Chapter 10

Composition and Inheritance

Chapter 6 introduced some basic object-oriented aspects of Scala. This chapter picks up where Chapter 6 left off and dives into Scala's support for object-oriented programming in much greater detail.

We'll compare two fundamental relationships between classes: composition and inheritance. Composition means one class holds a reference to another, using the referenced class to help it fulfill its mission. Inheritance is the superclass/subclass relationship.

In addition to these topics, we'll discuss abstract classes, parameterless methods, extending classes, overriding methods and fields, parametric fields, invoking superclass constructors, polymorphism and dynamic binding, final members and classes, and factory objects and methods.

10.1 A TWO-DIMENSIONAL LAYOUT LIBRARY

As a running example in this chapter, we'll create a library for building and rendering two-dimensional layout elements. Each element will represent a rectangle filled with text. For convenience, the library will provide factory methods named "elem" that construct new elements from passed data. For example, you'll be able to create a layout element containing a string using a factory method with the following signature:

```
elem(s: String): Element
```

As you can see, elements will be modeled with a type named Element. You'll be able to call aboveor beside on an element, passing in a second element, to get a new element that combines the two. For example, the following expression would construct a larger element consisting of two columns, each with a height of two:

```
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2
```

Printing the result of this expression would give you:

```
hello ***
 *** world
```

Layout elements are a good example of a system in which objects can be constructed from simple parts with the aid of composing operators. In this chapter, we'll define classes that enable element objects to be constructed from arrays, lines, and rectangles. These basic element objects will be the simple parts. We'll also define composing operators above andbeside. Such composing operators are also often called combinators because they combine elements of some domain into new elements.

Thinking in terms of combinators is generally a good way to approach library design: it pays to think about the fundamental ways to construct objects in an application domain. What are the simple objects?

In what ways can more interesting objects be constructed out of simpler ones? How do combinators hang together? What are the most general combinations? Do they satisfy any interesting laws? If you have good answers to these questions, your library design is on track.

10.2 ABSTRACT CLASSES

Our first task is to define type Element, which represents layout elements. Since elements are two dimensional rectangles of characters, it makes sense to include a member, contents, that refers to the contents of a layout element. The contents can be represented as an array of strings, where each string represents a line. Hence, the type of the result returned by contentswill be Array[String]. Listing 10.1 shows what it will look like.

```
abstract class Element {
  def contents: Array[String]
}
```

Listing 10.1 - Defining an abstract method and class.

In this class, contents is declared as a method that has no implementation. In other words, the method is an abstract member of class Element. A class with abstract members must itself be declared abstract, which is done by writing an abstract modifier in front of the class keyword:

```
abstract class Element ...
```

The abstract modifier signifies that the class may have abstract members that do not have an implementation. As a result, you cannot instantiate an abstract class. If you try to do so, you'll get a compiler error:

```
scala> new Element
<console>:5: error: class Element is abstract;
    cannot be instantiated
       new Element
           ^
```

Later in this chapter, you'll see how to create subclasses of class Element, which you'll be able to instantiate because they fill in the missing definition for contents.

Note that the contents method in class Element does not carry an abstract modifier. A method is abstract if it does not have an implementation (*i.e.*, no equals sign or body). Unlike Java, no abstract modifier is necessary (or allowed) on method declarations. Methods that have an implementation are called concrete.

Another bit of terminology distinguishes between declarations and definitions.
Class Element*declares* the abstract method contents, but currently *defines* no concrete methods. In the next section, however, we'll enhance Element by defining some concrete methods.

10.3 DEFINING PARAMETERLESS METHODS

As a next step, we'll add methods to Element that reveal its width and height, as shown inListing 10.2. The height method returns the number of lines in contents. The width method returns the length of the first line, or if there are no lines in the element, returns zero. (This means you cannot define an element with a height of zero and a non-zero width.)

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
}
```

Listing 10.2 - Defining parameterless methods width and height.

Note that none of Element's three methods has a parameter list, not even an empty one. For example, instead of:

```
def width(): Int
```

the method is defined without parentheses:

```
def width: Int
```

Such *parameterless methods* are quite common in Scala. By contrast, methods defined with empty parentheses, such as def height(): Int, are called *empty-paren methods*. The recommended convention is to use a parameterless method whenever there are no parameters *and* the method accesses mutable state only by reading fields of the containing object (in particular, it does not change mutable state). This convention supports the uniform access principle,[1] which says that client code should not be affected by a decision to implement an attribute as a field or method.

For instance, we could implement width and height as fields, instead of methods, simply by changing the def in each definition to a val:

```
abstract class Element {
  def contents: Array[String]
  val height = contents.length
  val width =
    if (height == 0) 0 else contents(0).length
}
```

The two pairs of definitions are completely equivalent from a client's point of view. The only difference is that field accesses might be slightly faster than method invocations because the field values are pre-computed when the class is initialized, instead of being computed on each method call. On the other hand, the fields require extra memory space in each Element object. So it depends on the usage profile of a class whether an attribute is better represented as a field or method, and that usage profile might change over time. The point is that clients of theElement class should not be affected when its internal implementation changes.

In particular, a client of class Element should not need to be rewritten if a field of that class gets changed into an access function, so long as the access function is pure (*i.e.*, it does not have any side effects and does not depend on mutable state). The client should not need to care either way.

So far so good. But there's still a slight complication that has to do with the way Java handles things. The problem is that Java does not implement the uniform access principle. So it'sstring.length() in Java, not string.length, even though it's array.length, not array.length(). Needless to say, this is very confusing.

To bridge that gap, Scala is very liberal when it comes to mixing parameterless and empty-paren methods. In particular, you can override a parameterless method with an empty-paren method, and *vice versa*. You can also leave off the empty parentheses on an invocation of any function that takes no arguments. For instance, the following two lines are both legal in Scala:

```
Array(1, 2, 3).toString
"abc".length
```

In principle it's possible to leave out all empty parentheses in Scala function calls. However, it's still recommended to write the empty parentheses when the invoked method represents more than a property of its receiver object. For instance, empty parentheses are appropriate if the method performs I/O, writes reassignable variables (vars), or reads vars other than the receiver's fields, either directly or indirectly by using mutable objects. That way, the parameter list acts as a visual clue that some interesting computation is triggered by the call. For instance:

```
"hello".length  // no () because no side-effect
println()       // better to not drop the ()
```

To summarize, it is encouraged in Scala to define methods that take no parameters and have no side effects as parameterless methods (*i.e.*, leaving off the empty parentheses). On the other hand, you should never define a method that has side-effects without parentheses, because invocations of that method would then look like a field selection. So your clients might be surprised to see the side effects.

Similarly, whenever you invoke a function that has side effects, be sure to include the empty parentheses when you write the invocation. Another way to think about this is if the function you're calling performs an operation, use the parentheses. But if it merely provides access to a property, leave the parentheses off.

10.4 EXTENDING CLASSES

We still need to be able to create new element objects. You have already seen that "new Element" cannot be used for this because class Element is abstract. To instantiate an element, therefore, we will need to create a subclass that extends Element and implements the abstract contentsmethod. Listing 10.3 shows one possible way to do that:

```
class ArrayElement(conts: Array[String]) extends Element {
  def contents: Array[String] = conts
}
```

Listing 10.3 - Defining ArrayElement as a subclass of Element.

Class ArrayElement is defined to extend class Element. Just like in Java, you use an extends clause after the class name to express this:

```
... extends Element ...
```

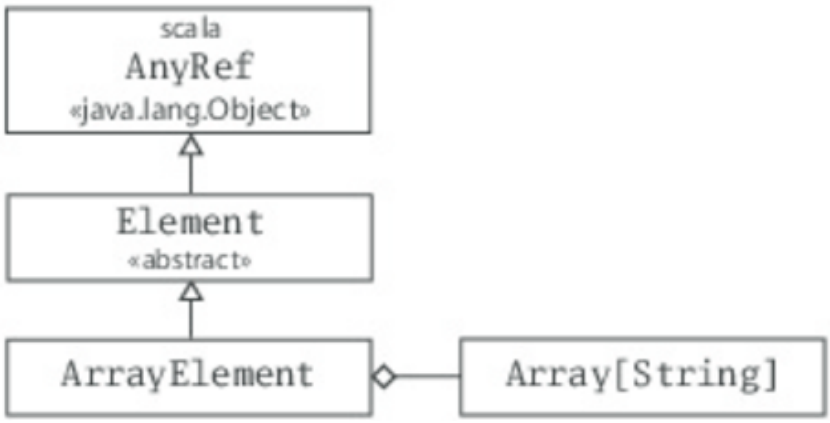

Figure 10.1 - Class diagram for ArrayElement.

Such an extends clause has two effects: It makes class ArrayElement inherit all non-private members from class Element, and it makes the type ArrayElement a subtype of the type Element. Given ArrayElement extends Element, class ArrayElement is called a subclass of class Element.Conversely, Element is a *superclass* of ArrayElement. If you leave out an extends clause, the Scala compiler implicitly assumes your class extends from scala.AnyRef, which on the Java platform is the same as class java.lang.Object. Thus, class Element implicitly extends class AnyRef. You can see these inheritance relationships in Figure 10.1.

Inheritance means that all members of the superclass are also members of the subclass, with two exceptions. First, private members of the superclass are not inherited in a subclass. Second, a member of a superclass is not inherited if a member with the same name and parameters is already implemented in the subclass. In that case we say the member of the subclass overrides the member of the superclass. If the member in the subclass is concrete and the member of the superclass is abstract, we also say that the concrete memberimplements the abstract one.

For example, the contents method in ArrayElement overrides (or alternatively: implements) abstract method contents in class Element.[2] By contrast, class ArrayElement inherits the widthand height methods from class Element. For example, given an ArrayElement ae, you can query its width using ae.width, as if width were defined in class ArrayElement:

```
scala> val ae = new ArrayElement(Array("hello", "world"))
ae: ArrayElement = ArrayElement@39274bf7

scala> ae.width
res0: Int = 5
```

Subtyping means that a value of the subclass can be used wherever a value of the superclass is required. For example:

```
val e: Element = new ArrayElement(Array("hello"))
```

Variable e is defined to be of type Element, so its initializing value should also be an Element. In fact, the initializing value's type is ArrayElement. This is OK, because class ArrayElement extends class Element, and as a result, the type ArrayElement is compatible with the type Element.[3]

Figure 10.1 also shows the *composition* relationship that exists between ArrayElement andArray[String]. This relationship is called composition because class ArrayElement is "composed" out of class Array[String], in that the Scala compiler will place into the binary class it generates for ArrayElement a field that holds a reference to the passed conts array. We'll discuss some design considerations concerning composition and inheritance later in this chapter, inSection 10.11.

10.5 OVERRIDING METHODS AND FIELDS

The uniform access principle is just one aspect where Scala treats fields and methods more uniformly than Java. Another difference is that in Scala, fields and methods belong to the same namespace. This makes it possible for a field to override a parameterless method. For instance, you could change the implementation of contents in class ArrayElement from a method to a field without having to modify the abstract method definition of contents in class Element, as shown in Listing 10.4:

```
class ArrayElement(conts: Array[String]) extends Element {
  val contents: Array[String] = conts
}
```

Listing 10.4 - Overriding a parameterless method with a field.

Field contents (defined with a val) in this version of ArrayElement is a perfectly good implementation of the parameterless method contents (declared with a def) in class Element. On the other hand, in Scala it is forbidden to define a field and method with the same name in the same class, whereas this is allowed in Java.

For example, this Java class would compile just fine:

```
// This is Java
class CompilesFine {
  private int f = 0;
  public int f() {
    return 1;
  }
}
```

But the corresponding Scala class would not compile:

```
class WontCompile {
  private var f = 0 // Won't compile, because a field
  def f = 1         // and method have the same name
}
```

Generally, Scala has just two namespaces for definitions in place of Java's four. Java's four namespaces are fields, methods, types, and packages. By contrast, Scala's two namespaces are:

- values (fields, methods, packages, and singleton objects)
- types (class and trait names)

The reason Scala places fields and methods into the same namespace is precisely so you can override a parameterless method with a val, something you can't do with Java.[4]

10.6 DEFINING PARAMETRIC FIELDS

Consider again the definition of class ArrayElement shown in the previous section. It has aparameter conts whose sole purpose is to be copied into the contents field. The name conts of the parameter was chosen just so that it would look similar to the field name contents without actually clashing with it. This is a "code smell," a sign that there may be some unnecessary redundancy and repetition in your code.

You can avoid the code smell by combining the parameter and the field in a single*parametric field* definition, as shown in Listing 10.5:

```
class ArrayElement(
  val contents: Array[String]
) extends Element
```

Listing 10.5 - Defining contents as a parametric field.

Note that now the contents parameter is prefixed by val. This is a shorthand that defines at the same time a parameter and field with the same name. Specifically, class ArrayElement now has an (unreassignable) field contents, which can be accessed from outside the class. The field is initialized with the value of the parameter. It's as if the class had been written as follows, where x123 is an arbitrary fresh name for the parameter:

```
class ArrayElement(x123: Array[String]) extends Element {
  val contents: Array[String] = x123
}
```

You can also prefix a class parameter with var, in which case the corresponding field would be reassignable. Finally, it is possible to add modifiers, such as private, protected,[5] or override to these parametric fields, just as you can for any other class member. Consider, for instance, the following class definitions:

```
class Cat {
  val dangerous = false
}
class Tiger(
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

Tiger's definition is a shorthand for the following alternate class definition with an overriding member dangerous and a private member age:

```
class Tiger(param1: Boolean, param2: Int) extends Cat {
  override val dangerous = param1
  private var age = param2
}
```

Both members are initialized from the corresponding parameters. We chose the names of those parameters, param1 and param2, arbitrarily. The important thing was that they not clash with any other name in scope.

10.7 INVOKING SUPERCLASS CONSTRUCTORS

You now have a complete system consisting of two classes: an abstract class Element, which is extended by a concrete class ArrayElement. You might also envision other ways to express an element. For example, clients might want to create a layout element consisting of a single line given by a string. Object-oriented programming makes it easy to extend a system with new data-variants. You can simply add subclasses. For example, Listing 10.6 shows a LineElementclass that extends ArrayElement:

```
class LineElement(s: String) extends ArrayElement(Array(s)) {
  override def width = s.length
  override def height = 1
}
```

Listing 10.6 - Invoking a superclass constructor.

