Chapter 19

Type Parameterization

In this chapter, we'll explain the details of type parameterization in Scala. Along the way we'll demonstrate some of the techniques for information hiding introduced in Chapter 13 by using a concrete example: the design of a class for purely functional queues. We're presenting type parameterization and information hiding together, because information hiding can be used to obtain more general type parameterization variance annotations.

Type parameterization allows you to write generic classes and traits. For example, sets are generic and take a type parameter: they are defined as Set[T]. As a result, any particular set instance might be a Set[String], a Set[Int], *etc.*, but it must be a set of *something*. Unlike Java, which allows raw types, Scala requires that you specify type parameters. Variance defines inheritance relationships of parameterized types, such as whether a Set[String], for example, is a subtype of Set[AnyRef].

The chapter contains three parts. The first part develops a data structure for purely functional queues. The second part develops techniques to hide internal representation details of this structure. The final part explains variance of type parameters and how it interacts with information hiding.

19.1 FUNCTIONAL QUEUES

A functional queue is a data structure with three operations:

head	returns the first element of the queue
tail	returns a queue without its first element
enqueue	returns a new queue with a given element appended at the end

Unlike a mutable queue, a functional queue does not change its contents when an element is appended. Instead, a new queue is returned that contains the element. The goal of this chapter will be to create a class, which we'll name Queue, that works like this:

```
scala> val q = Queue(1, 2, 3)
q: Queue[Int] = Queue(1, 2, 3)

scala> val q1 = q enqueue 4
q1: Queue[Int] = Queue(1, 2, 3, 4)

scala> q
res0: Queue[Int] = Queue(1, 2, 3)
```

If Queue were a mutable implementation, the enqueue operation in the second input line above would affect the contents of q; in fact both the result, q1, and the original queue, q, would contain the sequence 1, 2, 3, 4 after the operation. But for a functional queue, the appended value shows up only in the result, q1, not in the queue, q, being operated on.

Purely functional queues also have some similarity with lists. Both are so called fully persistent data structures, where old versions remain available even after extensions or modifications. Both support head and tail operations. But where a list is usually extended at the front, using a :: operation, a queue is extended at the end, using enqueue.

How can this be implemented efficiently? Ideally, a functional (immutable) queue should not have a fundamentally higher overhead than an imperative (mutable) one. That is, all three operations, head, tail, and enqueue, should operate in constant time.

One simple approach to implement a functional queue would be to use a list as representation type. Then head and tail would just translate into the same operations on the list, whereas enqueue would be concatenation.

This would give the following implementation:

```
class SlowAppendQueue[T](elems: List[T]) { // Not efficient
  def head = elems.head
  def tail = new SlowAppendQueue(elems.tail)
  def enqueue(x: T) = new SlowAppendQueue(elems ::: List(x))
}
```

The problem with this implementation is in the enqueue operation. It takes time proportional to the number of elements stored in the queue. If you want constant time append, you could also try to reverse the order of the elements in the representation list, so that the last element that's appended comes first in the list. This would lead to the following implementation:

```
class SlowHeadQueue[T](smele: List[T]) { // Not efficient
  // smele is elems reversed
  def head = smele.last
  def tail = new SlowHeadQueue(smele.init)
  def enqueue(x: T) = new SlowHeadQueue(x :: smele)
}
```

Now enqueue is constant time, but head and tail are not. They now take time proportional to the number of elements stored in the queue.

Looking at these two examples, it does not seem easy to come up with an implementation that's constant time for all three operations. In fact, it looks doubtful that this is even possible! However, by combining the two operations you can get very close. The idea is to represent a queue by two lists, called leading and trailing. The leading list contains elements towards the front, whereas the trailing list contains elements towards the back of the queue in reversed order. The contents of the whole queue are at each instant equal to "leading ::: trailing.reverse".

Now, to append an element, you just cons it to the trailing list using the :: operator, so enqueueis constant time. This means that, when an initially empty queue is constructed from successive enqueue operations, the trailing list will grow whereas the leading list will stay empty. Then, before the first head or tail operation is performed on an empty leading list, the whole trailing list is copied to leading, reversing the order of the elements. This is done in an operation called mirror. Listing 19.1 shows an implementation of queues that uses this approach.

```
class Queue[T](
  private val leading: List[T],
  private val trailing: List[T]
) {
  private def mirror =
    if (leading.isEmpty)
      new Queue(trailing.reverse, Nil)
    else
      this

  def head = mirror.leading.head

  def tail = {
    val q = mirror
    new Queue(q.leading.tail, q.trailing)
  }

  def enqueue(x: T) =
    new Queue(leading, x :: trailing)
}
```

Listing 19.1 - A basic functional queue.

