version of Scala before 2.11. I'll explain why in a moment.

The `B` type isn't the same as the new head value's type, `Double` in this case. Instead, `B` is inferred to be the *least upper bound* (LUB), i.e., the closest supertype of the original type `A` (`Int`) and the type of the new element (`Double`). Hence, `B` is inferred to be `AnyVal`.

For `Option`, `B` was inferred to be the same type as the default argument. If the object was a `None`, the default was returned and we could "forget about" the original `A` type.

In the case of a list, we are keeping the existing `A`-typed values and adding a new value of type `B`, so a LUB has to be inferred that is a parent of both `A` and `B`.

While convenient, inferring a broader, LUB type can be a surprise if you thought you were not changing from the original type parameter. That's why Scala 2.11 added a warning when an expression infers a broad LUB type.

The workaround is to explicitly declare the expected return type:

```
// Scala 2.11 workaround for warning.
scala> val l2: List[AnyVal] = 0.1 +: res0
l2: List[AnyVal] = List(0.1, 1, 2, 3)
```

Now the compiler knows that you want the broader LUB type.

To recap, there is an intimate relationship between parameterized types that are *covariant* in their parameters and *lower type bounds* in method arguments.

Finally, you can combine upper and lower type bounds:

```
class Upper
class Middle1 extends Upper
class Middle2 extends Middle1
class Lower extends Middle2
case class C[A >: Lower <: Upper](a: A)
// case class C2[A <: Upper >: Lower](a: A)  // Does not
compile
```

The type parameter, `A`, must appear first. Note that the `C2` case does *not* compile; the lower bound has to appear before the upper bound.

## Context Bounds

We learned about *context bounds* and their uses in Using implicitly. Here is the example that we considered then:

```
// src/main/scala/progscala2/implicits/implicitly-
args.sc
import math.Ordering

case class MyList[A](list: List[A]) {
  def sortBy1[B](f: A => B)(implicit ord: Ordering[B]): List[A] =
    list.sortBy(f)(ord)

  def sortBy2[B : Ordering](f: A => B): List[A] =
    list.sortBy(f)(implicitly[Ordering[B]])
}

val list = MyList(List(1,3,5,2,4))

list sortBy1 (i => -i)
list sortBy2 (i => -i)
```

Comparing the two versions of `sortBy`, note that the implicit parameter shown explicitly in `sortBy1` and "hidden" in `sortBy2` is a parameterized type. The type expression `B : Ordering` is equivalent to `B` with no modification and an implicit parameter of type `Ordering[B]`. This means that no particular type can be used for `B` unless there exists a corresponding `Ordering[B]`.

A similar concept is *view bounds*.

## View Bounds

*View bounds* look similar to context bounds and they can be considered a special case of context bounds. They can be declared in either of the following ways:

```
class C[A] {
  def m1[B](...)(implicit view: A => B): ReturnType = {...}
  def m2[A <% B](...): ReturnType = {...}
}
```

Contrast with the previous context bound case, where the implicit value for `A : B` had to be of type `B[A]`. Here, we need an implicit function that converts an `A` to a `B`. We say that "B is a view onto A." Also, contrast with a upper bound expression `A <: B`, which says that `A` *is* a subtype of `B`. A view bound is a looser requirement. It says that `A` must be convertable to `B`.

Here is a sketch of how this feature might be used. The Hadoop Java API requires data values to be wrapped in custom serializers, which implement a so-called `Writable` interface, for sending values to remote processes. Users of the API must work with these `Writable`s explicitly, an inconvenience. We can use view bounds to handle this automatically (we'll use our own `Writable` for simplicity):

```
// src/main/scala/progscala2/typesystem/bounds/view-
bounds.sc
import scala.language.implicitConversions

object Serialization {
  case class Writable(value: Any) {
                              "-- $value --
    def serialized: String = s"
//  ❶
  }

  implicit def fromInt(i: Int) = Writable(i)
//  ❷
  implicit def fromFloat(f: Float) = Writable(f)
  implicit def fromString(s: String) = Writable(s)
}

import Serialization._

object RemoteConnection {
//  ❸
  def write[T <% Writable](t: T): Unit =
//  ❹
                           // Use stdout as the "remote
    println(t.serialized)  connection"
}

                                  // Prints -- 100 -
RemoteConnection.write(100)       -
                                  // Prints -- 3.14 -
RemoteConnection.write(3.14f)     -
                                  // Prints -- hello! -
RemoteConnection.write("hello!")  -
// RemoteConnection.write((1, 2))                                      ❺
```

❶

Use `String` as the “binary” format, for simplicity.

