Table of Contents for Programming Scala, 2nd Edition

Chapter 12. The Scala Collections Library

With this chapter we finish our discussion of standard library topics by discussing the design of the collections library. The techniques used in this design solve particular problems that arise when designing collections that combine functional and object-oriented features, and address other concerns.

The collections library was significantly redesigned for Scala version 2.8. See the Scaladoc for a detailed discussion of this redesign, which is still current.

Generic, Mutable, Immutable, Concurrent, and Parallel Collections, Oh My!

If you open the Scaladocs and type `Map` into the search box, you'll get five types! Fortunately, most are traits that declare or implement parts of the concrete `Map`s that you really care about. Most of the differences between those concrete types boil down to a few design questions you might have. Do you need mutability for performance (which you determined through profiling, of course)? Do you need concurrent access? Do you have operations that could be performed in parallel? Do you need the ability to iterate over the keys in sorted order, as well as perform the normal key-based lookup?

Table 12-1 lists the collection-related packages and their purposes. For the rest of this section, we'll drop the `scala` prefix, because you don't need it in import statements.

Table 12-1. The collection-related packages

Name	Description
`collection`	Defines the base traits and objects needed to use and extend Scala's collections library, including all definitions in subpackages. Most of the abstractions you'll work with are defined here.
`collection.concurrent`	Defines a `Map` trait and `TrieMap` class with atomic, lock-free access operations.
`collection.convert`	Defines types for wrapping Scala collections with Java collection abstractions and wrapping Java collections with Scala collection abstractions.
`collection.generic`	Defines reusable components used to build the specific mutable, immutable, etc. collections.
`collection.immutable`	Defines the immutable collections, the ones you'll use most frequently.
`collection.mutable`	Defines mutable collections. Most of the specific collection types are available in mutable and immutable forms, but not all.
`collection.parallel`	Defines reusable components used to build specific mutable and immutable collections that distribute processing to parallel threads.
`collection.parallel.immutable`	Defines parallel, immutable collections.
`collection.parallel.mutable`	Defines parallel, mutable collections.

Name	Description
`collection.script`	A deprecated package of tools for observing collection operations.

We won't discuss most of the types defined in these packages, but let's discuss the most important aspects of each package. We won't discuss the deprecated `collection.script` further.

The scala.collection Package

The types defined in `collection` declare and in some cases define the abstractions shared in common by the mutable and immutable sequential, mutable and immutable parallel, and concurrent collection types. That means, for example, that the destructive (mutation) operations that you'll find only in the `mutable` types aren't defined here. However, keep in mind that an actual collection instance at runtime might be mutable, where thread safety might be an issue.

Specifically, recall from Sequences that the default `Seq` you get through `Predef` is `collection.Seq`, while the other common types `Predef` exposes, such as `List`, `Map`, and `Set`, are specifically the `collection.immutable` variants. The reason `Predef` uses `collection.Seq` is so that Java arrays, which are mutable, can be treated uniformly as sequences. (`Predef` actually defines implicit conversions from Java arrays to `collection.mutable.ArrayOps`, which implements the sequence operations.) The plan is to change this in a future release of Scala to use the immutable `Seq` instead.

Unfortunately, for now, this also means that if a method declares that it returns an unqualified `Seq`, it might be returning a mutable instance. Similarly, if a method takes a `Seq` argument, a caller might pass a mutable instance to it.

If you prefer to use the safer `immutable.Seq` as the default, a common technique is to define a package object for your project with a type definition for `Seq` that effectively shadows the default definition in `Predef`, like the following:

```
// src/main/scala/progscala2/collections/safeseq/package.scala
package progscala2.collections
package object safeseq {
  type Seq[T] = collection.immutable.Seq[T]
}
```