What is the complexity of this implementation of queues? The mirror operation might take time proportional to the number of queue elements, but only if list leading is empty. It returns directly if leading is non-empty. Because head and tail call mirror, their complexity might be linear in the size of the queue, too. However, the longer the queue gets, the less often mirror is called.

Indeed, assume a queue of length *n* with an empty leading list. Then mirror has to reverse-copy a list of length *n*. However, the next time mirror will have to do any work is once the leading list is empty again, which will be the case after *n* tail operations. This means you can "charge" each of these *n* tail operations with one *n*'th of the complexity of mirror, which means a constant amount of work. Assuming that head, tail, and enqueue operations appear with about the same frequency, the amortized complexity is hence constant for each operation. So functional queues are asymptotically just as efficient as mutable ones.

Now, there are some caveats that need to be attached to this argument. First, the discussion was only about asymptotic behavior; the constant factors might well be somewhat different. Second, the argument rested on the fact that head, tail and enqueue are called with about the same frequency. If head is called much more often than the other two operations, the argument is not valid, as each call to head might involve a costly re-organization of the list with mirror. The second caveat can be avoided; it is possible to design functional queues so that in a sequence of successive head operations only the first one might require a re-organization. You will find out at the end of this chapter how this is done.

19.2 INFORMATION HIDING

The implementation of Queue shown in Listing 19.1 is now quite good with regards toefficiency. You might object, though, that this efficiency is paid for by exposing a needlessly detailed implementation. The Queue constructor, which is globally accessible, takes two lists as parameters, where one is reversed—hardly an intuitive representation of a queue. What's needed is a way to hide this constructor from client code. In this section, we'll show you some ways to accomplish this in Scala.

Private constructors and factory methods

In Java, you can hide a constructor by making it private. In Scala, the primary constructor does not have an explicit definition; it is defined implicitly by the class parameters and body. Nevertheless, it is still possible to hide the primary constructor by adding a private modifier in front of the class parameter list, as shown in Listing 19.2:

```
class Queue[T] private (
  private val leading: List[T],
  private val trailing: List[T]
)
```

Listing 19.2 - Hiding a primary constructor by making it private.

The private modifier between the class name and its parameters indicates that the constructor of Queue is private: it can be accessed only from within the class itself and its companion object. The class name Queue is still public, so you can use it as a type, but you cannot call its constructor:

```
scala> new Queue(List(1, 2), List(3))
<console>:9: error: constructor Queue in class Queue cannot
be accessed in object $iw
              new Queue(List(1, 2), List(3))
              ^
```

Now that the primary constructor of class Queue can no longer be called from client code, there needs to be some other way to create new queues. One possibility is to add an auxiliary constructor, like this:

```
def this() = this(Nil, Nil)
```

The auxiliary constructor shown in the previous example builds an empty queue. As a refinement, the auxiliary constructor could take a list of initial queue elements:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Recall that T* is the notation for repeated parameters, as described in Section 8.8.

Another possibility is to add a factory method that builds a queue from such a sequence of initial elements. A neat way to do this is to define an object Queue that has the same name as the class being defined and contains an apply method, as shown in Listing 19.3:

```
object Queue {
  // constructs a queue with initial elements `xs'
  def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)
}
```

Listing 19.3 - An apply factory method in a companion object.

By placing this object in the same source file as class Queue, you make the object a companion object of the class. You saw in Section 13.5 that a companion object has the same access rights as its class. Because of this, the apply method in object Queue can create a new Queue object, even though the constructor of class Queue is private.

Note that, because the factory method is called apply, clients can create queues with an expression such as Queue(1, 2, 3). This expression expands to Queue.apply(1, 2, 3) since Queue is an object instead of a function. As a result, Queue looks to clients as if it was a globally defined factory method. In reality, Scala has no globally visible methods; every method must be contained in an object or a class. However, using methods named apply inside global objects, you can support usage patterns that look like invocations of global methods.

```
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def enqueue(x: T): Queue[T]
}

object Queue {

  def apply[T](xs: T*): Queue[T] =
    new QueueImpl[T](xs.toList, Nil)

  private class QueueImpl[T](
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T] {

    def mirror =
      if (leading.isEmpty)
        new QueueImpl(trailing.reverse, Nil)
      else
        this

    def head: T = mirror.leading.head

    def tail: QueueImpl[T] = {
      val q = mirror
      new QueueImpl(q.leading.tail, q.trailing)
    }

    def enqueue(x: T) =
      new QueueImpl(leading, x :: trailing)
  }
}
```

Listing 19.4 - Type abstraction for functional queues.

