# Advanced types

«Previous Next»

This lesson covers:


- View bounds ("type classes")
- Other Type Bounds
- Higher kinded types & ad-hoc polymorphism
- F-bounded polymorphism / recursive types
- Structural types
- Abstract types members
- Type erasures & manifests
- Case study: Finagle


## View bounds ("type classes")

Sometimes you don't need to specify that one type is equal/sub/super another, just that you could fake it with conversions. A view bound specifies a type that can be "viewed as" another. This makes sense for an operation that needs to "read" an object but doesn't modify the object.

**Implicit** functions allow automatic conversion. More precisely, they allow on-demand function application when this can help satisfy type inference. e.g.:

```
scala> implicit def strToInt(x: String) = x.toInt
strToInt: (x: String)Int

scala> "123"
res0: java.lang.String = 123

scala> val y: Int = "123"
y: Int = 123

scala> math.max("123", 111)
res1: Int = 123
```

View bounds, like type bounds demand such a function exists for the given type. You specify a view bound with `<%` e.g.,

```
scala> class Container[A <% Int] { def addIt(x: A) = 123 + x }
defined class Container
```

This says that **A** has to be "viewable" as **Int**. Let's try it.

```
scala> (new Container[String]).addIt("123")
res11: Int = 246

scala> (new Container[Int]).addIt(123)
res12: Int = 246

scala> (new Container[Float]).addIt(123.2F)
<console>:8: error: could not find implicit value for evidence parameter of type (Float) => Int
       (new Container[Float]).addIt(123.2)
        ^
```

## Other type bounds

Methods can enforce more complex type bounds via implicit parameters. For example, `List` supports `sum` on numeric contents but not on others. Alas, Scala's numeric types don't all share a superclass, so we can't just say `T <: Number`. Instead, to make this work, Scala's math library defines an implicit `Numeric[T]` for the appropriate types T. Then in `List`'s definition uses it:

```
sum[B >: A](implicit num: Numeric[B]): B
```

If you invoke `List(1,2).sum()`, you don't need to pass a *num* parameter; it's set implicitly. But if you invoke `List("whoop").sum()`, it complains that it couldn't set `num`.

Methods may ask for some kinds of specific "evidence" for a type without setting up strange objects as with `Numeric`. Instead, you can use one of these type-relation operators:

| | |
|---|---|
| A =:= B | A must be equal to B |

| | |
|---|---|
| A <:< B | A must be a subtype of B |
| A <%< B | A must be viewable as B |

(If you get errors trying to use <:< or <%<, be aware that those went away in Scala 2.10. Scala School examples work with Scala 2.9.x . You can use a newer Scala, but expect errors.)

```
scala> class Container[A](value: A) { def addIt(implicit evidence: A =:= Int) = 123 + value }
defined class Container

scala> (new Container(123)).addIt
res11: Int = 246

scala> (new Container("123")).addIt
<console>:10: error: could not find implicit value for parameter evidence: =:=[java.lang.String,Int]
```

Similarly, given our previous implicit, we can relax the constraint to viewability:

```
scala> class Container[A](value: A) { def addIt(implicit evidence: A <%< Int) = 123 + value }
defined class Container

scala> (new Container("123")).addIt
res15: Int = 246
```

## Generic programming with views

In the Scala standard library, views are primarily used to implement generic functions over collections. For example, the “min” function (on **Seq[]**), uses this technique:

```
def min[B >: A](implicit cmp: Ordering[B]): A = {
  if (isEmpty)
    throw new UnsupportedOperationException("empty.min")

  reduceLeft((x, y) => if (cmp.lteq(x, y)) x else y)
}
```

The main advantages of this are:

- Items in the collections aren’t required to implement **Ordered**, but **Ordered** uses are still statically type checked.
- You can define your own orderings without any additional library support:

```
scala> List(1,2,3,4).min
res0: Int = 1

scala> List(1,2,3,4).min(new Ordering[Int] { def compare(a: Int, b: Int) = b compare a })
res3: Int = 4
```

As a sidenote, there are views in the standard library that translates **Ordered** into **Ordering** (and vice versa).

```
trait LowPriorityOrderingImplicits {
  implicit def ordered[A <: Ordered[A]]: Ordering[A] = new Ordering[A] {
    def compare(x: A, y: A) = x.compare(y)
  }
}
```

### Context bounds & implicitly[]

Scala 2.8 introduced a shorthand for threading through & accessing implicit arguments.

```
scala> def foo[A](implicit x: Ordered[A]) {}
foo: [A](implicit x: Ordered[A])Unit

scala> def foo[A : Ordered] {}
foo: [A](implicit evidence$1: Ordered[A])Unit
```

Implicit values may be accessed via **implicitly**

```
scala> implicitly[Ordering[Int]]
res37: Ordering[Int] = scala.math.Ordering$Int$@3a9291cf
```

Combined, these often result in less code, especially when threading through views.