Since LineElement extends ArrayElement, and ArrayElement's constructor takes a parameter (anArray[String]), LineElement needs to pass an argument to the primary constructor of its superclass. To invoke a superclass constructor, you simply place the argument or arguments you want to pass in parentheses following the name of the superclass. For example, classLineElement passes Array(s) to ArrayElement's primary constructor by placing it in parentheses after the superclass ArrayElement's name:

```
... extends ArrayElement(Array(s)) ...
```

With the new subclass, the inheritance hierarchy for layout elements now looks as shown inFigure 10.2.

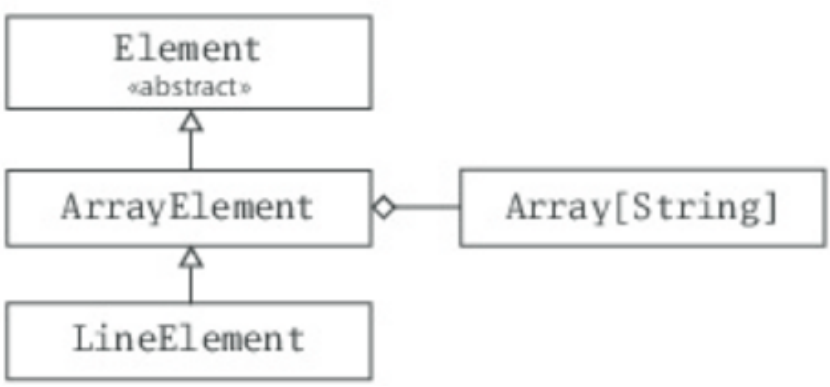

Figure 10.2 - Class diagram for LineElement.

10.8 USING OVERRIDE MODIFIERS

Note that the definitions of width and height in LineElement carry an override modifier. In Section 6.3, you saw this modifier in the definition of a toString method. Scala requires such a modifier for all members that override a concrete member in a parent class. The modifier is optional if a member implements an abstract member with the same name. The modifier is forbidden if a member does not override or implement some other member in a base class. Since height and width in class LineElement override concrete definitions in class Element, overrideis required.

This rule provides useful information for the compiler that helps avoid some hard-to-catch errors and makes system evolution safer. For instance, if you happen to misspell the method or accidentally give it a different parameter list, the compiler will respond with an error message:

```
$ scalac LineElement.scala
.../LineElement.scala:50:
error: method hight overrides nothing
  override def hight = 1
               ^
```

The override convention is even more important when it comes to system evolution. Say you defined a library of 2D drawing methods. You made it publicly available, and it is widely used. In the next version of the library you want to add to your base class Shape a new method with this signature:

```
def hidden(): Boolean
```

Your new method will be used by various drawing methods to determine whether a shape needs to be drawn. This could lead to a significant speedup, but you cannot do this without the risk of breaking client code. After all, a client could have defined a subclass of Shape with a different implementation of hidden. Perhaps the client's method actually makes the receiver object disappear instead of testing whether the object is hidden. Because the two versions ofhidden override each other, your drawing methods would end up making objects disappear, which is certainly not what you want!

These "accidental overrides" are the most common manifestation of what is called the "fragile base class" problem. The problem is that if you add new members to base classes (which we usually call

superclasses) in a class hierarchy, you risk breaking client code. Scala cannot completely solve the fragile base class problem, but it improves on the situation compared to Java.[6] If the drawing library and its clients were written in Scala, then the client's original implementation of hidden could not have had an override modifier, because at the time there was no other method with that name.

Once you add the hidden method to the second version of your shape class, a recompile of the client would give an error like the following:

```
.../Shapes.scala:6: error: error overriding method
    hidden in class Shape of type ()Boolean;
method hidden needs `override' modifier
def hidden(): Boolean =
^
```

That is, instead of wrong behavior your client would get a compile-time error, which is usually much preferable.

10.9 POLYMORPHISM AND DYNAMIC BINDING

You saw in Section 10.4 that a variable of type Element could refer to an object of typeArrayElement. The name for this phenomenon is *polymorphism*, which means "many shapes" or "many forms." In this case, Element objects can have many forms.[7]

So far, you've seen two such forms: ArrayElement and LineElement. You can create more forms ofElement by defining new Element subclasses. For example, you could define a new form ofElement that has a given width and height, and is filled everywhere with a given character:

```
class UniformElement(
  ch: Char,
  override val width: Int,
  override val height: Int
) extends Element {
  private val line = ch.toString * width
  def contents = Array.fill(height)(line)
}
```

The inheritance hierarchy for class Element now looks as shown in Figure 10.3. As a result, Scala will accept all of the following assignments, because the type of the assigning expression conforms to the type of the defined variable:

```
val e1: Element = new ArrayElement(Array("hello", "world"))
val ae: ArrayElement = new LineElement("hello")
val e2: Element = ae
val e3: Element = new UniformElement('x', 2, 3)
```

If you check the inheritance hierarchy, you'll find that in each of these four val definitions, the type of the expression to the right of the equals sign is below the type of the val being initialized to the left of the equals sign.

