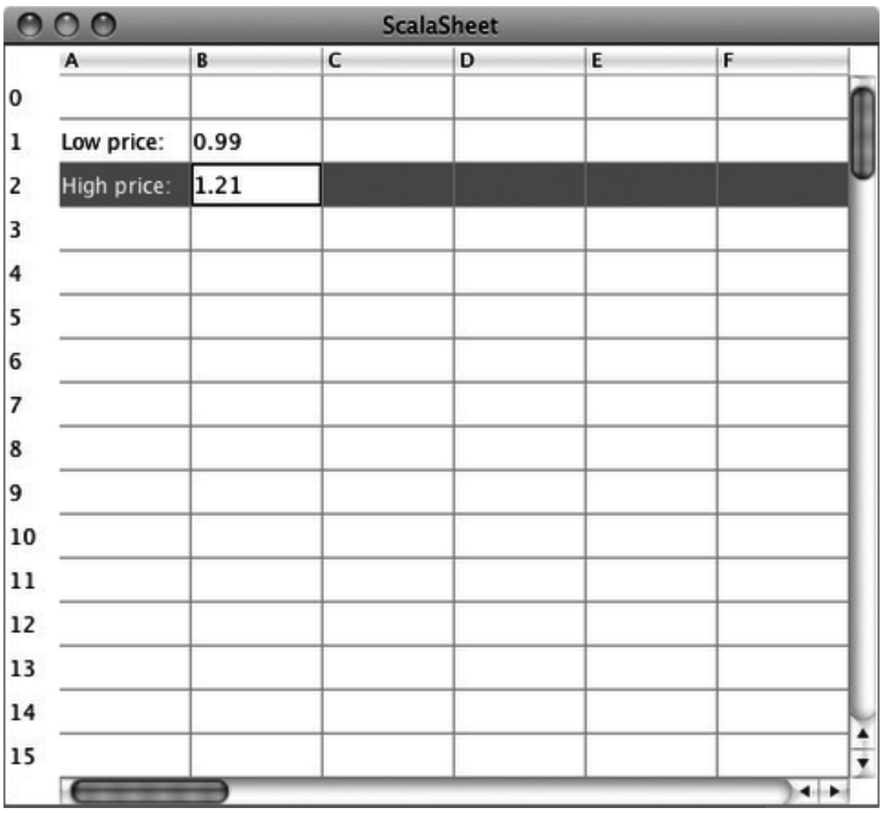
Chapter 35

The SCells Spreadsheet

In the previous chapters you saw many different constructs of the Scala programming language. In this chapter you'll see how these constructs play together in the implementation of a sizable application. The task is to write a spreadsheet application, which will be namedSCells.

There are several reasons why this task is interesting. First, everybody knows spreadsheets, so it is easy to understand what the application should do. Second, spreadsheets are programs that exercise a large range of different computing tasks. There's the visual aspect, where a spreadsheet is seen as a rich GUI application. There's the symbolic aspect, having to do with formulas and how to parse and interpret them. There's the calculational aspect, dealing with how to update possibly large tables incrementally. There's the reactive aspect, where spreadsheets are seen as programs that react in intricate ways to events. Finally, there's the component aspect where the application is constructed as a set of reusable components. All these aspects will be treated in depth in this chapter.



	A	B	C	D	E	F
0						
1	Low price:	0.99				
2	High price:	1.21				
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

Figure 35.1 - A simple spreadsheet table.

35.1 THE VISUAL FRAMEWORK

We'll start by writing the basic visual framework of the application. Figure 35.1 shows the first iteration of the user interface. You can see that a spreadsheet is a scrollable table. It has rows going from 0 to 99 and columns going from A to Z. You express this in Swing by defining a spreadsheet as a ScrollPane containing a Table. Listing 35.1 shows the code.

```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {

  val table = new Table(height, width) {
    rowHeight = 25
    autoResizeMode = Table.AutoResizeMode.Off
    showGrid = true
    gridColor = new java.awt.Color(150, 150, 150)
  }

  val rowHeader =
    new ListView((0 until height) map (_.toString)) {
      fixedCellWidth = 30
      fixedCellHeight = table.rowHeight
    }

  viewportView = table
  rowHeaderView = rowHeader
}
```

Listing 35.1 - Code for spreadsheet in Figure 35.1.