❷

Define a few implicit conversions. Note that we defined methods here, but we said that functions of type `A => B` are required. Recall that Scala will lift methods to functions when needed.

❸

Object that encapsulates writing to a “remote” connection.

❹

A method that accepts an instance of any type and writes it to the connection. It invokes the implicit conversion so the `serialized` method can be called on it.

❺

Can’t use a tuple, because there is no implicit “view” available for it.

Note that we don't need `Predef.implictly` or something like it. The implicit conversion is invoked for us automatically by the compiler.

View bounds can be implemented with context bounds, which are more general, although view bounds provide a nice, shorthand syntax. Hence, there has been some discussion in the Scala community of deprecating view bounds. Here is the previous example reworked using context bounds:

```
// src/main/scala/progscala2/typesystem/bounds/view-to-context-
bounds.sc
import scala.language.implicitConversions

object Serialization {
  case class Rem[A](value: A) {
                              "-- $value --
    def serialized: String = s"
  }
  type Writable[A] = A => Rem[A]
//  ❶
  implicit val fromInt: Writable[Int]        = (i: Int)     => Rem(i)
  implicit val fromFloat: Writable[Float]    = (f: Float)   => Rem(f)
  implicit val fromString: Writable[String] = (s: String) => Rem(s)
}

import Serialization._

object RemoteConnection {
  def write[T : Writable](t: T): Unit =
//  ❷
                           // Use stdout as the "remote
    println(t.serialized)  connection"
}

RemoteConnection.write(100)       // Prints -- 100 --
❸
                                  // Prints -- 3.14 -
RemoteConnection.write(3.14f)     -
                                  // Prints -- hello! -
RemoteConnection.write("hello!")  -
// RemoteConnection.write((1,
2))
```

❶

A type alias that makes it more convenient to use context bounds, followed by the implicit definitions corresponding to the previous example.

❷

The `write` method now implemented with a context bound.

❸

The same calls to `write` from the previous example, producing the same results.

So, consider avoiding view bounds in your code, because they may be deprecated in the future.

## Understanding Abstract Types

Parameterized types are common in statically typed, object-oriented languages. Scala also supports abstract types, which are common in some functional languages. We introduced abstract types in Abstract Types Versus Parameterized Types. These two approaches overlap somewhat, as we'll explore in a moment. First, let's discuss using abstract types:

```
// src/main/scala/progscala2/typesystem/abstracttypes/abstract-types-
ex.sc

trait exampleTrait {
                        // t1 is
  type t1               unconstrained
  type t2 >: t3 <: t1
// t2 must be a supertype of t3 and a subtype of
t1
                        // t3 must be a subtype of
  type t3 <: t1         t1
                        // t4 must be a subtype of Seq of
  type t4 <: Seq[t1]    t1
  // type t5 = +AnyRef  // ERROR: Can't use variance
  annotations

                        // Can't initialize until t1
  val v1: t1            defined.
                        //
  val v2: t2            ditto...
                        //
  val v3: t3            ...
                        //
  val v4: t4            ...
}
```

The comments explain most of the details. The relationships between `t1`, `t2`, and `t3` have some interesting points. First, the declaration of `t2` says that it must be "between" `t1` and `t3`. Whatever `t1` becomes, it must be a superclass of `t2` (or equal to it), and `t3` must be a subclass of `t2` (or equal to it).