## Higher-kinded types & ad-hoc polymorphism

Scala can abstract over “higher kinded” types. For example, suppose that you needed to use several types of containers for several types of data. You might define a `Container` interface that might be implemented by means of several container types: an `Option`, a `List`, etc. You want to define an interface for using values in these containers without nailing down the values' type.

This is analogous to function currying. For example, whereas “unary types” have constructors like `List[A]`, meaning we have to satisfy one “level” of type variables in order to produce a concrete types (just like an uncurried function needs to be supplied by only one argument list to be invoked), a higher-kinded type needs more.

```
scala> trait Container[M[_]] { def put[A](x: A): M[A]; def get[A](m: M[A]): A }

scala> val container = new Container[List] { def put[A](x: A) = List(x); def get[A](m: List[A]) = m.head }
container: java.lang.Object with Container[List] = $anon$1@7c8e3f75

scala> container.put("hey")
res24: List[java.lang.String] = List(hey)

scala> container.put(123)
res25: List[Int] = List(123)
```

Note that **Container** is polymorphic in a parameterized type (“container type”).

If we combine using containers with implicits, we get “ad-hoc” polymorphism: the ability to write generic functions over containers.

```
scala> trait Container[M[_]] { def put[A](x: A): M[A]; def get[A](m: M[A]): A }

scala> implicit val listContainer = new Container[List] { def put[A](x: A) = List(x); def get[A](m: List[A]) = m.head }

scala> implicit val optionContainer = new Container[Some] { def put[A](x: A) = Some(x); def get[A](m: Some[A]) = m.get }

scala> def tupleize[M[_]: Container, A, B](fst: M[A], snd: M[B]) = {
     | val c = implicitly[Container[M]]
     | c.put(c.get(fst), c.get(snd))
     | }
tupleize: [M[_],A,B](fst: M[A],snd: M[B])(implicit evidence$1: Container[M])M[(A, B)]

scala> tupleize(Some(1), Some(2))
res33: Some[(Int, Int)] = Some((1,2))

scala> tupleize(List(1), List(2))
res34: List[(Int, Int)] = List((1,2))
```

## F-bounded polymorphism

Often it's necessary to access a concrete subclass in a (generic) trait. For example, imagine you had some trait that is generic, but can be compared to a particular subclass of that trait.

```
trait Container extends Ordered[Container]
```

However, this now necessitates the compare method

```
def compare(that: Container): Int
```

And so we cannot access the concrete subtype, e.g.:

```
class MyContainer extends Container {
  def compare(that: MyContainer): Int
}
```

fails to compile, since we are specifying Ordered for **Container**, not the particular subtype.

To reconcile this, we instead use F-bounded polymorphism.

```
trait Container[A <: Container[A]] extends Ordered[A]
```

Strange type! But note now how Ordered is parameterized on **A**, which itself is **Container[A]**

So, now

```
class MyContainer extends Container[MyContainer] {
  def compare(that: MyContainer) = 0
}
```

They are now ordered:

```
scala> List(new MyContainer, new MyContainer, new MyContainer)
res3: List[MyContainer] = List(MyContainer@30f02a6d, MyContainer@67717334, MyContainer@49428ffa)

scala> List(new MyContainer, new MyContainer, new MyContainer).min
res4: MyContainer = MyContainer@33dfeb30
```

Given that they are all subtypes of **Container[_]**, we can define another subclass & create a mixed list of **Container[_]**:

```
scala> class YourContainer extends Container[YourContainer] { def compare(that: YourContainer) = 0 }
defined class YourContainer

scala> List(new MyContainer, new MyContainer, new MyContainer, new YourContainer)
res2: List[Container[_ >: YourContainer with MyContainer <: Container[_ >: YourContainer with MyContainer <: ScalaObject]]]
  = List(MyContainer@3be5d207, MyContainer@6d3fe849, MyContainer@7eab48a7, YourContainer@1f2f0ce9)
```