The spreadsheet component shown in Listing 35.1 is defined in packageorg.stairwaybook.scells, which will contain all classes, traits, and objects needed for the application. It imports from package scala.swing essential elements of Scala's Swing wrapper.Spreadsheet itself is a class that takes height and width (in numbers of cells) as parameters. The class extends ScrollPane, which gives it the scroll-bars at the bottom and right in Figure 35.1. It contains two sub-components named table and rowHeader.

The table component is an instance of an anonymous subclass of class scala.swing.Table. The four lines in its body set some of its attributes: rowHeight for the height of a table row in points,autoResizeMode to turn auto-sizing the table off, showGrid to show a grid of lines between cells, and gridColor to set the color of the grid to a dark gray.

The rowHeader component, which contains the row-number headers at the left of the spreadsheet in Figure 35.1, is a ListView that displays in its elements the strings 0 through 99.The two lines in its body fix the width of a cell to be 30 points and the height to be the same as the table's rowHeight.

The whole spreadsheet is assembled by setting two fields in ScrollPane. The field viewportView is set to the table, and the field rowHeaderView is set to the rowHeader list. The difference between the two views is that a view port of a scroll pane is the area that scrolls with the two bars, whereas the row header on the left stays fixed when you move the horizontal scroll bar. By some quirk, Swing already supplies by default a column header at the top of the table, so there's no need to define one explicitly.

```
    package org.stairwaybook.scells
    import swing._

    object Main extends SimpleSwingApplication {
      def top = new MainFrame {
        title = "ScalaSheet"
        contents = new Spreadsheet(100, 26)
      }
    }
```

Listing 35.2 - The main program for the spreadsheet application.

To try out the rudimentary spreadsheet shown in Listing 35.1, you just need to define a main program that creates the Spreadsheet component. Such a program is shown in Listing 35.2.

Program Main inherits from SimpleSwingApplication, which takes care of all the low-level details that need to be set up before a Swing application can be run. You only need to define the top-level window of the application in the top method. In our example, top is a MainFrame that has two elements defined: its title, set to "ScalaSheet," and its contents, set to an instance of classSpreadsheet with 100 rows and 26 columns. That's all. If you launch this application withscala org.stairwaybook.scells.Main, you should see the spreadsheet in Figure 35.1.

35.2 DISCONNECTING DATA ENTRY AND DISPLAY

If you play a bit with the spreadsheet written so far, you'll quickly notice that the output that's displayed in a cell is always exactly what you entered in the cell. A real spreadsheet does not behave like that. In a real spreadsheet, you would enter a formula and you'd see its value. So what is entered into a cell is different from what is displayed.

```
  package org.stairwaybook.scells
  import swing._

  class Spreadsheet(val height: Int, val width: Int)
      extends ScrollPane {

    val cellModel = new Model(height, width)
    import cellModel._

    val table = new Table(height, width) {

      // settings as before...

      override def rendererComponent(isSelected: Boolean,
          hasFocus: Boolean, row: Int, column: Int): Component =

        if (hasFocus) new TextField(userData(row, column))
        else
          new Label(cells(row)(column).toString) {
            xAlignment = Alignment.Right
          }

      def userData(row: Int, column: Int): String = {
        val v = this(row, column)
        if (v == null) "" else v.toString
      }
    }
    // rest as before...
  }
```

Listing 35.3 - A spreadsheet with a rendererComponent method.

As a first step to a real spreadsheet application, you should concentrate on disentangling data entry and display. The basic mechanism for display is contained in the rendererComponentmethod of class Table. By default, rendererComponent always displays what's entered. If you want to change that, you need to override rendererComponent to do something different. Listing 35.3shows a new version of Spreadsheet with a rendererComponent method.

The rendererComponent method overrides a default method in class Table. It takes four parameters. The isSelected and hasFocus parameters are Booleans that indicate whether the cell has been selected and whether it has focus, meaning that keyboard events will go into the cell. The remaining two parameters, row and column, give the cell's coordinates.