Then, import its contents wherever needed. Note how the behavior changes for `Seq` in the following REPL session:

```
// src/main/scala/progscala2/collections/safeseq/safeseq.sc

scala> val mutableSeq1: Seq[Int] = List(1,2,3,4)
mutableSeq1: Seq[Int] = List(1, 2, 3, 4)

scala> val mutableSeq2: Seq[Int] = Array(1,2,3,4)
mutableSeq2: Seq[Int] = WrappedArray(1, 2, 3, 4)

scala> import progscala2.collections.safeseq._
import progscala2.collections.safeseq._

scala> val immutableSeq1: Seq[Int] = List(1,2,3,4)
immutableSeq1: safeseq.Seq[Int] = List(1, 2, 3, 4)

scala> val immutableSeq2: Seq[Int] = Array(1,2,3,4)
<console>:10: error: type mismatch;
 found   : Array[Int]
 required: safeseq.Seq[Int]
    (which expands to)  scala.collection.immutable.Seq[Int]
       val immutableSeq2: Seq[Int] = Array(1,2,3,4)
                                          ^
```

The first two `Seq` instances are the default `collection.Seq` exposed by `Predef`. The first references an immutable list and the second references a mutable (wrapped) Java array.

Then the new definition of `Seq` is imported, thereby shadowing the `Predef` definition.

Now the `Seq` type for the list is the `safeseq.Seq` alias, but we're not allowed to use it to reference an array, because the alias for `immutable.Seq` can't reference a mutable collection.

Either way, `Seq` is a convenient abstraction for any concrete collection where we just want the first few elements or we want to traverse from end to end.

The collection.concurrent Package

This package defines only two types, a `collection.concurrent.Map` trait and a hash-trie `collection.concurrent.TrieMap` class that implements the trait.

`Map` extends `collection.mutable.Map`, but it makes the operations atomic, so they support thread-safe, concurrent access.

The one implementation of `collection.mutable.Map` is a hash-trie class `collection.concurrent.TrieMap`. It is a concurrent. *lock-free* implementation of a hash array mapped trie. It aims for scalable concurrent insert and remove operations and memory efficiency.

The collection.convert Package

The types defined in this package are used to implement implicit conversion methods to wrap Scala collections as Java collections and vice versa. We discussed them in Scala's Built-in Implicits.

The collection.generic Package

Whereas `collection` declares abstractions for all collections, `collection.generic` provides reusable components for implementing the specific mutable, immutable, parallel, and concurrent collections. Most of the types are only of interest to implementers of collections.

The collection.immutable Package

You'll work with collections defined in the `immutable` package most of the time. They provide single-threaded (as opposed to parallel) operations. Because they are immutable, they are thread-safe. Table 12-2 provides an alphabetical list of the most commonly used types in this package.

Table 12-2. Most commonly used immutable collections

Name	Description
`BitSet`	Memory-efficient sets of nonnegative integers. The entries are represented as variable-size arrays of bits packed into 64-bit words. The largest entry determines the memory footprint of the set.
`HashMap`	Maps implemented with hash trie for the keys.
`HashSet`	Sets implemented with a hash trie.
`List`	A trait for linked lists, with *O(1)* head access and *O(n)* access to interior elements. The companion object has `apply` and other "factory" methods for constructing instances of implementing subclasses.
`ListMap`	An immutable map backed by a list.
`ListSet`	An immutable set backed by a list.
`Map`	Trait for all key-value, immutable maps, with *O(1)* random access. The companion object has `apply` and other "factory" methods for constructing instances of implementing subclasses.
`Nil`	An object for empty lists.
`NumericRange`	Generalizes ranges to arbitrary integral types. `NumericRange` is a more generic version of the `Range` class that works with arbitrary types. It must be supplied with an Integral implementation of the range type.
`Queue`	An immutable FIFO (first-in, first-out) queue.
`Seq`	A trait for immutable sequences. The companion object has `apply` and other "factory" methods for constructing instances of implementing subclasses.
`Set`	A trait that declares the operations for immutable sets. The companion object has `apply` and other "factory" methods for constructing instances of implementing subclasses.
`SortedMap`	The trait for immutable maps with an iterator that traverses the elements in sorted order. The companion object has `apply` and other "factory" methods for constructing instances of implementing subclasses.
`SortedSet`	The trait for immutable sets with an iterator that traverses the elements in sorted order. The companion object has `apply` and other "factory" methods for constructing instances of implementing subclasses.
`Stack`	An immutable LIFO (last-in, first-out) stack.
`Stream`	A lazy list of values, thereby able to support a potentially infinite sequence of values.