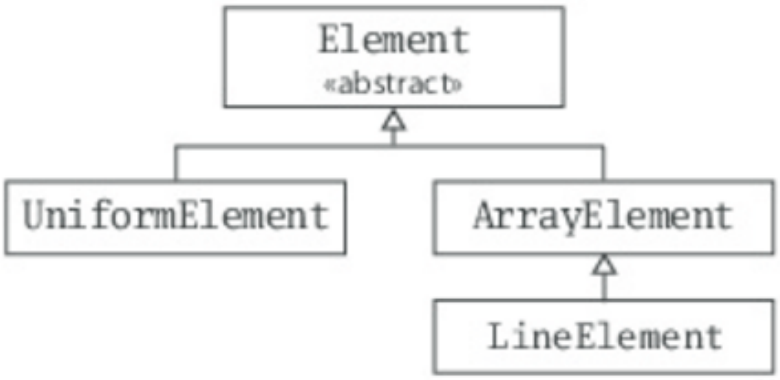

Figure 10.3 - Class hierarchy of layout elements.

The other half of the story, however, is that method invocations on variables and expressions are *dynamically bound*. This means that the actual method implementation invoked is determined at run time based on the class of the object, not the type of the variable or expression. To demonstrate this behavior, we'll temporarily remove all existing members from our Element classes and add a method named demo to Element. We'll override demo inArrayElement and LineElement, but not in UniformElement:

```
abstract class Element {
  def demo() = {
    println("Element's implementation invoked")
  }
}

class ArrayElement extends Element {
  override def demo() = {
    println("ArrayElement's implementation invoked")
  }
}

class LineElement extends ArrayElement {
  override def demo() = {
    println("LineElement's implementation invoked")
  }
}

// UniformElement inherits Element's demo
class UniformElement extends Element
```

If you enter this code into the interpreter, you can then define this method that takes anElement and invokes demo on it:

```
def invokeDemo(e: Element) = {
  e.demo()
}
```

If you pass an ArrayElement to invokeDemo, you'll see a message indicating ArrayElement's implementation of demo was invoked, even though the type of the variable, e, on which demowas invoked is Element:

```
scala> invokeDemo(new ArrayElement)
ArrayElement's implementation invoked
```

Similarly, if you pass a LineElement to invokeDemo, you'll see a message that indicates LineElement'sdemo implementation was invoked:

```
scala> invokeDemo(new LineElement)
LineElement's implementation invoked
```

The behavior when passing a UniformElement may at first glance look suspicious, but it is correct:

```
scala> invokeDemo(new UniformElement)
Element's implementation invoked
```

Because UniformElement does not override demo, it inherits the implementation of demo from its superclass, Element. Thus, Element's implementation is the correct implementation of demo to invoke when the class of the object is UniformElement.

10.10 DECLARING FINAL MEMBERS

Sometimes when designing an inheritance hierarchy, you want to ensure that a member cannot be overridden by subclasses. In Scala, as in Java, you do this by adding a finalmodifier to the member. As shown in Listing 10.7, you could place a final modifier onArrayElement's demo method.

```
class ArrayElement extends Element {
  final override def demo() = {
    println("ArrayElement's implementation invoked")
  }
}
```

Listing 10.7 - Declaring a final method.

Given this version of ArrayElement, an attempt to override demo in its subclass, LineElement, would not compile:

```
elem.scala:18: error: error overriding method demo
   in class ArrayElement of type ()Unit;
method demo cannot override final member
    override def demo() = {
                 ^
```

You may also at times want to ensure that an entire class not be subclassed. To do this you simply declare the entire class final by adding a final modifier to the class declaration. For example, Listing 10.8 shows how you would declare ArrayElement final:

```
final class ArrayElement extends Element {
  override def demo() = {
    println("ArrayElement's implementation invoked")
  }
}
```

Listing 10.8 - Declaring a final class.

With this version of ArrayElement, any attempt at defining a subclass would fail to compile:

```
elem.scala: 18: error: illegal inheritance from final class
    ArrayElement
  class LineElement extends ArrayElement {
                            ^
```

We'll now remove the final modifiers and demo methods, and go back to the earlier implementation of the Element family. We'll focus our attention in the remainder of this chapter to completing a working version of the layout library.

10.11 USING COMPOSITION AND INHERITANCE

Composition and inheritance are two ways to define a new class in terms of another existing class. If what you're after is primarily code reuse, you should in general prefer composition to inheritance. Only inheritance suffers from the fragile base class problem, in which you can inadvertently break subclasses by changing a superclass.