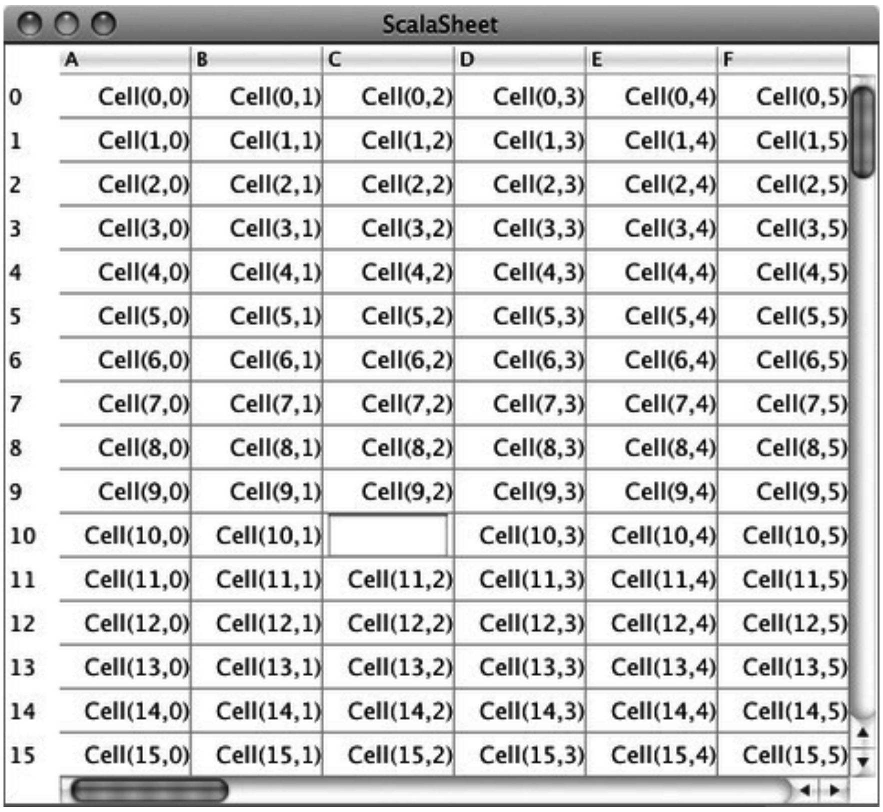
The new rendererComponent method checks whether the cell has input focus. If hasFocus is true, the cell is used for editing. In this case you want to display an editable TextField that contains the data the user has entered so far. This data is returned by the helper method userData, which displays the contents of the table at a given row and column. The contents are retrieved by the call this(row, column).[1] The userData method also takes care to display a null element as the empty string instead of "null."

So far so good. But what should be displayed if the cell does not have focus? In a real spreadsheet this would be the value of a cell. Thus, there are really two tables at work. The first table, named table contains what the user entered. A second "shadow" table contains the internal representation of cells and what should be displayed. In the spreadsheet example, this table is a two-dimensional array called cells. If a cell at a given row and column does not have editing focus, the rendererComponent method will display the element cells(row)(column). The element cannot be edited, so it should be displayed in a Label instead of in an editableTextField.

It remains to define the internal array of cells. You could do this directly in the Spreadsheetclass, but it's generally preferable to separate the view of a GUI component from its internal model. That's why in the example above the cells array is defined in a separate class namedModel. The model is integrated into the Spreadsheet by defining a value cellModel of type Model. The import clause that follows this val definition makes the members of cellModel available inside Spreadsheet without having to prefix them. Listing 35.4 shows a first simplified version of a Model class. The class defines an inner class, Cell, and a two-dimensional array, cells, ofCell elements. Each element is initialized to be a fresh Cell.

```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int) {

  case class Cell(row: Int, column: Int)

  val cells = Array.ofDim[Cell](height, width)

  for (i <- 0 until height; j <- 0 until width)
    cells(i)(j) = new Cell(i, j)
}
```

Listing 35.4 - First version of the Model class.