Note how the resulting type is now lower-bound by **YourContainer with MyContainer**. This is the work of the type inferencer. Interestingly- this type doesn't even need to make sense, it only provides a logical greatest lower bound for the unified type of the list. What happens if we try to use **Ordered** now?

```
(new MyContainer, new MyContainer, new MyContainer, new YourContainer).min
<console>:9: error: could not find implicit value for parameter cmp:
  Ordering[Container[_ >: YourContainer with MyContainer <: Container[_ >: YourContainer with MyContainer <: ScalaObject]]]
```

No **Ordered[]** exists for the unified type. Too bad.

## Structural types

Scala has support for **structural types** — type requirements are expressed by interface *structure* instead of a concrete type.

```
scala> def foo(x: { def get: Int }) = 123 + x.get
foo: (x: AnyRef{def get: Int})Int

scala> foo(new { def get = 10 })
res0: Int = 133
```

This can be quite nice in many situations, but the implementation uses reflection, so be performance-aware!

## Abstract type members

In a trait, you can leave type members abstract.

```
scala> trait Foo { type A; val x: A; def getX: A = x }
defined trait Foo

scala> (new Foo { type A = Int; val x = 123 }).getX
res3: Int = 123
```

```
scala> (new Foo { type A = String; val x = "hey" }).getX
res4: java.lang.String = hey
```

This is often a useful trick when doing dependency injection, etc.

You can refer to an abstract type variable using the hash-operator:

```
scala> trait Foo[M[_]] { type t[A] = M[A] }
defined trait Foo

scala> val x: Foo[List]#t[Int] = List(1)
x: List[Int] = List(1)
```

## Type erasures & manifests

As we know, type information is lost at compile time due to *erasure*. Scala features **Manifests**, allowing us to selectively recover type information. Manifests are provided as an implicit value, generated by the compiler as needed.

```
scala> class MakeFoo[A](implicit manifest: Manifest[A]) { def make: A = manifest.erasure.newInstance.asInstanceOf[A] }

scala> (new MakeFoo[String]).make
res10: String = ""
```

## Case study: Finagle

See: https://github.com/twitter/finagle

```
trait Service[-Req, +Rep] extends (Req => Future[Rep])

trait Filter[-ReqIn, +RepOut, +ReqOut, -RepIn]
  extends ((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
{
  def andThen[Req2, Rep2](next: Filter[ReqOut, RepIn, Req2, Rep2]) =
    new Filter[ReqIn, RepOut, Req2, Rep2] {
      def apply(request: ReqIn, service: Service[Req2, Rep2]) = {
        Filter.this.apply(request, new Service[ReqOut, RepIn] {
          def apply(request: ReqOut): Future[RepIn] = next(request, service)
          override def release() = service.release()
          override def isAvailable = service.isAvailable
        })
      }
    }

  def andThen(service: Service[ReqOut, RepIn]) = new Service[ReqIn, RepOut] {
    private[this] val refcounted = new RefcountedService(service)

    def apply(request: ReqIn) = Filter.this.apply(request, refcounted)
    override def release() = refcounted.release()
    override def isAvailable = refcounted.isAvailable
  }
}
```

A service may authenticate requests with a filter.

```
trait RequestWithCredentials extends Request {
  def credentials: Credentials
}

class CredentialsFilter(credentialsParser: CredentialsParser)
  extends Filter[Request, Response, RequestWithCredentials, Response]
{
  def apply(request: Request, service: Service[RequestWithCredentials, Response]): Future[Response] = {
    val requestWithCredentials = new RequestWrapper with RequestWithCredentials {
      val underlying = request
      val credentials = credentialsParser(request) getOrElse NullCredentials
    }
```

```
    service(requestWithCredentials)
  }
}
```

Note how the underlying service requires an authenticated request, and that this is statically verified. Filters can thus be thought of as service transformers.

Many filters can be composed together:

```
val upFilter =
  logTransaction     andThen
  handleExceptions   andThen
  extractCredentials andThen
  homeUser           andThen
  authenticate       andThen
  route
```

Type safely!

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;