One question you can ask yourself about an inheritance relationship is whether it models an *is-a* relationship.[8] For example, it would be reasonable to say that ArrayElement *is-an* Element. Another question you can ask is whether clients will want to use the subclass type as a superclass type.[9] In the case of ArrayElement, we do indeed expect clients will want to use anArrayElement as an Element.

If you ask these questions about the inheritance relationships shown in Figure 10.3, do any of the relationships seem suspicious? In particular, does it seem obvious to you that a LineElement *is-an* ArrayElement? Do you think clients would ever need to use a LineElement as an ArrayElement?

In fact, we defined LineElement as a subclass of ArrayElement primarily to reuse ArrayElement's definition of contents. Perhaps it would be better, therefore, to define LineElement as a direct subclass of Element, like this:

```
class LineElement(s: String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

In the previous version, LineElement had an inheritance relationship with ArrayElement, from which it inherited contents. It now has a composition relationship with Array: it holds a reference to an array of strings from its own contents field.[10] Given this implementation ofLineElement, the inheritance hierarchy for Element now looks as shown in Figure 10.4.

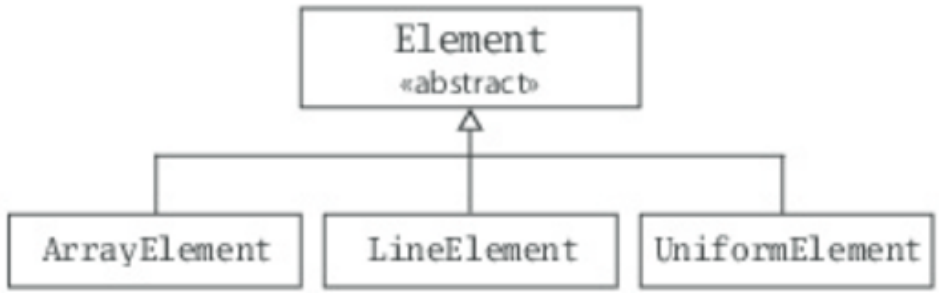

Figure 10.4 - Class hierarchy with revised LineElement.

10.12 IMPLEMENTING ABOVE, BESIDE, AND TOSTRING

As a next step, we'll implement method above in class Element. Putting one element above another means concatenating the two contents values of the elements. So a first draft of method above could look like this:

```
def above(that: Element): Element =
  new ArrayElement(this.contents ++ that.contents)
```

The ++ operation concatenates two arrays. Arrays in Scala are represented as Java arrays, but support many more methods. Specifically, arrays in Scala can be converted to instances of a class scala.Seq, which represents sequence-like structures and contains a number of methods for accessing and transforming sequences. Some other array methods will be explained in this chapter and a more comprehensive discussion will be given in Chapter 17.

In fact, the code shown previously is not quite sufficient because it does not let you put elements of different widths on top of each other. To keep things simple in this section, however, we'll leave this as is and only pass elements of the same length to above. In Section 10.14, we'll make an enhancement to above so that clients can use it to combine elements of different widths.

The next method to implement is beside. To put two elements beside each other, we'll create a new element in which every line results from concatenating corresponding lines of the two elements. As before, to keep things simple, we'll start by assuming the two elements have the same height. This leads to the following design of method beside:

```
def beside(that: Element): Element = {
  val contents = new Array[String](this.contents.length)
  for (i <- 0 until this.contents.length)
    contents(i) = this.contents(i) + that.contents(i)
  new ArrayElement(contents)
}
```

The beside method first allocates a new array, contents, and fills it with the concatenation of the corresponding array elements in this.contents and that.contents. It finally produces a newArrayElement containing the new contents.

Although this implementation of beside works, it is in an imperative style, the telltale sign of which is the loop in which we index through arrays. Alternatively, the method could be abbreviated to one expression:

```
new ArrayElement(
  for (
    (line1, line2) <- this.contents zip that.contents
  ) yield line1 + line2
)
```

Here, the two arrays, this.contents and that.contents, are transformed into an array of pairs (asTuple2s are called) using the zip operator. The zip operator picks corresponding elements in its two operands and forms an array of pairs. For instance, this expression:

```
Array(1, 2, 3) zip Array("a", "b")
```

will evaluate to:

```
Array((1, "a"), (2, "b"))
```

If one of the two operand arrays is longer than the other, zip will drop the remaining elements. In the expression above, the third element of the left operand, 3, does not form part of the result, because it does not have a corresponding element in the right operand.

The zipped array is then iterated over by a for expression. Here, the syntax "for ((line1, line2) <- ...)" allows you to name both elements of a pair in one pattern (*i.e.*, line1stands now for the first element of the pair, and line2 stands for the second). Scala's pattern-matching system will be described in detail in Chapter 15. For now, you can just think of this as a way to define two vals, line1 and line2, for each step of the iteration.

The for expression has a yield part and therefore yields a result. The result is of the same kind as the expression iterated over (*i.e.*, it is an array). Each element of the array is the result of concatenating the corresponding lines, line1 and line2. So the end result of this code is the same as in the first version of beside, but because it avoids explicit array indexing, the result is obtained in a less error-prone way.

You still need a way to display elements. As usual, this is done by defining a toString method that returns an element formatted as a string. Here is its definition:

```
override def toString = contents mkString "\n"
```

The implementation of toString makes use of mkString, which is defined for all sequences, including arrays. As you saw in Section 7.8, an expression like "arr mkString sep" returns a string consisting of all elements of the array arr. Each element is mapped to a string by calling its toString method. A separator string sep is inserted between consecutive element strings. So the expression, "contents mkString "\n"" formats the contents array as a string, where each array element appears on a line by itself.