	A	B	C	D	E	F
0	Cell(0,0)	Cell(0,1)	Cell(0,2)	Cell(0,3)	Cell(0,4)	Cell(0,5)
1	Cell(1,0)	Cell(1,1)	Cell(1,2)	Cell(1,3)	Cell(1,4)	Cell(1,5)
2	Cell(2,0)	Cell(2,1)	Cell(2,2)	Cell(2,3)	Cell(2,4)	Cell(2,5)
3	Cell(3,0)	Cell(3,1)	Cell(3,2)	Cell(3,3)	Cell(3,4)	Cell(3,5)
4	Cell(4,0)	Cell(4,1)	Cell(4,2)	Cell(4,3)	Cell(4,4)	Cell(4,5)
5	Cell(5,0)	Cell(5,1)	Cell(5,2)	Cell(5,3)	Cell(5,4)	Cell(5,5)
6	Cell(6,0)	Cell(6,1)	Cell(6,2)	Cell(6,3)	Cell(6,4)	Cell(6,5)
7	Cell(7,0)	Cell(7,1)	Cell(7,2)	Cell(7,3)	Cell(7,4)	Cell(7,5)
8	Cell(8,0)	Cell(8,1)	Cell(8,2)	Cell(8,3)	Cell(8,4)	Cell(8,5)
9	Cell(9,0)	Cell(9,1)	Cell(9,2)	Cell(9,3)	Cell(9,4)	Cell(9,5)
10	Cell(10,0)	Cell(10,1)		Cell(10,3)	Cell(10,4)	Cell(10,5)
11	Cell(11,0)	Cell(11,1)	Cell(11,2)	Cell(11,3)	Cell(11,4)	Cell(11,5)
12	Cell(12,0)	Cell(12,1)	Cell(12,2)	Cell(12,3)	Cell(12,4)	Cell(12,5)
13	Cell(13,0)	Cell(13,1)	Cell(13,2)	Cell(13,3)	Cell(13,4)	Cell(13,5)
14	Cell(14,0)	Cell(14,1)	Cell(14,2)	Cell(14,3)	Cell(14,4)	Cell(14,5)
15	Cell(15,0)	Cell(15,1)	Cell(15,2)	Cell(15,3)	Cell(15,4)	Cell(15,5)

Figure 35.2 - Cells displaying themselves.

That's it. If you compile the modified Spreadsheet class with the Model class and run the Mainapplication you should see a window as in Figure 35.2.

The objective of this section was to arrive at a design where the displayed value of a cell is different from the string that was entered into it. This objective has clearly been met, albeit in a very crude way. In the new spreadsheet you can enter anything you want into a cell, but it will always display just its coordinates once it loses focus. Clearly, we are not done yet.

35.3 FORMULAS

In reality, a spreadsheet cell holds two things: An actual value and a formula to compute this value. There are three types of formulas in a spreadsheet:

1. Numeric values such as 1.22, -3, or 0.
2. Textual labels such as Annual sales, Deprecation, or total.
3. Formulas that compute a new value from the contents of cells, such as "=add(A1,B2)", or "=sum(mul(2, A2), C1:D16)"

A formula that computes a value always starts with an equals sign and is followed by an arithmetic expression. The SCells spreadsheet has a particularly simple and uniform convention for arithmetic expressions: every expression is an application of some function to a list of arguments. The function name is an identifier such as add for binary addition, or sumfor summation of an arbitrary number of operands. A function argument can be a number, a reference to a cell, a reference to a range of cells such as C1:D16, or another function application. You'll see later that SCells has an open architecture that makes it easy to install your own functions via mixin composition.