Name	Description
`TreeMap`	An immutable map with underlying red-black tree storage with *O(log(n))* operations.
`TreeSet`	An immutable set with underlying red-black tree storage with *O(log(n))* operations.
`Vector`	The default implementation of immutable, indexed sequences.

Bitsets are sets of nonnegative integers that are represented as variable-size arrays of bits packed into 64-bit words. The memory footprint of a bitset is determined by the largest number stored in it.

`Vector` is implemented using as a tree-based, *persistent data structure*, as discussed in What About Making Copies?. It provides excellent performance, with amortized *O(1))* operations.

It's worth looking at the source code for `Map`, particularly the companion object. Notice that several implementations of `Map`s are declared for the special cases of zero to four key-value pairs. When you call `Map.apply` (defined in a parent trait), it tries to create an instance that's optimal for the actual data in the `Map`.

The scala.collection.mutable Package

There are times when you'll need a mutable collection with single-threaded operations. We've discussed how immutability should be the default choice. The mutation operations on these collections are *not* thread-safe. However, principled and careful use of mutable data can be appropriate for performance and other reasons. Table 12-3 provides an alphabetical list of the most commonly used collections in the `mutable` package.

Table 12-3. Most commonly used mutable collections

Name	Description
`AnyRefMap`	Map for `AnyRef` keys that uses a hash table with open addressing. Most operations are generally faster than for `HashMap`.
`ArrayBuffer`	A buffer class that uses an array for internal storage. Append, update, and random access take *O(1)* (amortized) time. Prepends and removes are *O(n)*.
`ArrayOps`	A wrapper class for Java arrays that implements the sequence operations.
`ArrayStack`	A stack backed by an array. It's faster than the general-purpose `Stack`.
`BitSet`	Memory-efficient sets of nonnegative integers. See the discussion of `immutable.BitSet` in Table 12-2.
`HashMap`	The mutable version of a hash-table based map.
`HashSet`	The mutable version of a hash-table based set.
`HashTable`	The trait used to implement mutable collections based on hash tables.
`ListMap`	A mutable map backed by a list.
`LinkedHashMap`	A hash-table based map where the elements can be traversed in their insertion order.
`LinkedHashSet`	A hash-table based set where the elements can be traversed in their insertion order.

Name	Description
`LongMap`	A mutable map backed by a hash table with open addressing where the keys are `Long`s. Most operations are substantially faster than for `HashMap`.
`Map`	A trait for the mutable version of the `Map` abstraction. The companion object has `apply` and other “factory” methods for constructing instances of implementing subclasses.
`MultiMap`	The mutable `Map` where multiple values can be assigned to the same key.
`PriorityQueue`	A heap-based, mutable priority queue. For the elements of type `A`, there must be an implicit `Ordering[A]` instance.
`Queue`	A mutable FIFO (first-in, first-out) queue.
`Seq`	A trait for mutable sequences. The companion object has `apply` and other “factory” methods for constructing instances of implementing subclasses.
`Set`	A trait that declares the operations for mutable sets. The companion object has `apply` and other “factory” methods for constructing instances of implementing subclasses.
`SortedSet`	The trait for mutable sets with an iterator that traverses the elements in sorted order. The companion object has `apply` and other “factory” methods for constructing instances of implementing subclasses.
`Stack`	A mutable LIFO (last-in, first-out) stack.
`TreeSet`	A mutable set with underlying red-black tree storage with *O(log(n))* operations.
`WeakHashMap`	A mutable hash map with references to entries that are weakly reachable. Entries are removed from this map when the key is no longer (strongly) referenced. This class wraps `WeakHashMap`.
`WrappedArray`	A wrapper class for Java arrays that implements the sequence operations.

`WrappedArray` is almost identical to `ArrayOps`. The difference is in methods that return a new `Array`. For `ArrayOps`, those methods return a new `Array[T]`, while for `WrappedArray`, they return a new `WrappedArray[T]`. Hence, `ArrayOps` is better for contexts where the user is expecting an `Array`, but when the user doesn’t care, a `WrappedArray` is more efficient if a sequence of transformations is required, because repeated “boxing” and “unboxing” of the `Array` in an `ArrayOps` (or `WrappedArray`) is avoided.