An alternative: private classes

Private constructors and private members are one way to hide the initialization and representation of a class. Another more radical way is to hide the class itself and only export a trait that reveals the public interface of the class. The code in Listing 19.4 implements this design. There's a trait Queue, which declares the methods head, tail, and enqueue. All three methods are implemented in a subclass QueueImpl, which is itself a private inner class of objectQueue. This exposes to clients the same information as before, but using a different technique. Instead of hiding individual constructors and methods, this version hides the whole implementation class.

19.3 VARIANCE ANNOTATIONS

Queue, as defined in Listing 19.4, is a trait, but not a type. Queue is not a type because it takes a type parameter.

As a result, you cannot create variables of type Queue:

```
scala> def doesNotCompile(q: Queue) = {}
<console>:8: error: class Queue takes type parameters
       def doesNotCompile(q: Queue) = {}
                             ^
```

Instead, trait Queue enables you to specify *parameterized* types, such as Queue[String],Queue[Int], or Queue[AnyRef]:

```
scala> def doesCompile(q: Queue[AnyRef]) = {}
doesCompile: (q: Queue[AnyRef])Unit
```

Thus, Queue is a trait and Queue[String] is a type. Queue is also called a *type constructor* because you can construct a type with it by specifying a type parameter. (This is analogous to constructing an object instance with a plain-old constructor by specifying a value parameter.) The type constructor Queue "generates" a family of types, which includes Queue[Int],Queue[String], and Queue[AnyRef].

You can also say that Queue is a *generic* trait. (Classes and traits that take type parameters are "generic," but the types they generate are "parameterized," not generic.) The term "generic" means that you are defining many specific types with one generically written class or trait. For example, trait Queue in Listing 19.4 defines a generic queue. Queue[Int] and Queue[String],*etc.*, would be the specific queues.

The combination of type parameters and subtyping poses some interesting questions. For example, are there any special subtyping relationships between members of the family of types generated by Queue[T]? More specifically, should a Queue[String] be considered a subtype of Queue[AnyRef]? Or more generally, if S is a subtype of type T, then should Queue[S] be considered a subtype of Queue[T]? If so, you could say that trait Queue is *covariant* (or "flexible") in its type parameter T. Or, since it just has one type parameter, you could say simply that Queues are covariant. Covariant Queues would mean, for example, that you could pass a Queue[String] to the doesCompile method shown previously, which takes a value parameter of type Queue[AnyRef].

Intuitively, all this seems OK, since a queue of Strings looks like a special case of a queue ofAnyRefs. In Scala, however, generic types have by default *nonvariant* (or "rigid") subtyping. That is, with Queue defined as in Listing 19.4, queues with different element types would never be in a subtype relationship. A Queue[String] would not be usable as a Queue[AnyRef]. However, you can demand covariant (flexible) subtyping of queues by changing the first line of the definition of class Queue like this:

```
trait Queue[+T] { ... }
```

Prefixing a formal type parameter with a + indicates that subtyping is covariant (flexible) in that parameter. By adding this single character, you are telling Scala that you wantQueue[String], for example, to be considered a subtype of Queue[AnyRef]. The compiler will check that Queue is defined in a way that this subtyping is sound.

Besides +, there is also a prefix -, which indicates *contravariant* subtyping. If Queue were defined like this:

```
trait Queue[-T] { ... }
```

then if T is a subtype of type S, this would imply that Queue[S] is a subtype of Queue[T] (which in the case of queues would be rather surprising!). Whether a type parameter is covariant, contravariant, or nonvariant is called the parameter's *variance*. The + and - symbols you can place next to type parameters are called *variance annotations*.

In a purely functional world, many types are naturally covariant (flexible). However, the situation changes once you introduce mutable data. To find out why, consider the simple type of one-element cells that can be read or written, shown in Listing 19.5.

```
class Cell[T](init: T) {
  private[this] var current = init
  def get = current
  def set(x: T) = { current = x }
}
```

Listing 19.5 - A nonvariant (rigid) Cell class.

The Cell type of Listing 19.5 is declared nonvariant (rigid). For the sake of argument, assume for a moment that Cell was declared covariant instead—*i.e.*, it was declared class Cell[+T]—and that this passed the Scala compiler. (It doesn't, and we'll explain why shortly.) Then you could construct the following problematic statement sequence:

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

Seen by itself, each of these four lines looks OK. The first line creates a cell of strings and stores it in a val named c1. The second line defines a new val, c2, of type Cell[Any], which initialized with c1. This is OK since Cells are assumed to be covariant. The third line sets the value of cell c2 to 1. This is also OK because the assigned value 1 is an instance of c2's element type Any. Finally, the last line assigns the element value of c1 into a string. Nothing strange here, as both the sides are of the same type. But taken together, these four lines end up assigning the integer 1 to the string s. This is clearly a violation of type soundness.