The first step to handling formulas is writing down the types that represent them. As you might expect, the different kinds of formulas are represented by case classes. Listing 35.5shows the contents of a file named Formulas.scala, where these case classes are defined:

```
package org.stairwaybook.scells

trait Formula

case class Coord(row: Int, column: Int) extends Formula {
  override def toString = ('A' + column).toChar.toString + row
}
case class Range(c1: Coord, c2: Coord) extends Formula {
  override def toString = c1.toString + ":" + c2.toString
}
case class Number(value: Double) extends Formula {
  override def toString = value.toString
}
case class Textual(value: String) extends Formula {
  override def toString = value
}
case class Application(function: String,
    arguments: List[Formula]) extends Formula {

  override def toString =
    function + arguments.mkString("(", ",", ")")
}
object Empty extends Textual("")
```

Listing 35.5 - Classes representing formulas.

Trait Formula, shown in Listing 35.5, has five case classes as children:

Coord	for cell coordinates such as A3,
Range	for cell ranges such as A3:B17,

Number	for floating-point numbers such as 3.1415,
Textual	for textual labels such as Deprecation,
Application	for function applications such as sum(A1,A2).

Each case class overrides the toString method so that it displays its kind of formula in the standard way shown above. For convenience there's also an Empty object that represents the contents of an empty cell. The Empty object is an instance of the Textual class with an empty string argument.

35.4 PARSING FORMULAS

In the previous section you saw the different kinds of formulas and how they display as strings. In this section you'll see how to reverse the process: to transform a user input string into a Formula tree. The rest of this section explains one by one the different elements of a classFormulaParsers, which contains the parsers that do the transformation. The class builds on the combinator framework given in Chapter 33. Specifically, formula parsers are an instance of the RegexParsers class explained in that chapter:

```
package org.stairwaybook.scells
import scala.util.parsing.combinator._

object FormulaParsers extends RegexParsers {
```

The first two elements of object FormulaParsers are auxiliary parsers for identifiers and decimal numbers:

```
def ident: Parser[String] = """[a-zA-Z_]\w*""".r
def decimal: Parser[String] = """-?\d+(\.\d*)?""".r
```

As you can see from the first regular expression above, an identifier starts with a letter or underscore. This is followed by an arbitrary number of "word" characters represented by the regular expression code \w, which recognizes letters, digits or underscores. The second regular expression describes decimal numbers, which consist of an optional minus sign, one or more digits that are represented by regular expression code \d, and an optional decimal part consisting of a period followed by zero or more digits.

The next element of object FormulaParsers is the cell parser, which recognizes the coordinates of a cell, such as C11 or B2. It first calls a regular expression parser that determines the form of a coordinate: a single letter followed by one or more digits. The string returned from that parser is then converted to a cell coordinate by separating the letter from the numerical part and converting the two parts to indices for the cell's column and row:

```
def cell: Parser[Coord] =
  """[A-Za-z]\d+""".r ^^ { s =>
    val column = s.charAt(0).toUpper - 'A'
    val row = s.substring(1).toInt
    Coord(row, column)
  }
```

Note that the cell parser is a bit restrictive in that it allows only column coordinates consisting of a single letter. Hence the number of spreadsheet columns is in effect restricted to be at most 26, because further columns cannot be parsed. It's a good idea to generalize the parser so that it accepts cells with several leading letters. This is left as an exercise to you.

The range parser recognizes a range of cells. Such a range is composed of two cell coordinates with a colon between them:

```
def range: Parser[Range] =
  cell~":"~cell ^^ {
    case c1~":"~c2 => Range(c1, c2)
  }
```

The number parser recognizes a decimal number, which is converted to a Double and wrapped in an instance of the Number class:

```
def number: Parser[Number] =
  decimal ^^ (d => Number(d.toDouble))
```

The application parser recognizes a function application. Such an application is composed of an identifier followed by a list of argument expressions in parentheses:

```
def application: Parser[Application] =
  ident~"("~repsep(expr, ",")~")" ^^ {
    case f~"("~ps~")" => Application(f, ps)
  }
```

The expr parser recognizes a formula expression—either a top-level formula following an `=', or an argument to a function. Such a formula expression is defined to be a cell, a range of cells, a number, or an application:

```
def expr: Parser[Formula] =
  range | cell | number | application
```

This definition of the expr parser contains a slight oversimplification because ranges of cells should only appear as function arguments; they should not be allowed as top-level formulas. You could change the formula grammar so that the two uses of expressions are separated, and ranges are excluded syntactically from top-level formulas. In the spreadsheet presented here such an error is instead detected once an expression is evaluated.