The scala.collection.parallel Package

The idea behind the parallel collections is to exploit modern multicore systems that provide parallel hardware threads. Any collection operations that can be performed in parallel could exploit this parallelism, in principle.

Specifically, the collection is split into pieces, combinator operations (e.g., `map`) are applied to the pieces, and then the results are combined to create the final result. That is, a *divide and conquer* strategy is used.

In practice, the parallel collections are not widely used, because the overhead of parallelization can overwhelm the advantages in many situations and not all operations can be parallelized. The overhead includes thread scheduling and the task of dividing the data into chunks, then combining results later on. Often, unless the collection is large, serial execution will be faster. So, be sure to profile real-world scenarios in your environment to determine whether your target collections are large enough and parallel operations perform fast enough to use a parallel collection.

For a concrete parallel collection, you can either construct an instance directly using the same idioms as for the nonparallel counterpart, or you can call the `par` method on the corresponding, nonparallel collection.

The parallel collections are organized like the nonparallel variants, as well. They have common traits and classes defined in the `scala.collection.parallel` package, with immutable concrete collections in the `immutable` child package and mutable concrete collections in the `mutable` child package.

Finally, it's essential to understand that parallelization means that the order of nested operations is undefined. Consider the following example, where we combine the numbers from 1 to 10 into a string:

```
// src/main/scala/progscala2/collections/parallel.sc

                                             "$s1 -
scala> ((1 to 10) fold "") ((s1, s2) => s$s2"        )
             " - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
res0: Any = 10"

                                             "$s1 -
scala> ((1 to 10) fold "") ((s1, s2) => s$s2"        )
             " - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
res1: Any = 10"

                                                 "$s1 -
scala> ((1 to 10).par fold "") ((s1, s2) => s$s2"        )
             " - 1 -   - 2 -   - 3 - 4 - 5 -   - 6 -   - 7 -   - 8 -   - 9 -
res2: Any = - 10"

                                                 "$s1 -
scala> ((1 to 10).par fold "") ((s1, s2) => s$s2"        )
             " - 1 -   - 2 -   - 3 -   - 4 - 5 -   - 6 -   - 7 -   - 8 - 9 -
res3: Any = 10"
```

For the nonparallel case, the same result is returned consistently, but *not* for repeated invocations when using a parallel collection!

However, addition works fine:

```
scala> ((1 to 10) fold 0) ((s1, s2) => s1 + s2)
res4: Int = 55

scala> ((1 to 10) fold 0) ((s1, s2) => s1 + s2)
res5: Int = 55

scala> ((1 to 10).par fold 0) ((s1, s2) => s1 + s2)
res6: Int = 55

scala> ((1 to 10).par fold 0) ((s1, s2) => s1 + s2)
res7: Int = 55
```

All runs yield the same result.

To be specific, the operation must be *associative* to yield predictable results for parallel operations. That is, `(a+b)+c == a+(b+c)` must always be true. The inconsistent spacing and "-" separators when building the strings in parallel indicate that the string composition operation used here isn't associative. In each run with the parallel collection, the collection is subdivided differently and somewhat unpredictably.

Addition is associative. It's also commutative, but that isn't necessary. Note that the string examples compose the elements in a predictable, left to right order, indicative of the fact that commutativity isn't required.

Because the parallel collections have nonparallel counterparts that we've already discussed, I won't list the specific types here. Instead, see the Scaladocs for the `parallel`, `parallel.immutable`, and `parallel.mutable` packages. The Scaladocs also discuss other usage issues that we haven't discussed here.

Choosing a Collection

Aside from the decision to use a mutable versus immutable and nonparallel versus parallel collection, which collection type should you pick for a given situation?

Here are some informal criteria and options to consider. It's worth studying the *O(n)* performance of different operations for the collection types. See the Scaladoc for an exhaustive list. There is also a useful StackOverflow discussion on choosing a collection.