Which operation is to blame for the runtime fault? It must be the second one, which uses covariant subtyping. The other statements are too simple and fundamental. Thus, a Cell ofString is not also a Cell of Any, because there are things you can do with a Cell of Any that you cannot do with a Cell of String. You cannot use set with an Int argument on a Cell of String, for example.

In fact, were you to pass the covariant version of Cell to the Scala compiler, you would get a compile-time error:

```
Cell.scala:7: error: covariant type T occurs in
contravariant position in type T of value x
   def set(x: T) = current = x
              ^
```

Variance and arrays

It's interesting to compare this behavior with arrays in Java. In principle, arrays are just like cells except that they can have more than one element. Nevertheless, arrays are treated as covariant in Java.

You can try an example analogous to the cell interaction described here with Java arrays:

```
// this is Java
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
```

If you try out this example, you will find that it compiles. But executing the program will cause an ArrayStore exception to be thrown when a2[0] is assigned to an Integer:

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
        at JavaArrays.main(JavaArrays.java:8)
```

What happens here is that Java stores the element type of the array at runtime. Then, every time an array element is updated, the new element value is checked against the stored type. If it is not an instance of that type, an ArrayStore exception is thrown.

You might ask why Java adopted this design, which seems both unsafe and expensive. When asked this question, James Gosling, the principal inventor of the Java language, answered that they wanted to have a simple means to treat arrays generically. For instance, they wanted to be able to write a method to sort all elements of an array, using a signature like the following that takes an array of Object:

```
void sort(Object[] a, Comparator cmp) { ... }
```

Covariance of arrays was needed so that arrays of arbitrary reference types could be passed to this sort method. Of course, with the arrival of Java generics, such a sort method can now be written with a type parameter, so the covariance of arrays is no longer necessary. For compatibility reasons, though, it has persisted in Java to this day.

Scala tries to be purer than Java in not treating arrays as covariant. Here's what you get if you translate the first two lines of the array example to Scala:

```
scala> val a1 = Array("abc")
a1: Array[String] = Array(abc)

scala> val a2: Array[Any] = a1
<console>:8: error: type mismatch;
 found   : Array[String]
 required: Array[Any]
       val a2: Array[Any] = a1
                            ^
```

What happened here is that Scala treats arrays as nonvariant (rigid), so an Array[String] is not considered to conform to an Array[Any]. However, sometimes it is necessary to interact with legacy methods in Java that use an Object array as a means to emulate a generic array. For instance, you might want to call a sort method like the one described previously with an array of Strings as argument. To make this possible, Scala lets you cast an array of Ts to an array of any supertype of T:

```
scala> val a2: Array[Object] =
         a1.asInstanceOf[Array[Object]]
a2: Array[Object] = Array(abc)
```

The cast is always legal at compile-time, and it will always succeed at run-time because the JVM's underlying run-time model treats arrays as covariant, just as Java the language does. But you might get ArrayStore exceptions afterwards, again just as you would in Java.

19.4 CHECKING VARIANCE ANNOTATIONS

Now that you have seen some examples where variance is unsound, you may be wondering which kind of class definitions need to be rejected and which can be accepted. So far, all violations of type soundness involved some reassignable field or array element. The purely functional implementation of queues, on the other hand, looks like a good candidate for covariance. However, the following example shows that you can "engineer" an unsound situation even if there is no reassignable field.

To set up the example, assume that queues as defined in Listing 19.4 are covariant. Then, create a subclass of queues that specializes the element type to Int and overrides the enqueuemethod:

```
class StrangeIntQueue extends Queue[Int] {
  override def enqueue(x: Int) = {
    println(math.sqrt(x))
    super.enqueue(x)
  }
}
```

The enqueue method in StrangeIntQueue prints out the square root of its (integer) argument before doing the append proper.

Now, you can write a counterexample in two lines:

```
val x: Queue[Any] = new StrangeIntQueue
x.enqueue("abc")
```

The first of these two lines is valid because StrangeIntQueue is a subclass of Queue[Int] and, assuming covariance of queues, Queue[Int] is a subtype of Queue[Any]. The second line is valid because you can append a String to a Queue[Any]. However, taken together, these two lines have the effect of applying a square root method to a string, which makes no sense.

Clearly it's not just mutable fields that make covariant types unsound. The problem is more general. It turns out that as soon as a generic parameter type appears as the type of a method parameter, the containing class or trait may not be covariant in that type parameter.

For queues, the enqueue method violates this condition:

```
class Queue[+T] {
  def enqueue(x: T) =
   ...
}
```

Running a modified queue class like the one above through a Scala compiler would yield:

```
Queues.scala:11: error: covariant type T occurs in
contravariant position in type T of value x
  def enqueue(x: T) =
             ^
```

Reassignable fields are a special case of the rule that disallows type parameters annotated with + from being used as method parameter types. As mentioned in Section 18.2, a reassignable field, "var x: T", is treated in Scala as a getter method, "def x: T", and a setter method, "def x_=(y: T)". As you can see, the setter method has a parameter of the field's type T. So that type may not be covariant.

THE FAST TRACK

In the rest of this section, we'll describe the mechanism by which the Scala compiler checks variance annotations. If you're not interested in such detail right now, you can safely skip toSection 19.5. The most important thing to understand is that the Scala compiler will check any variance annotations you place on type parameters. For example, if you try to declare a type parameter to be covariant (by adding a +), but that could lead to potential runtime errors, your program won't compile.

To verify correctness of variance annotations, the Scala compiler classifies all positions in a class or trait body as *positive*, *negative* or *neutral*. A "position" is any location in the class or trait (but from now on we'll just write "class") body where a type parameter may be used. For example, every method value parameter is a position because a method value parameter has a type. Therefore a type parameter could appear in that position.

The compiler checks each use of each of the class's type parameters. Type parameters annotated with + may only be used in positive positions, while type parameters annotated with - may only be used in negative positions. A type parameter with no variance annotation may be used in any position, and is, therefore, the only kind of type parameter that can be used in neutral positions of the class body.

To classify the positions, the compiler starts from the declaration of a type parameter and then moves inward through deeper nesting levels. Positions at the top level of the declaring class are classified as positive. By default, positions at deeper nesting levels are classified the same as that at enclosing levels, but there are a handful of exceptions where the classification changes. Method value parameter positions are classified to the *flipped* classification relative to positions outside the method, where the flip of a positive classification is negative, the flip of a negative classification is positive, and the flip of a neutral classification is still neutral.

Besides method value parameter positions, the current classification is also flipped at the type parameters of methods. A classification is sometimes flipped at the type argument position of a type, such as the Arg in C[Arg], depending on the variance of the corresponding type parameter. If C's type parameter is annotated with a + then the classification stays the same. If C's type parameter is annotated with a -, then the current classification is flipped. IfC's type parameter has no variance annotation then the current classification is changed to neutral.

As a somewhat contrived example, consider the following class definition, where several positions are annotated with their classifications, ^+ (for positive) or ^- (for negative):

```
abstract class Cat[-T, +U] {
  def meow[W^-](volume: T^-, listener: Cat[U^+, T^-]^-)
    : Cat[Cat[U^+, T^-]^-, U^+]^+
}
```

The positions of the type parameter, W, and the two value parameters, volume and listener, are all negative. Looking at the result type of meow, the position of the first Cat[U, T] argument is negative because Cat's first type parameter, T, is annotated with a -. The type U inside this argument is again in positive position (two flips), whereas the type T inside that argument is still in negative position.

You see from this discussion that it's quite hard to keep track of variance positions. That's why it's a welcome relief that the Scala compiler does this job for you.

Once the classifications are computed, the compiler checks that each type parameter is only used in positions that are classified appropriately. In this case, T is only used in negative positions, and U is only used in positive positions. So class Cat is type correct.

19.5 LOWER BOUNDS

Back to the Queue class. You saw that the previous definition of Queue[T] shown in Listing 19.4cannot be made covariant in T because T appears as a type of a parameter of the enqueuemethod, and that's a negative position.

Fortunately, there's a way to get unstuck: you can generalize enqueue by making it polymorphic (*i.e.*, giving the enqueue method itself a type parameter) and using a *lower bound* for its type parameter. Listing 19.6 shows a new formulation of Queue that implements this idea.