```
  abstract class Element {
```

```
  def contents: Array[String]

  def width: Int =
    if (height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)

  def beside(that: Element): Element =
    new ArrayElement(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

Listing 10.9 - Class Element with above, beside, and toString.

Note that toString does not carry an empty parameter list. This follows the recommendations for the uniform access principle, because toString is a pure method that does not take any parameters. With the addition of these three methods, class Element now looks as shown inListing 10.9.

10.13 DEFINING A FACTORY OBJECT

You now have a hierarchy of classes for layout elements. This hierarchy could be presented to your clients "as is," but you might also choose to hide the hierarchy behind a factory object.

A factory object contains methods that construct other objects. Clients would then use these factory methods to construct objects, rather than constructing the objects directly with new. An advantage of this approach is that object creation can be centralized and the details of how objects are represented with classes can be hidden. This hiding will both make your library simpler for clients to understand, because less detail is exposed, and provide you with more opportunities to change your library's implementation later without breaking client code.

The first task in constructing a factory for layout elements is to choose where the factory methods should be located. Should they be members of a singleton object or of a class? What should the containing object or class be called? There are many possibilities. A straightforward solution is to create a companion object of class Element and make this the factory object for layout elements. That way, you need to expose only the class/object combo of Element to your clients, and you can hide the three implementation classes ArrayElement,LineElement, and UniformElement.

Listing 10.10 is a design of the Element object that follows this scheme. The Element object contains three overloaded variants of an elem method and each constructs a different kind of layout object.

```
object Element {

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)
```

```
  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

Listing 10.10 - A factory object with factory methods.

With the advent of these factory methods, it makes sense to change the implementation of class Element so that it goes through the elem factory methods rather than creating newArrayElement instances explicitly. To call the factory methods without qualifying them withElement, the name of the singleton object, we will import Element.elem at the top of the source file. In other words, instead of invoking the factory methods with Element.elem inside classElement, we'll import Element.elem so we can just call the factory methods by their simple name,elem. Listing 10.11 shows what class Element will look like after these changes.

```
import Element.elem

abstract class Element {

  def contents: Array[String]

  def width: Int =
    if (height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    elem(this.contents ++ that.contents)

  def beside(that: Element): Element =
    elem(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

Listing 10.11 - Class Element refactored to use factory methods.

In addition, given the factory methods, the subclasses, ArrayElement, LineElement, andUniformElement, could now be private because they no longer need to be accessed directly by clients. In Scala, you can define classes and singleton objects inside other classes and singleton objects. One way to make the Element subclasses private is to place them inside theElement singleton object and declare them private there. The classes will still be accessible to the three elem factory methods, where they are needed. Listing 10.12 shows how that will look.

10.14 HEIGHTEN AND WIDEN

We need one last enhancement. The version of Element shown in Listing 10.11 is not quite sufficient because it does not allow clients to place elements of different widths on top of each other, or place elements of different heights beside each other.

For example, evaluating the following expression won't work correctly, because the second line in the combined element is longer than the first:

```
new ArrayElement(Array("hello")) above
new ArrayElement(Array("world!"))
```

Similarly, evaluating the following expression would not work properly, because the firstArrayElement has a height of two and the second a height of only one:

```
new ArrayElement(Array("one", "two")) beside
new ArrayElement(Array("one"))
```

Listing 10.13 shows a private helper method, widen, which takes a width and returns an Elementof that width. The result contains the contents of this Element, centered, padded to the left and right by any spaces needed to achieve the required width. Listing 10.13 also shows a similar method, heighten, which performs the same function in the vertical direction. The widenmethod is invoked by above to ensure that Elements placed above each other have the same width. Similarly, the heighten method is invoked by beside to ensure that elements placed beside each other have the same height. With these changes, the layout library is ready for use.

```
object Element {

  private class ArrayElement(
    val contents: Array[String]
  ) extends Element

  private class LineElement(s: String) extends Element {
    val contents = Array(s)
    override def width = s.length
    override def height = 1
  }

  private class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
  ) extends Element {
    private val line = ch.toString * width
    def contents = Array.fill(height)(line)
  }

  def elem(contents:  Array[String]): Element =
    new ArrayElement(contents)

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
```

```
      new LineElement(line)
  }
```