I'll use the convention `immutable.List` (`mutable.LinkedList`) to indicate immutable and mutable options, when there are both.

Do you need ordered, traversable sequences? Consider an `immutable.List` (`mutable.LinkedList`), an `immutable.Vector`, or a `mutable.ArrayBuffer`.

`List`s provide *O(1)* prepending and reading of the head element, but *O(n)* appending and reading of internal elements.

Because `Vector` is a *persistent data structure* (as discussed previously), it is effectively *O(1)* for *all* operations.

`ArrayBuffer` is better if you need random access. Appending, updating, and random access all take *O(1)* (amortized) time, but prepending and deleting are *O(n)*.

So, when you need a sequence, you'll almost always use a `List`, when you mostly work with the head elements, and a `Vector` for more general access patterns. `Vector` is a powerful, general-purpose collection with excellent all-around performance. However, there are some situations where an `ArrayBuffer` will provide lower constant-time overhead and hence higher performance.

The other general scenario is the need for *O(1)*, key-based storage and retrieval, i.e., values stored by keys in an `immutable.Map` (`mutable.Map`). Similarly, `immutable.Set` (`mutable.Set`) is used to test for the existence of a value.

Design Idioms in the Collections Library

A number of *idioms* are used in the collections library to solve design problems and promote reuse. Let's discuss them and along the way, learn more about the "helper" types in the library that are used in the implementations.

Builder

I mentioned previously that the mutable collections are an appropriate compromise for performance when used carefully. In fact, the collections API uses them internally to build new output collections in operations like `map`.

Implementations of the `collection.mutable.Builder` trait are used internally to construct new instances during operations like `map`.

`Builder` has the following signature:

```
trait Builder[-Elem, +To] {
  def +=(elem: Elem): Builder.this.type
  def clear()
  def result(): To
  ...
// Other methods derived from these three abstract
methods.
}
```

The unusual `Builder.this.type` signature is a *singleton type*. It ensures that the `+=` method can only return the `Builder` instance it was called on, i.e., `this`. If an implementation attempts to return a new instance of a `Builder`, for example, it won't type check! We'll study singleton types in Singleton Types.

Here is an example implementation of a builder for `Lists`:

```
//
src/main/scala/progscala2/collections/ListBuilder.sc
import collection.mutable.Builder

class ListBuilder[T] extends Builder[T, List[T]] {

  private var storage = Vector.empty[T]

  def +=(elem: T) = {
    storage = storage :+ elem
    this
  }

  def clear(): Unit = { storage = Vector.empty[T] }

  def result(): List[T] = storage.toList
}

val lb = new ListBuilder[Int]
(1 to 3) foreach (i => lb += i)
lb.result
// Result: List(1, 2,
3)
```

A more efficient choice for the internal storage than `Vector` could be made, but it illustrates the point.

CanBuildFrom

Consider this simple example of mapping over a list of numbers:

```
scala> List(1, 2, 3, 4, 5) map (2 * _)
res0: List[Int] = List(2, 4, 6, 8, 10)
```

The *simplified* signature of this method in `List` is the following:

```
map[B](f: (A) => B): List[B]
```

However, the standard library exploits reuse where possible. Recall from Constraining Allowed Instances that `map` is actually defined in `scala.collection.TraversableLike`, which is a mixin trait for `List`. The actual signature for `map` is the following:

```
trait TraversableLike[+A, +Repr] extends ... {
  ...
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That =
{...}
}
```

`Repr` is the type of the collection used internally to store the items. `B` is the type of elements created by the function `f`. `That` is the type parameter of the target collection we want to create, which may or may not be the same as the original collection.

`TraversableLike` knows nothing of subtypes like `List`, but it can construct a new `List` to return because the implicit `CanBuildFrom` instance encapsulates the details.

`CanBuildFrom` is a trait for factories that create `Builder` instances, which do that actual incremental construct of new collections.

A drawback of using the `CanBuildFrom` technique is the extra complexity in the actual method signature. However, besides enabling object-oriented reuse of operations like `map`, `CanBuildFrom` modularizes and generalizes construction in other useful ways.