```
class Queue[+T] (private val leading: List[T],
    private val trailing: List[T] ) {
  def enqueue[U >: T](x: U) =
    new Queue[U](leading, x :: trailing) // ...
}
```

Listing 19.6 - A type parameter with a lower bound.

The new definition gives enqueue a type parameter U, and with the syntax, "U >: T", defines T as the lower bound for U. As a result, U is required to be a supertype of T.[1] The parameter toenqueue is now of type U instead of type T, and the return value of the method is now Queue[U]instead of Queue[T].

For example, suppose there is a class Fruit with two subclasses, Apple and Orange. With the new definition of class Queue, it is possible to append an Orange to a Queue[Apple]. The result will be aQueue[Fruit].

This revised definition of enqueue is type correct. Intuitively, if T is a more specific type than expected (for example, Apple instead of Fruit), a call to enqueue will still work because U (Fruit) will still be a supertype of T (Apple).[2]

The new definition of enqueue is arguably better than the old, because it is more general. Unlike the old version, the new definition allows you to append an arbitrary supertype U of the queue element type T. The result is then a Queue[U]. Together with queue covariance, this gives the right kind of flexibility for modeling queues of different element types in a natural way.

This shows that variance annotations and lower bounds play well together. They are a good example of type-driven design, where the types of an interface guide its detailed design and implementation. In the case of queues, it's likely you would not have thought of the refined implementation of enqueue with a lower bound. But you might have decided to make queues covariant, in which case, the compiler would have pointed out the variance error for enqueue. Correcting the variance error by adding a lower bound makes enqueue more general and queues as a whole more usable.

This observation is also the main reason that Scala prefers declaration-site variance over use-site variance as it is found in Java's wildcards. With use-site variance, you are on your own designing a class. It will be the clients of the class that need to put in the wildcards, and if they get it wrong, some important instance methods will no longer be applicable. Variance being a tricky business, users usually get it wrong, and they come away thinking that wildcards and generics are overly complicated. With definition-side variance, you express your intent to the compiler, and the compiler will double check that the methods you want available will indeed be available.

19.6 CONTRAVARIANCE

So far in this chapter, all examples you've seen were either covariant or nonvariant. But there are also cases where contravariance is natural. For instance, consider the trait of output channels shown in Listing 19.7:

```
trait OutputChannel[-T] {
  def write(x: T)
}
```

Listing 19.7 - A contravariant output channel.

Here, OutputChannel is defined to be contravariant in T. So an output channel of AnyRefs, say, is a subtype of an output channel of Strings. Although it may seem non-intuitive, it actually makes sense. To see why, consider what you can do with an OutputChannel[String]. The only supported operation is writing a String to it. The same operation can also be done on anOutputChannel[AnyRef]. So it is safe to substitute an OutputChannel[AnyRef] for anOutputChannel[String]. By contrast, it would not be safe to substitute an OutputChannel[String]where an OutputChannel[AnyRef] is required. After all, you can send any object to anOutputChannel[AnyRef], whereas an OutputChannel[String] requires that the written values are all strings.

This reasoning points to a general principle in type system design: It is safe to assume that a type T is a subtype of a type U if you can substitute a value of type T wherever a value of type Uis required. This is called the Liskov Substitution Principle. The principle holds if T supports the same operations as U, and all of T's operations require less and provide more than the corresponding operations in U. In the case of output channels, an OutputChannel[AnyRef] can be a subtype of an OutputChannel[String] because the two support the same write operation, and this operation requires less in OutputChannel[AnyRef] than in OutputChannel[String]. "Less" means the argument is only required to be an AnyRef in the first case, whereas it is required to be a Stringin the second case.

Sometimes covariance and contravariance are mixed in the same type. A prominent example is Scala's function traits. For instance, whenever you write the function type A => B, Scala expands this to Function1[A, B]. The definition of Function1 in the standard library uses both covariance and contravariance: the Function1 trait is contravariant in the function argument type S and covariant in the result type T, as shown in Listing 19.8. This satisfies the Liskov Substitution Principle because arguments are something that's required, whereas results are something that's provided.

```
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

Listing 19.8 - Covariance and contravariance of Function1s.

As an example, consider the application shown in Listing 19.9. Here, class Publicationcontains one parametric field, title, of type String. Class Book extends Publication and forwards its string title parameter to the constructor of its superclass. The Library singleton object defines a set of books and a method printBookList, which takes a function, named info, of typeBook => AnyRef. In other words, the type of the lone parameter to printBookList is a function that takes one Book argument and returns an AnyRef. The Customer application defines a method,getTitle, which takes a Publication as its lone parameter and returns a String, the title of the passed Publication.

```
class Publication(val title: String)
class Book(title: String) extends Publication(title)

object Library {
 val books: Set[Book] =
   Set(
     new Book("Programming in Scala"),
     new Book("Walden")
   )
 def printBookList(info: Book => AnyRef) = {
   for (book <- books) println(info(book))
 }
}

object Customer extends App {
 def getTitle(p: Publication): String = p.title
 Library.printBookList(getTitle)
}
```

Listing 19.9 - Demonstration of function type parameter variance.

Now take a look at the last line in Customer. This line invokes Library's printBookList method and passes getTitle, wrapped in a function value:

```
  Library.printBookList(getTitle)
```

This line of code type checks even though String, the function's result type, is a subtype ofAnyRef, the result type of printBookList's info parameter. This code passes the compiler because function result types are declared to be covariant (the +T in Listing 19.8). If you look inside the body of printBookList, you can get a glimpse of why this makes sense.