Note the line that declares `t3`. It must specify that it is a subtype of `t1` to be consistent with the declaration of `t2`. It would be an error to omit the type bound, because `t3 <: t1` is implied by the previous declaration of `t2`. Trying `t3 <: t2` triggers an error for an "illegal cyclic reference to `t2`" in the declaration type `t2 >: t3 <: t1`. We also can't omit the explicit declaration of `t3` and assume its existence is "implied" somehow by the declaration for `t2`. Of course, this complex example is contrived to demonstrate the behaviors.

We can't use variance annotations on type members. Remember that the types are *members* of the enclosing type, not type parameters, as for parameterized types. The enclosing type may have an inheritance relationship with other types, but member types behave just like member methods and variables. They don't affect the inheritance relationships of their enclosing type. Like other members, member types can be declared abstract or concrete. However, they can also be refined in subtypes without being fully defined, unlike variables and methods. Of course, instances can only be created when the abstract types are given concrete definitions.

Let's define some traits and a class to test these types:

```
trait T1 { val name1: String }
trait T2 extends T1 { val name2: String }
case class C(name1: String, name2: String) extends T2
```

Finally, we can declare a concrete type that defines the abstract type members and initializes the values accordingly:

```
object example extends exampleTrait {
  type t1 = T1
  type t2 = T2
  type t3 = C
  type t4 = Vector[T1]

  val v1 = new T1 { val name1 = "T1"}
  val v2 = new T2 { val name1 = "T1"; val name2 = "T2" }
  val v3 = C("1", "2")
  val v4 = Vector(C("3", "4"))
}
```

## Comparing Abstract Types and Parameterized Types

Technically, you could implement almost all the idioms that parameterized types support using abstract types and vice versa. However, in practice, each feature is a natural fit for different design problems.

Parameterized types work nicely for containers, like collections, where there is little connection between the types represented by the type parameter and the container itself. For example, a list works the same if it's a list of strings, a list of doubles, or a list of integers.

What about using type parameters instead? Consider the declaration of `Some` from the standard library:

```
case final class Some[+A](val value : A) { ... }
```

If we try to convert this to use abstract types, we might start with the following:

```
case final class Some(val value : ???) {
  type A
  ...
}
```

What should be the type of the argument `value`? We can't use `A` because it's not in scope at the point of the constructor argument. We could use `Any`, but that defeats the purpose of type safety.

Hence, parameterized types are the only good approach when arguments of the type are given to the constructor.

In contrast, abstract types tend to be most useful for type "families," types that are closely linked. Recall the example we saw in Abstract Types Versus Parameterized Types (some unimportant details not repeated):

```
// src/main/scala/progscala2/typelessdomore/abstract-types.sc

import java.io._

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

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

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

BulkReader declares the abstract type In with no type bounds. The subtypes StringBulkReader and FileBulkReader define the type. Note that the user no longer specifies a type through a type parameter. Instead we have total control over the type member In and its enclosing class, so the implementation keeps them consistent.

Let's consider another example, a potential design approach for the *Observer Pattern* we've encountered before in Traits as Mixins and again in Overriding fields in traits. Our first approach will fail, but we'll fix it in the next section on self-type annotations:

```
// src/main/scala/progscala2/typesystem/abstracttypes/SubjectObserver.scalaX
package progscala2.typesystem.abstracttypes

abstract class SubjectObserver {                                    //
❶
  type S <: Subject                                                 //
❷
  type O <: Observer

  trait Subject {                                                   //
❸
    private var observers = List[O]()

    def addObserver(observer: O) = observers ::= observer

    def notifyObservers() = observers.foreach(_.receiveUpdate(this)) // ❹
  }

  trait Observer {                                                  //
❺
    def receiveUpdate(subject: S)
  }
}
```

❶

Encapsulate the subject-observer relationship in a single type.

❷

Declare abstract type members for the subject and observer types, bounded by the `Subject` and `Observer` traits declared here.

❸

The `Subject` trait, which maintains a list of observers.

❹

Notify the observers. This line *doesn't compile*.

❺

The `Observer` trait with a method for receiving updates.

Attempting to compile this file produces the following error:

```
em/abstracttypes/observer.scala:14: type mismatch;
[error]  found   : Subject.this.type (with underlying type
                   SubjectObserver.this.Subject)
[error]  required: SubjectObserver.this.S
[error]     def notifyObservers = observers foreach
(_.receiveUpdate(this))
[error]                                                         ^
```

What we wanted to do is use bounded, abstract type members for the subject and observer, so that when we specify concrete types for them, especially the `S` type, our `Observer.receiveUpdate(subject: S)` will have the exact type for the subject, not the less useful parent type, `Subject`.

However, when we compile it, `this` is of type `Subject` when we pass it to `receiveUpdate`, not the more specific type `S`.

We can fix the problem with a self-type annotation.

## Self-Type Annotations

You can use `this` in a method to refer to the enclosing instance, which is useful for referencing another member of the instance. Explicitly using `this` is not usually necessary for this purpose, but it's occasionally useful for disambiguating a reference when several items are in scope with the same name.

*Self-type annotations* support two objectives. First, they let you specify additional type expectations for `this`. Second, they can be used to create aliases for `this`.

To illustrate specifying additional type expectations, let's revisit our `SubjectObserver` class from the previous section. By specifying additional type expectations, we'll solve the compilation problem we encountered. Only two changes are required:

```
//
src/main/scala/progscala2/typesystem/selftype/SubjectObserver.scala
package progscala2.typesystem.selftype

abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer

  trait Subject {
    self: S =>
//  ❶
    private var observers = List[O]()

    def addObserver(observer: O) = observers ::= observer

    def notifyObservers() = observers.foreach(_.receiveUpdate(self)) //
❷
  }

  trait Observer {
    def receiveUpdate(subject: S)
  }
}
```

❶

Declare a self-type annotation for `Subject`, which is `self: S`. This means that we can now “assume” that a `Subject` will really be an instance of the subtype `S`, which will be whatever concrete classes we define that mix in `Subject`.

❷

Pass `self` rather than `this` to `receiveUpdate`.

Now it compiles. Let’s see how the types might be used to observe button clicks:

```
// src/main/scala/progscala2/typesystem/selftype/ButtonSubjectObserver.scala
package progscala2.typesystem.selftype

case class Button(label: String) {                                   //
❶
  def click(): Unit = {}
}

object ButtonSubjectObserver extends SubjectObserver {               // ❷
  type S = ObservableButton
  type O = ButtonObserver

  class ObservableButton(label: String) extends Button(label) with Subject {
    override def click() = {
      super.click()
      notifyObservers()
    }
  }

  trait ButtonObserver extends Observer {
    def receiveUpdate(button: ObservableButton)
  }
}

import ButtonSubjectObserver._

class ButtonClickObserver extends ButtonObserver {                   // ❸
 val clicks = new scala.collection.mutable.HashMap[String,Int]()

  def receiveUpdate(button: ObservableButton) = {
    val count = clicks.getOrElse(button.label, 0) + 1
    clicks.update(button.label, count)
  }
}
```

❶

A simple `Button` class.

❷

A concrete subtype of `SubjectObserver` for buttons, where `Subject` and `Observer` are both subtyped to the more specific types we want (note the type of the value passed to `ButtonObserver.receiveUpdate`). `ObservableButton` overrides `Button.click` to notify the observers after calling `Button.click`.

❸

Implement `ButtonObserver` to track the number of clicks for each button in a UI.

The following script creates two `ObservableButton`s, attaches the same observer to both, clicks them a few times, and prints the number of counts observed for each one:

```
//
src/main/scala/progscala2/typesystem/selftype/ButtonSubjectObserver.sc
import progscala2.typesystem.selftype._

val buttons = Vector(new ObservableButton("one"), new ObservableButton("two"))
val observer = new ButtonClickObserver
buttons foreach (_.addObserver(observer))
for (i <- 0 to 2) buttons(0).click()
for (i <- 0 to 4) buttons(1).click()
println(observer.clicks)
// Map("one" -> 3, "two" ->
5)
```

So, we can use self-type annotations to solve a typing problem when using abstract type members.

Another example is a pattern for specifying “modules” and wiring them together. Consider this example that sketches a three-tier application, with a persistence layer, middle tier, and UI:

```
// src/main/scala/progscala2/typesystem/selftype/selftype-cake-
pattern.sc

trait Persistence { def startPersistence(): Unit }                    //
❶
trait Midtier { def startMidtier(): Unit }
trait UI { def startUI(): Unit }

trait Database extends Persistence {
// ❷
  def startPersistence(): Unit = println("Starting Database")
}
trait BizLogic extends Midtier {
  def startMidtier(): Unit = println("Starting BizLogic")
}
trait WebUI extends UI {
                                 "Starting
  def startUI(): Unit = println(WebUI"          )
}

trait App { self: Persistence with Midtier with UI =>                 //
❸

  def run() = {
    startPersistence()
    startMidtier()
    startUI()
  }
}

object MyApp extends App with Database with BizLogic with WebUI       //
❹

MyApp.run
// ❺
```

❶

Define traits for the persistence, middle, and UI tiers of the application.

❷

Implement the “concrete” behaviors as traits.

❸

Define a trait (or it could be an abstract class) that defines the “skeleton” of how the tiers glue together. For this simple example, the `run` method just starts each tier. The self-type annotation is discussed in the following text.

❹

Define the `MyApp` object that extends `App` and mixes in the three concrete traits that implement the required behaviors.

❺

Run the application.

Running the script prints the following output from `run`:

```
Starting
Database
Starting
BizLogic
Starting WebUI
```

This script shows a schematic layout for an `App` (application) infrastructure supporting several tiers. Each abstract trait declares a `start*` method that does the work of initializing the tier. Each abstract tier is implemented by a corresponding concrete trait, not a class, so we can use each one as a mixin.

The `App` trait wires the tiers together. It’s `run` method starts each tier. Note that no concrete implementations of these traits is specified here. A concrete application must be constructed by mixing in implementations of these traits.

The *self-type annotation* is the crucial point:

```
self: Persistence with Midtier with UI =>
```

When a type annotation is added to a self-type annotation, `Persistence with Midtier with UI`, in this case, it specifies that the trait or abstract class must be mixed with those traits or subtypes that implement any abstract members, in order to define a concrete instance. Because this assumption is made, the trait is allowed to access members of those traits, even though they are not yet part of the type. Here, `App.run` calls the `start*` methods from the other traits.

The concrete instance `MyApp` extends `App` and mixes in the traits that satisfy the dependencies expressed in the self type.

This picture of stacking layers leads to the name *Cake Pattern*, where modules are declared with traits and another abstract type is used to integrate the traits with a self-type annotation. A concrete object mixes in the actual implementation traits and extends an optional parent class (we'll discuss the implications of this pattern, pro and con, in more detail in Traits as Modules).

This use of self-type annotations is actually equivalent to using inheritance and mixins instead (with the exception that `self` would not be defined):

```
trait App extends Persistence with Midtier with UI {
  def run = { ... }
}
```

There are a few special cases where self-type annotations behave differently than inheritance, but in practice, the two approaches behave interchangeably.

However, they express different intent. The inheritance-based implementation just shown suggests that `App` is a subtype of `Persistence`, `Midtier`, and `UI`. In contrast, the self-type annotation expresses composition of behavior through mixins more explicitly.

## Tip

Self-type annotations emphasize mixin composition. Inheritance can imply a subtype relationship.

That said, most Scala code tends to use the inheritance approach, rather than self-type annotations, unless integration of larger-scale "modules" (traits) is being done, where the self-type annotation conveys the design decisions more clearly.

Now let's consider the second usage of self-type annotations, aliasing `this`:

```
// src/main/scala/progscala2/typesystem/selftype/this-alias.sc

class C1 { self =>
//  ❶
                                      "C1.talk:
  def talk(message: String) = println("           + message)
  class C2 {
    class C3 {
                                            "C3.talk:
      def talk(message: String) = self.talk("           + message)   //
❷
    }
    val c3 = new C3
  }
  val c2 = new C2
}
val c1 = new C1
c1.talk("Hello")
//  ❸
c1.c2.c3.talk("World")
//  ❹
```

❶

Define `self` to be an alias of `this` in the context of `C1`.

❷

Use `self` to call `C1.talk`.

❸

Call `C1.talk` via the `c1` instance.

❹

Call `C3.talk` via the `c1.c2.c3` instance, which will itself call `C1.talk`.

Note that the name `self` is arbitrary. It is not a keyword. Any name could be used. We could also define self-type annotations inside `C2` and `C3`, if we needed them.

The script prints the following:

```
C1.talk: Hello
C1.talk: C3.talk:
World
```

Without the self-type declaration, we can't invoke `C1.talk` directly from within `C3.talk`, because the latter shadows the former, since they share the same name. `C3` is not a direct subtype of `C1` either, so `super.talk` can't be used.

So, you can think of the self-type annotation in this context as a "generalized this" reference.

## Structural Types

You can think of *structural types* as a type-safe approach to *duck typing*, the popular name for the way method resolution works in dynamically typed languages ("If it walks like a duck and talks like a duck, it must be a duck"). For example, in Ruby, when your code contains `starFighter.shootWeapons`, the runtime doesn't yet know if `shootWeapons` actually exists for the `starFighter` instance, but it follows various rules to locate the method to call or handle the failure if one isn't found.

Scala doesn't support this kind of runtime method resolution (an exception is discussed in Chapter 19). Instead, Scala supports a similar mechanism at compile time. Scala allows you to specify that an object must adhere to a certain *structure*: that it contains certain members (types, fields, or methods), without requiring a specific *named* type that encloses those members.

We normally use *nominal typing*, so called because we work with types that have names. In structural typing, we only consider the type's structure. It can be anonymous.

To see an example, let's examine how we might use structural types in the *Observer Pattern*. We'll start with the simpler implementation we saw in Traits as Mixins, as opposed to the one we considered previously in this chapter. First, here are the important details from that example:

```
trait Observer[-State] {
  def receiveUpdate(state: State): Unit
}
trait Subject[State] {
  private var observers: List[Observer[State]] = Nil
  ...
}
```

A drawback of this implementation is that any type that should watch for state changes in `Subject`s must implement the `Observer` trait. But really, the *true minimum* requirement is that they implement the `receiveUpdate` method.

So, here is a reimplementation of the example using a *structural type* for the `Observer`:

```
//
src/main/scala/progscala2/typesystem/structuraltypes/Observer.scala
package progscala2.typesystem.structuraltypes

trait Subject {
// ❶

  import scala.language.reflectiveCalls
// ❷

  type State
// ❸

  type Observer = { def receiveUpdate(state: Any): Unit }             //
❹

  private var observers: List[Observer] = Nil
// ❺

  def addObserver(observer:Observer): Unit =
    observers ::= observer

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

❶

An unrelated change, but illustrative; remove the previous type parameter `State` and make it an abstract type instead.

❷

Enable the optional feature to allow reflective method calls (see the following text).

❸

The `State` abstract type.

❹

The type `Observer` is a *structural type*.

❺

The `State` type parameter was removed from `Observer`, as well.

The declaration `type Observer = { def receiveUpdate(subject: Any): Unit }` says that any object with this `receiveUpdate` method can be used as an observer. Unfortunately, Scala won't let a structural type refer to an abstract type or type parameter. So, we can't use `State`. We have to use a type that's already known, like `Any`. That means that the receiver may need to cast the instance to the correct type, a big drawback.

Another drawback is implied by the import statement. Because we don't have a type name to use to verify that a candidate observer instance implements the correct method, the compiler has to use reflection to confirm the method is present on the instance. This adds overhead, although it won't be noticeable unless observers are added frequently. Using reflection is considered an optional feature, hence the import statement.

This script tries the new implementation:

```
// src/main/scala/progscala2/typesystem/structuraltypes/Observer.sc
import progscala2.typesystem.structuraltypes.Subject
import scala.language.reflectiveCalls

object observer {
// ❶
  def receiveUpdate(state: Any): Unit = println("got one! "+state)
}

val subject = new Subject {
// ❷
  type State = Int
  protected var count = 0

  def increment(): Unit = {
    count += 1
    notifyObservers(count)
  }
}

subject.increment()
subject.increment()
subject.addObserver(observer)
subject.increment()
subject.increment()
```

❶

Declare an observer object with the correct method.