The textual parser recognizes an arbitrary input string, as long as it does not start with an equals sign (recall that strings that start with `=' are considered to be formulas):

```
def textual: Parser[Textual] =
  """[^=].*""".r ^^ Textual
```

The formula parser recognizes all kinds of legal inputs into a cell. A formula is either a number, or a textual entry, or a formula starting with an equals sign:

```
def formula: Parser[Formula] =
  number | textual | "="~>expr
```

This concludes the grammar for spreadsheet cells. The final method parse uses this grammar in a method that converts an input string into a Formula tree:

```
    def parse(input: String): Formula =
      parseAll(formula, input) match {
        case Success(e, _) => e
        case f: NoSuccess => Textual("[" + f.msg + "]")
      }
  } //end FormulaParsers
```

The parse method parses all of the input with the formula parser. If that succeeds, the resulting formula is returned. If it fails, a Textual object with an error message is returned instead.

```
    package org.stairwaybook.scells
    import swing._
    import event._

    class Spreadsheet(val height: Int, val width: Int) ... {
      val table = new Table(height, width) {
        ...
        reactions += {
          case TableUpdated(table, rows, column) =>
            for (row <- rows)
              cells(row)(column).formula =
                FormulaParsers.parse(userData(row, column))
        }
      }
    }
```

Listing 35.6 - A spreadsheet that parses formulas.

That's everything there is to parsing formulas. The only thing that remains is to integrate the parser into the spreadsheet. To do this, you can enrich the Cell class in class Model by a formulafield:

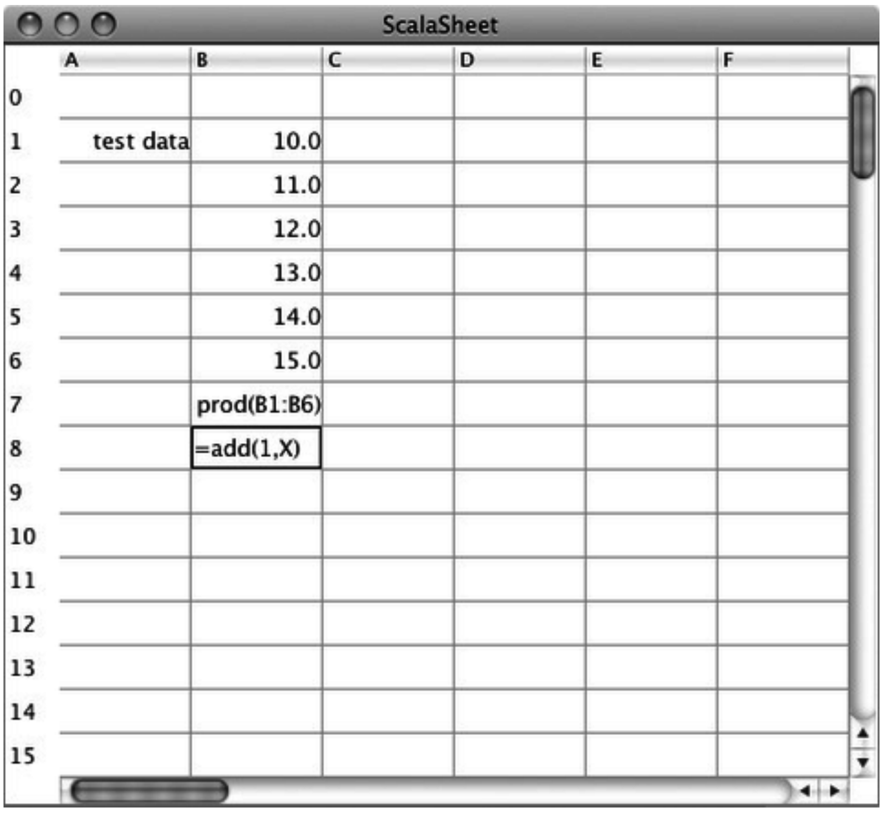
```
  case class Cell(row: Int, column: Int) {
    var formula: Formula = Empty
    override def toString = formula.toString
  }
```

In the new version of the Cell class, the toString method is defined to display the cell's formula. That way you can check whether formulas have been correctly parsed.