The printBookList method iterates through its book list and invokes the passed function on each book. It passes the AnyRef result returned by info to println, which invokes toString on it and prints the result. This activity will work with String as well as any other subclass of AnyRef, which is what covariance of function result types means.

Now consider the parameter type of the function being passed to the printBookList method. Although printBookList's parameter type is declared as Book, the getTitle we're passing in takes a Publication, a *supertype* of Book. The reason this works is that since printBookList's parameter type is Book, the body of the printBookList method will only be allowed to pass a Book into the function. And because getTitle's parameter type is Publication, the body of that function will only be able to access on its parameter, p, members that are declared in class Publication. Because any method declared in Publication is also available on its subclass Book, everything should work, which is what contravariance of function parameter types means. You can see all this graphically in Figure 19.1.

The code in Listing 19.9 compiles because Publication => String is a subtype of Book => AnyRef, as shown in the center of the Figure 19.1. Because the result type of a Function1 is defined as covariant, the inheritance relationship of the two result types, shown at the right of the diagram, is in the same direction as that of the two functions shown in the center. By contrast, because the parameter type of a Function1 is defined as contravariant, the inheritance relationship of the two parameter types, shown at the left of the diagram, is in the opposite direction as that of the two functions.

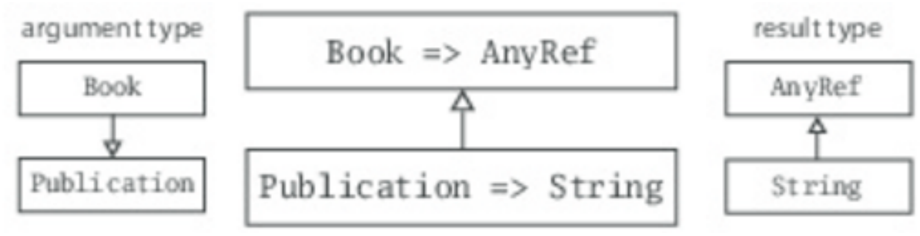

Figure 19.1 - Covariance and contravariance in function type parameters.

```
class Queue[+T] private (
  private[this] var leading: List[T],
  private[this] var trailing: List[T]
) {

  private def mirror() =
    if (leading.isEmpty) {
      while (!trailing.isEmpty) {
        leading = trailing.head :: leading
        trailing = trailing.tail
      }
    }

  def head: T = {
    mirror()
    leading.head
  }

  def tail: Queue[T] = {
    mirror()
    new Queue(leading.tail, trailing)
  }

  def enqueue[U >: T](x: U) =
    new Queue[U](leading, x :: trailing)
}
```

Listing 19.10 - An optimized functional queue.

19.7 OBJECT PRIVATE DATA

The Queue class seen so far has a problem in that the mirror operation will repeatedly copy thetrailing into the leading list if head is called several times in a row on a list where leading is empty. The wasteful copying could be avoided by adding some judicious side effects. Listing 19.10 presents a new implementation of Queue, which performs at most one trailing to leadingadjustment for any sequence of head operations.

What's different with respect to the previous version is that now leading and trailing are reassignable variables, and mirror performs the reverse copy from trailing to leading as a side effect on the current queue instead of returning a new queue. This side effect is purely internal to the implementation of the Queue operation; since leading and trailing are private variables, the effect is not visible to clients of Queue. So by the terminology established inChapter 18, the new version of Queue still defines purely functional objects, in spite of the fact that they now contain reassignable fields.

You might wonder whether this code passes the Scala type checker. After all, queues now contain two reassignable fields of the covariant parameter type T. Is this not a violation of the variance rules? It would be indeed, except for the detail that leading and trailing have aprivate[this] modifier, and are thus declared to be object private.

As mentioned in Section 13.5, object private members can be accessed only from within the object in which they are defined. It turns out that accesses to variables from the same object in which they are defined do not cause problems with variance. The intuitive explanation is that, in order to construct a case where variance would lead to type errors, you need to have a reference to a containing object that has a statically weaker type than the type the object was defined with. For accesses to object private values, however, this is impossible.

Scala's variance checking rules contain a special case for object private definitions. Such definitions are omitted when it is checked that a type parameter with either a + or -annotation occurs only in positions that have the same variance classification. Therefore, the code in Listing 19.10 compiles without error. On the other hand, if you had left out the [this]qualifiers from the two private modifiers, you would see two type errors:

```
Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of parameter of
setter leading_=
class Queue[+T] private (private var leading: List[T],
                                     ^
Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of parameter of
setter trailing_=
                         private var trailing: List[T]) {
                                     ^
```

19.8 UPPER BOUNDS

In Listing 16.1 here, we showed a merge sort function for lists that took a comparison function as a first argument and a list to sort as a second, curried argument. Another way you might want to organize such a sort function is by requiring the type of the list to mix in theOrdered trait. As mentioned in Section 12.4, by mixing Ordered into a class and implementingOrdered's one abstract method, compare, you enable clients to compare instances of that class with <, >, <=, and >=. For example, Listing 19.11 shows Ordered being mixed into a Person class.