❷

Instantiate the `State` trait, providing a definition for the `State` abstract type and additional behavior.

Note that the observer is registered after two increments have occurred, so it will only print that it received the numbers 3 and 4.

Despite their disadvantages, structural types have the virtue of minimizing the coupling between two things. In this case, the coupling consists of only a single method signature, rather than a type, such as a shared trait.

Taking one last look at our example, we *still* couple to a particular *name*, the method `receiveUpdate`! In a sense, we've only moved the problem of coupling from a type name to a method name. This name is completely arbitrary, so we can push the decoupling to the next level; define the `Observer` type to be an alias for a one-argument function. Here is the final form of the example:

```
//
src/main/scala/progscala2/typesystem/structuraltypes/SubjectFunc.scala
package progscala2.typesystem.structuraltypes

trait SubjectFunc {
// ❶

  type State

  type Observer = State => Unit
// ❷

  private var observers: List[Observer] = Nil

  def addObserver(observer:Observer): Unit =
    observers ::= observer

  def notifyObservers(state: State): Unit =
// ❸
    observers foreach (o => o(state))
}
```

❶

Use a new name for `Subject`. Rename the whole file, because the observer has faded into "insignificance"!

❷

Make `Observer` a type alias for a function `State => Unit`.

❸

Notifying each observer now means calling its `apply` method.

The test script is nearly the same. Here are the differences:

```
//
src/main/scala/progscala2/typesystem/structuraltypes/SubjectFunc.sc

import progscala2.typeSystem.structuraltypes.SubjectFunc

val observer: Int => Unit = (state: Int) => println("got one! "+state)

val subject = new SubjectFunc { ... }
```

This is much better! All name-based coupling is gone, we eliminated the need for reflection calls, and we're able to use `State` again, rather than `Any`, as the function argument type.

This doesn't mean that structural typing is useless. Our example only needed a function to implement what we needed. In the general case, a structural type might have several members and an anonymous function might be insufficient for our needs.

## Compound Types

When you declare an instance that combines several types, you get a *compound type*:

```
trait T1
trait T2
class C
val c = new C with T1 with T2
// c's type: C with T1 with
T2
```

In this case, the type of `c` is `C with T1 with T2`. This is an alternative to declaring a type that extends `C` and mixes in `T1` and `T2`. Note that `c` is considered a subtype of all three types:

```
val t1: T1 = c
val t2: T2 = c
val c2: C  = c
```

### Type Refinements

*Type refinements* are an additional part of compound types. They are related to an idea you already know from Java, where it's common to provide an *anonymous inner class* that implements some interface, adding method implementations and optionally additional members.

For example, if you have a `java.util.List` of objects of type `C`, for some class `C`, you can sort the list in place using the static method, `java.util.Collections.sort`:

```
List<C> listOfC = ...
java.util.Collections.sort(listOfC, new Comparator<C>() {
  public int compare(C c1, C c2) {...}
  public boolean equals(Object obj) {...}
});
```

We "refine" the base type `Comparator` to create a new type. The JVM will give a unique synthetic name to this type in the byte code.

Scala takes this a step further. It synthesizes a new type that reflects our additions. For example, recall this type from the last section on structural typing and notice the type returned by the REPL (output wrapped to fit):

```
scala> val subject = new Subject {
         type State =
     | Int
         protected var count =
     | 0
         def increment(): Unit =
     | {
           count +=
     | 1
           notifyObservers(count
     | )

     | }
     | }
subject: TypeSystem.structuraltypes.Subject{
  type State = Int; def increment(): Unit} = $anon$1@4e3d11db
```

The type signature adds the extra structural components.