For example, a `CanBuildFrom` instance might instantiate `Builders` for a different concrete collection to be returned. Usually a new collection of the same type is returned, or perhaps a subtype that might be more efficient for the given elements.

For example, a `Map` with a lot of elements is best implemented by storing the keys in a hash table, providing amortized *O(1)* storage and retrieval. However, for a small `Map`, it can be faster to simply store the elements in an array or list, where the *O(n)* retrieval for small *n* is actually faster than the *O(1)* retrieval from a hash table, due to the larger constant factor overhead of the latter.

There are other cases where the input collection type can't be used for the output collection. Consider the following example:

```
scala> val set = collection.BitSet(1, 2, 3, 4, 5)
set: scala.collection.BitSet = BitSet(1, 2, 3, 4, 5)

scala> set map (_.toString)
res0: scala.collection.SortedSet[String] = TreeSet(1, 2, 3, 4, 5)
```

A `BitSet` can only hold integers, so if we map it to a a set of strings, the implicit `CanBuildFrom` has to instantiate a different output collection, a `SortedSet` in this case.

Similarly, for strings (sequences of characters), we encounter the following:

```
scala> "xyz" map (_.toInt)
res0: scala.collection.immutable.IndexedSeq[Int] = Vector(120, 121, 122)
```

Another benefit of `CanBuildFrom` is the ability of the instance to carry other context information that might not be known to the original collection or not suitable for it to carry around. For example, when working with a distributed computing API, special `CanBuildFrom` instances might be used for constructing collection instances that are optimal for serialization to remote processes.

Like Traits

We saw that `Builder` and `CanBuildFrom` take type parameters for the output collection. To support specifying these type parameters and to promote implementation reuse, most of the collections you know actually mix in corresponding `…Like` traits that add the appropriate return-type parameter and provide implementations of common methods.

For example, here is how `collection.immutable.Seq` is declared:

```
trait Seq[+A] extends Iterable[A] with collection.Seq[A]
 with GenericTraversableTemplate[A, Seq] with SeqLike[A, Seq[A
]]
 with Parallelizable[A, ParSeq[A]]
```

Note that `collection.SeqLike` is parameterized with both the element type `A` and `Seq[A]` itself. The latter parameter is used to constrain the allowed `CanBuildFrom` instances that can be used in methods like `map`. This trait also implements most of the familiar methods on `Seq`.

I encourage you to examine the Scaladoc entry for `collection.immutable.Seq` and some of the other common collection types we've discussed. Click the links to the other traits to see what they do. These traits and the traits they mix in form a nontrivial tree of types. Fortunately, most of these details are irrelevant for actually using the common concrete collection types.

To conclude, these are the three most important design idioms used in the collections:

1. `Builder` to abstract over construction
2. `CanBuildFrom` to provide implicit factories for constructing suitable `Builder` instances for a given context
3. `Like` traits that add the necessary return type parameter needed by `Builder` and `CanBuildFrom`, as well as providing most of the method implementations

If you build your own collections, you'll want to follow these idioms. Also, recall from Chapter 7 that if your collections implement `foreach`, `map`, `flatMap`, and `withFilter`, they can be used in `for` comprehensions, just like the built-in collections.

Specialization for Value Types

One benefit of Scala's uniform treatment of value types (e.g., `Int`, `Float`, etc.) and reference types is the ability to declare instances of parameterized types with the value types, e.g., `List[Int]`. In contrast, Java requires the boxed types to be used for containers, e.g., `List<Integer>`. Boxing requires extra memory per object and extra time for memory management. Also, primitive values that are contiguous in memory can improve cache hit ratios and therefore performance, for some algorithms.

Hence, it's common in data-centric Java libraries, like those libraries for *Big Data* applications, to have a long list of custom container types specialized for each of the primitive types, or perhaps just a few, like `long` and `double`. That is, you'll see a class dedicated to vectors of `long`s, a class dedicated to vectors of `double`s, and so forth. So, the size of these libraries is much bigger than it would be if Java supported parameterized containers of primitives, but the performance of the custom primitive containers are often more than ten times better than the corresponding `Object`-based implementations.