As a result, you can compare two people like this:

```
scala> val robert = new Person("Robert", "Jones")
robert: Person = Robert Jones

scala> val sally = new Person("Sally", "Smith")
sally: Person = Sally Smith

scala> robert < sally
res0: Boolean = true
```

```
  class Person(val firstName: String, val lastName: String)
```

```
        extends Ordered[Person] {

      def compare(that: Person) = {
        val lastNameComparison =
          lastName.compareToIgnoreCase(that.lastName)
        if (lastNameComparison != 0)
          lastNameComparison
        else
          firstName.compareToIgnoreCase(that.firstName)
      }

      override def toString = firstName + " " + lastName
    }
```

Listing 19.11 - A Person class that mixes in the Ordered trait.

```
  def orderedMergeSort[T <: Ordered[T]](xs: List[T]): List[T] = {
    def merge(xs: List[T], ys: List[T]): List[T] =
      (xs, ys) match {
        case (Nil, _) => ys
        case (_, Nil) => xs
        case (x :: xs1, y :: ys1) =>
          if (x < y) x :: merge(xs1, ys)
          else y :: merge(xs, ys1)
      }
    val n = xs.length / 2
    if (n == 0) xs
    else {
      val (ys, zs) = xs splitAt n
      merge(orderedMergeSort(ys), orderedMergeSort(zs))
    }
  }
```

Listing 19.12 - A merge sort function with an upper bound.

To require that the type of the list passed to your new sort function mixes in Ordered, you need to use an *upper bound*. An upper bound is specified similar to a lower bound, except instead of the >: symbol used for lower bounds, you use a <: symbol, as shown in Listing 19.12.

With the "T <: Ordered[T]" syntax, you indicate that the type parameter, T, has an upper bound,Ordered[T]. This means that the element type of the list passed to orderedMergeSort must be a subtype of Ordered. Thus, you could pass a List[Person] to orderedMergeSort because Person mixes in Ordered.

For example, consider this list:

```
  scala> val people = List(
           new Person("Larry", "Wall"),
           new Person("Anders", "Hejlsberg"),
           new Person("Guido", "van Rossum"),
           new Person("Alan", "Kay"),
           new Person("Yukihiro", "Matsumoto")
         )
  people: List[Person] = List(Larry Wall, Anders Hejlsberg,
    Guido van Rossum, Alan Kay, Yukihiro Matsumoto)
```

Because the element type of this list, Person, mixes in (and is therefore a subtype of)Ordered[People], you can pass the list to orderedMergeSort:

```
scala> val sortedPeople = orderedMergeSort(people)
sortedPeople: List[Person] = List(Anders Hejlsberg, Alan Kay,
  Yukihiro Matsumoto, Guido van Rossum, Larry Wall)
```

Now, although the sort function shown in Listing 19.12 serves as a useful illustration of upper bounds, it isn't actually the most general way in Scala to design a sort function that takes advantage of the Ordered trait.

For example, you couldn't use the orderedMergeSort function to sort a list of integers, because class Int is not a subtype of Ordered[Int]:

```
scala> val wontCompile = orderedMergeSort(List(3, 2, 1))
<console>:5: error: inferred type arguments [Int] do
   not conform to method orderedMergeSort's type
     parameter bounds [T <: Ordered[T]]
       val wontCompile = orderedMergeSort(List(3, 2, 1))
                         ^
```

In Section 21.6, we'll show you how to use *implicit parameters* and *context bounds* to achieve a more general solution.

19.9 CONCLUSION

In this chapter you saw several techniques for information hiding: private constructors, factory methods, type abstraction, and object private members. You also learned how to specify data type variance and what it implies for class implementation. Finally, you saw two techniques which help in obtaining flexible variance annotations: lower bounds for method type parameters and private[this] annotations for local fields and methods.

Footnotes for Chapter 19:

[1] Supertype and subtype relationships are reflexive, which means a type is both a supertype and a subtype of itself. Even though T is a lower bound for U, you could still pass in a T toenqueue.

[2] Technically, what happens is a flip occurs for lower bounds. The type parameter U is in a negative position (1 flip), while the lower bound (>: T) is in a positive position (2 flips).