The last step in this section is to integrate the parser into the spreadsheet. Parsing a formula happens as a reaction to the user's input into a cell. A completed cell input is modeled in the Swing library by a TableUpdated event. The TableUpdated class is contained in packagescala.swing.event. The event is of the form:

```
  TableUpdated(table, rows, column)
```

It contains the table that was changed, as well as a set of coordinates of affected cells given byrows and column. The rows parameter is a range value of type Range[Int].[2] The column parameter is an integer. So in general a TableUpdated event can refer to several affected cells, but they would be on a consecutive range of rows and share the same column.



	A	B	C	D	E	F
0						
1	test data	10.0				
2		11.0				
3		12.0				
4		13.0				
5		14.0				
6		15.0				
7		prod(B1:B6)				
8		=add(1,X)				
9						
10						
11						
12						
13						
14						
15						

Figure 35.3 - Cells displaying their formulas.

Once a table is changed, the affected cells need to be re-parsed. To react to a TableUpdatedevent, you add a case to the reactions value of the table component, as is shown in Listing 35.6. Now, whenever the table is edited the formulas of all affected cells will be updated by parsing the corresponding user data. When compiling the classes discussed so far and launching the scells.Main application you should see a spreadsheet application like the one shown in Figure 35.3. You can edit cells by typing into them. After editing is done, a cell displays the formula it contains. You can also try to type some illegal input such as the one reading =add(1, X) in the field that has the editing focus in Figure 35.3. Illegal input will show up as an error message. For instance, once you'd leave the edited field in Figure 35.3 you should see the error message [`(' expected] in the cell (to see all of the error message you might need to widen the column by dragging the separation between the column headers to the right).

35.5 EVALUATION

Of course, in the end a spreadsheet should evaluate formulas, not just display them. In this section, we'll add the necessary components to achieve this.

What's needed is a method, evaluate, which takes a formula and returns the value of that formula in the current spreadsheet, represented as a Double. We'll place this method in a new trait, Evaluator. The method needs to access the cells field in class Model to find out about the current values of cells that are referenced in a formula. On the other hand, the Model class needs to call evaluate. Hence, there's a mutual dependency between the Model and the Evaluator. A good way to express such mutual dependencies between classes was shown in Chapter 29: you use inheritance in one direction and self types in the other.

In the spreadsheet example, class Model inherits from Evaluator and thus gains access to itsevaluation method. To go the other way, class Evaluator defines its self type to be Model, like this:

```
package org.stairwaybook.scells
trait Evaluator { this: Model => ...
```

That way, the this value inside class Evaluator is assumed to be Model and the cells array is accessible by writing either cells or this.cells.

Now that the wiring is done, we'll concentrate on defining the contents of class Evaluator.Listing 35.7 shows the implementation of the evaluate method. As you might expect, the method contains a pattern match over the different types of formulas. For a coordinateCoord(row, column), it returns the value of the cells array at that coordinate. For a numberNumber(v), it returns the value v. For a textual label Textual(s), it returns zero. Finally, for an application Application(function, arguments), it computes the values of all arguments, retrieves a function object corresponding to the function name from an operations table and applies that function to all argument values.

```
  def evaluate(e: Formula): Double = try {
    e match {
      case Coord(row, column) =>
        cells(row)(column).value
      case Number(v) =>
        v
      case Textual(_) =>
        0
      case Application(function, arguments) =>
        val argvals = arguments flatMap evalList
        operations(function)(argvals)
    }
  } catch {
    case ex: Exception => Double.NaN
  }
```

Listing 35.7 - The evaluate method of trait Evaluator.

The operations table maps function names to function objects. It is defined as follows:

```
type Op = List[Double] => Double
val operations = new collection.mutable.HashMap[String, Op]
```

As you can see from this definition, operations are modeled as functions from lists of values to values. The Op type introduces a convenient alias for the type of an operation.