Unfortunately, although Scala lets us declare instances of containers with value types, it doesn't actually solve this problem. Because of *type erasure*, the fact that the JVM doesn't retain information about the type of the container's elements, the elements are assumed to be `Object`s and a single implementation of the container is used for all element types. So, a `List[Double]` will still use boxed `Double`s, for example.

Wouldn't it be great to have a mechanism to tell the compiler to generate "specialized" implementations of such containers that are optimized for desired primitives? In fact, Scala has a `@specialized` annotation for this purpose. It tells the compiler to generate a custom implementation for the value types listed in the annotation call:

```
class SpecialVector[@specialized(Int, Double, Boolean) T]
{...}
```

In this example, specialized versions of `SpecialVector` will be generated for `Int`, `Double`, and `Boolean`. If the list is omitted, specialized versions of the type will be generated for all the value types.

However, practical experience with `@specialized` since it was introduced has exposed some limitations. First, it can result in a lot of generated code, so injudicious use of `@specialized` can make a library excessively large.

Second, there are several design flaws in the implementation (see this recent presentation for a more detailed discussion of the issues). If a field is declared of the generic type in the original container, it is not converted to a primitive field in the specialization. Rather, a *duplicate* field of the appropriate primitive type is created, leading to bugs. Another flaw is that the specialized containers are implemented as subclasses of the original generic container. This breaks when the generic container and a subtype are both specialized. The specialized versions should have the same inheritance relationship, but this can't be supported due to the JVM's single inheritance model.

So, because of these limitations, the Scala library makes limited use of `@specialized`. Most uses are for the `FunctionN`, `TupleN`, and `ProductN` types, plus a few collections.

Before we discuss an emerging alternative, note that there is also an `@unspecialized` annotation for methods. It is used when the type has the `@specialized` annotation, but you don't want a specialized version of the method generated. You might use this annotation when the performance benefit doesn't outweigh the extra code size.

Miniboxing

An alternative mechanism, called *miniboxing*, is under development. It attempts to remove the limitations of specialization. It will most likely appear in a future version of Scala, although it is available for experimentation now as a compiler plug-in, so it's worth discussing now.

Once the plug-in is installed, it is used in essentially the same way as `@specialized`:

```
class SpecialVector[@miniboxed(Int, Double, Boolean) T]
{...}
```

It reduces code bloat by converting a generic container into a trait with two subclasses, one to use for primitive values and one for reference (i.e., `AnyRef`) values. The primitive version exploits the observation that an 8-byte value can hold a value of any of the primitive types. A “tag” is added to indicate how the 8-byte value should be interpreted. Hence, it behaves as a *tagged union*. Therefore, it’s not necessary to have a separate instantiation for each primitive type. The reference implementation works as before.

By converting the original container to a trait, any preexisting inheritance relations are preserved in the two class instantiations. For example, if we have two parameterized containers `class C[T]` and `D[T]`, with `class D[T] extends C[T]`, and both are specialized, then the generated code looks conceptually like the following:

```
trait C[T]                                // was class C[T]
class C_primitive[T] extends C[T]         // T is an AnyVal
class C_anyref[T] extends C[T]            // T is an AnyRef

trait D[T] extends C[T]
// was class D[T]
class D_primitive[T] extends C_primitive[T] with D[T]
// T is an AnyVal
class D_anyref[T] extends C_anyref[T] with D[T]
// T is an AnyRef
```

In the meantime, you can still use `@specialized` when you need the performance. Just be careful about the extra space required and the design limitations described earlier.

Recap and What’s Next

We rounded out our understanding of the Scala collections library, including the distinctions between the mutable, immutable, and parallel variants, how to convert to and from Java collections, and the important, unfinished topic of enabling the collections to work efficiently with JVM primitive values, where the overhead of boxing is avoided.

Before we tackle the major topic of Scala’s type system, the next chapter covers a topic you should know about, even though it won’t be a daily “concern”: Scala’s rich support for fine-grained control over *visibility*. Scala goes well beyond Java’s `public`, `protected`, `private`, and default package scoping capabilities.