Listing 10.12 - Hiding implementation with private classes.

```
import Element.elem

abstract class Element {
  def contents:  Array[String]

  def width: Int = contents(0).length
  def height: Int = contents.length

  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }

  def beside(that: Element): Element = {
    val this1 = this heighten that.height
    val that1 = that heighten this.height
    elem(
      for ((line1, line2) <- this1.contents zip that1.contents)
      yield line1 + line2)
  }

  def widen(w: Int): Element =
    if (w <= width) this
    else {
      val left = elem(' ', (w - width) / 2, height)
      val right = elem(' ', w - width - left.width, height)
      left beside this beside right
    }

  def heighten(h: Int): Element =
    if (h <= height) this
    else {
      val top = elem(' ', width, (h - height) / 2)
      val bot = elem(' ', width, h - height - top.height)
      top above this above bot
    }

  override def toString = contents mkString "\n"
}
```

Listing 10.13 - Element with widen and heighten methods.

10.15 PUTTING IT ALL TOGETHER

A fun way to exercise almost all elements of the layout library is to write a program that draws a spiral with a given number of edges. This Spiral program, shown in Listing 10.14, will do just that.

```
import Element.elem

object Spiral {

  val space = elem(" ")
```

```
  val corner = elem("+")

  def spiral(nEdges: Int, direction: Int): Element = {
    if (nEdges == 1)
      elem("+")
    else {
      val sp = spiral(nEdges - 1, (direction + 3) % 4)
      def verticalBar = elem('|', 1, sp.height)
      def horizontalBar = elem('-', sp.width, 1)
      if (direction == 0)
        (corner beside horizontalBar) above (sp beside space)
      else if (direction == 1)
        (sp above space) beside (corner above verticalBar)
      else if (direction == 2)
        (space beside sp) above (horizontalBar beside corner)
      else
        (verticalBar above corner) beside (space above sp)
    }
  }

  def main(args: Array[String]) = {
    val nSides = args(0).toInt
    println(spiral(nSides, 0))
  }
}
```

Listing 10.14 - The Spiral application.

Because Spiral is a standalone object with a main method with the proper signature, it is a Scala application. Spiral takes one command-line argument, an integer, and draws a spiral with the specified number of edges. For example, you could draw a six-edge spiral, as shown on the left, and larger spirals, as shown on the right.

```
$ scala Spiral 6
+-----
|
| +-+
| + |
|   |
+---+
```

```
$ scala Spiral 11
+----------
|
| +------+
| |      |
| | +--+ |
| | |  | |
| | ++ | |
| |    | |
| +----+ |
|        |
+--------+
```

```
$ scala Spiral 17
+----------------
|
| +-------------+
| |             |
| | +---------+ |
| | |         | |
| | | +----+  | |
| | | |    |  | |
| | | | ++ |  | |
| | | |  | |  | |
| | | +--+ |  | |
| | |      |  | |
| | +------+  | |
| |           | |
| +-----------+ |
|               |
+---------------+
```

10.16 CONCLUSION

In this section, you saw more concepts related to object-oriented programming in Scala. Among others, you encountered abstract classes, inheritance and subtyping, class hierarchies, parametric fields, and

method overriding. You should have developed a feel for constructing a non-trivial class hierarchy in Scala. We'll work with the layout library again inChapter 14.

Footnotes for Chapter 10:

[1] Meyer, *Object-Oriented Software Construction* [Mey00]

[2] One flaw with this design is that because the returned array is mutable, clients could change it. For the book we'll keep things simple, but were ArrayElement part of a real project, you might consider returning a *defensive copy* of the array instead. Another problem is we aren't currently ensuring that every String element of the contents array has the same length. This could be solved by checking the precondition in the primary constructor and throwing an exception if it is violated.

[3] For more perspective on the difference between subclass and subtype, see the glossary entry for *subtype*.

[4] The reason that packages share the same namespace as fields and methods in Scala is to enable you to import packages (in addition to just the names of types) and the fields and methods of singleton objects. This is also something you can't do in Java. It will be described in Section 13.3.

[5] The protected modifier, which grants access to subclasses, will be covered in detail inChapter 13.

[6] In Java 1.5, an @Override annotation was introduced that works similarly to Scala's overridemodifier, but unlike Scala's override, is not required.

[7] This kind of polymorphism is called *subtyping polymorphism*. Another kind of polymorphism in Scala called *universal polymorphism* is discussed in Chapter 19.

[8] Meyers, *Effective C++* [Mey91]

[9] Eckel, *Thinking in Java* [Eck98]

[10] Class ArrayElement also has a composition relationship with Array, because its parametriccontents field holds a reference to an array of strings. The code for ArrayElement is shown inListing 10.5 here. Its composition relationship is represented in class diagrams by a diamond, as shown, for example, in Figure 10.1 here.