Similarly, when we combine refinement with mixin traits as we instantiate an instance, a refined type is created. Consider this example where we mix in a logging trait (some details omitted):

```
scala> trait Logging {
     |
  def log(message: String): Unit = println(s"Log: $message"
)
     | }

scala> val subject = new Subject with Logging {...}
subject: TypeSystem.structuraltypes.Subject with Logging{
  type State = Int; def increment(): Unit} = $anon$1@8b5d08e
```

To access the additional members added to the refinement from outside the instance, you would have to use the reflection API (see Runtime Reflection).

## Existential Types

*Existential types* are a way of abstracting over types. They let you assert that some type "exists" without specifying exactly what it is, usually because you don't know what it is and you don't need to know it in the current context.

Existential types are particularly important for interfacing to Java's type system for three cases:

- The type parameters of generics are "erased" in JVM byte code (called *erasure*). For example, when a `List[Int]` is created, the `Int` type is not available in the byte code, so at runtime it's not possible to distinguish between a `List[Int]` and a `List[String]`, based on the known type information.
- You might encounter "raw" types, such as pre-Java 5 libraries where collections had no type parameters. (All type parameters are effectively `Object`.)
- When Java uses wildcards in generics to express variance behavior when the generics are *used*, the actual type is unknown.

Consider the case of matching on `Seq[A]` objects. You might want to define two versions of a function `double`.

One version takes a `Seq[Int]` and returns a new `Seq[Int]` with the elements doubled (multiplied by two). The other version takes a `Seq[String]`, maps the `String` elements to `Int`s by calling `toInt` on them (assuming the strings represent integers) and then calls the version of `double` that takes a `Seq[Int]` argument:

```
object Doubler {
  def double(seq: Seq[String]): Seq[Int] = double(seq map (_.toInt))
  def double(seq: Seq[Int]): Seq[Int] = seq map (_*2)
}
```

You'll get a compilation error that the two methods "have the same type after erasure." A somewhat ugly workaround is to examine the elements of the lists individually:

```
// src/main/scala/progscala2/typesystem/existentials/type-erasure-
workaround.sc

object Doubler {
  def double(seq: Seq[_]): Seq[Int] = seq match {
    case Nil => Nil
    case head +: tail => (toInt(head) * 2) +: double(tail)
  }

  private def toInt(x: Any): Int = x match {
    case i: Int => i
    case s: String => s.toInt
                                           "Unexpected list element
    case x => throw new RuntimeException(s$x"                        )
  }
}
```

When used in a type context like this, the expression `Seq[_]` is actually shorthand for the *existential type*, `Seq[T] forSome { type T }`. This is the most general case. We're saying the type parameter for the list could be any type. Table 14-1 lists some other examples that demonstrate the use of type bounds.

Table 14-1. Existential type examples

| Shorthand | Full | Description |
|---|---|---|
| `Seq[_]` | `Seq[T] forSome {type T}` | `T` can be any subtype of `Any`. |
| `Seq[_ <: A]` | `Seq[T] forSome {type T <: A}` | `T` can be any subtype of `A` (defined elsewhere). |
| `Seq[_ >: Z <: A]` | `Seq[T] forSome {type T >: Z <: A}` | `T` can be any subtype of `A` and supertype of `Z`. |

If you think about how Scala syntax for generics is mapped to Java syntax, you might have noticed that an expression like `java.util.List[_ <: A]` is structurally similar to the Java variance expression `java.util.List<? extends A>`. In fact, they are the same declarations. Although we said that variance behavior in Scala is defined at the declaration site, you can use existential type expressions in Scala to define call-site variance behavior, although it is not usually done.

You'll see type signatures like `Seq[_]` frequently in Scala code, where the type parameter can't be specified more specifically. You won't see the full `forSome` existential type syntax very often.

Existential types exist primarily to support Java generics while preserving correctness in Scala's type system. Type inference hides the details from us in most contexts.

## Recap and What's Next

This concludes a survey of the type system features you're most likely to encounter as you write Scala code and use libraries. Our primary focus was understanding the subtleties of object-oriented inheritance and why certain features like *variance* and *type bounds* are important. The next chapter continues the exploration with features that are less important to master as soon as possible.

If you would like a quick reference to type system and related concepts, bookmark "Scala's Types of Types" by my Typesafe colleague Konrad Malawski.