The computation in evaluate is wrapped in a try-catch to guard against input errors. There are actually quite a few things that can go wrong when evaluating a cell formula: coordinates might be out of range; function names might be undefined; functions might have the wrong number of arguments; arithmetic operations might be illegal or overflow. The reaction to any of these errors is the same: a "not-a-number" value is returned. The returned value,Double.NaN, is the IEEE representation for a computation that does not have a representable floating-point value. This might happen because of an overflow or a division by zero, for example. The evaluate method of Listing 35.7 chooses to return the same value also for all other kinds of errors. The advantage of this scheme is that it's simple to understand and doesn't require much code to implement. Its disadvantage is that all kinds of errors are lumped together, so a spreadsheet user does not get any detailed feedback on what went wrong. If you wish you can experiment with more refined ways of representing errors in theSCells application.

The evaluation of arguments is different from the evaluation of top-level formulas. Arguments may be lists whereas top-level functions may not. For instance, the argument
expression A1:A3 in sum(A1:A3) returns the values of cells A1, A2, A3 in a list. This list is then passed to the sum operation. It's also possible to mix lists and single values in argument expressions, for instance the operation sum(A1:A3, 1.0, C7), which would sum up five values. To handle arguments that might evaluate to lists, there's another evaluation function, calledevalList, which takes a formula and returns a list of values:

```
  private def evalList(e: Formula): List[Double] = e match {
    case Range(_, _) => references(e) map (_.value)
    case _ => List(evaluate(e))
  }
```

If the formula argument passed to evalList is a Range, the returned value is a list consisting of the values of all cells referenced by the range. For every other formula the result is a list consisting of the single result value of that formula. The cells referenced by a formula are computed by a third function, references. Here is its definition:

```
    def references(e: Formula): List[Cell] = e match {
      case Coord(row, column) =>
        List(cells(row)(column))
      case Range(Coord(r1, c1), Coord(r2, c2)) =>
        for (row <- (r1 to r2).toList; column <- c1 to c2)
        yield cells(row)(column)
      case Application(function, arguments) =>
        arguments flatMap references
      case _ =>
        List()
    }
  } // end Evaluator
```

The references method is actually more general than needed right now in that it computes the list of cells referenced by any sort of formula, not just a Range formula. It will turn out later that the added functionality is needed to compute the sets of cells that need updating. The body of the method is a straightforward pattern match on kinds of formulas. For a coordinateCoord(row, column), it returns a single-element list containing the cell at that coordinate. For a range expression Range(coord1, coord2), it returns all cells between the two coordinates, computed by a for expression. For a function application Application(function, arguments), it returns the cells referenced by each argument expression, concatenated via flatMap into a single list. For the other two types of
formulas, Textual and Number, it returns an empty list.

35.6 OPERATION LIBRARIES

The class Evaluator itself defines no operations that can be performed on cells: its operationstable is initially empty. The idea is to define such operations in other traits, which are then mixed into the Model class. Listing 35.8 shows an example trait that implements common arithmetic operations:

```
    package org.stairwaybook.scells
    trait Arithmetic { this: Evaluator =>
      operations += (
        "add"  -> { case List(x, y) => x + y },
        "sub"  -> { case List(x, y) => x - y },
        "div"  -> { case List(x, y) => x / y },
        "mul"  -> { case List(x, y) => x * y },
        "mod"  -> { case List(x, y) => x % y },
        "sum"  -> { xs => (0.0 /: xs)(_ + _) },
        "prod" -> { xs => (1.0 /: xs)(_ * _) }
      )
    }
```

Listing 35.8 - A library for arithmetic operations.

Interestingly, this trait has no exported members. The only thing it does is populate theoperations table during its initialization. It gets access to that table by using a self typeEvaluator, *i.e.*, by the same technique the Arithmetic class uses to get access to the model.

Of the seven operations that are defined by the Arithmetic trait, five are binary operations and two take an arbitrary number of arguments. The binary operations all follow the same schema. For instance, the addition operation add is defined by the expression:

```
  { case List(x, y) => x + y }
```

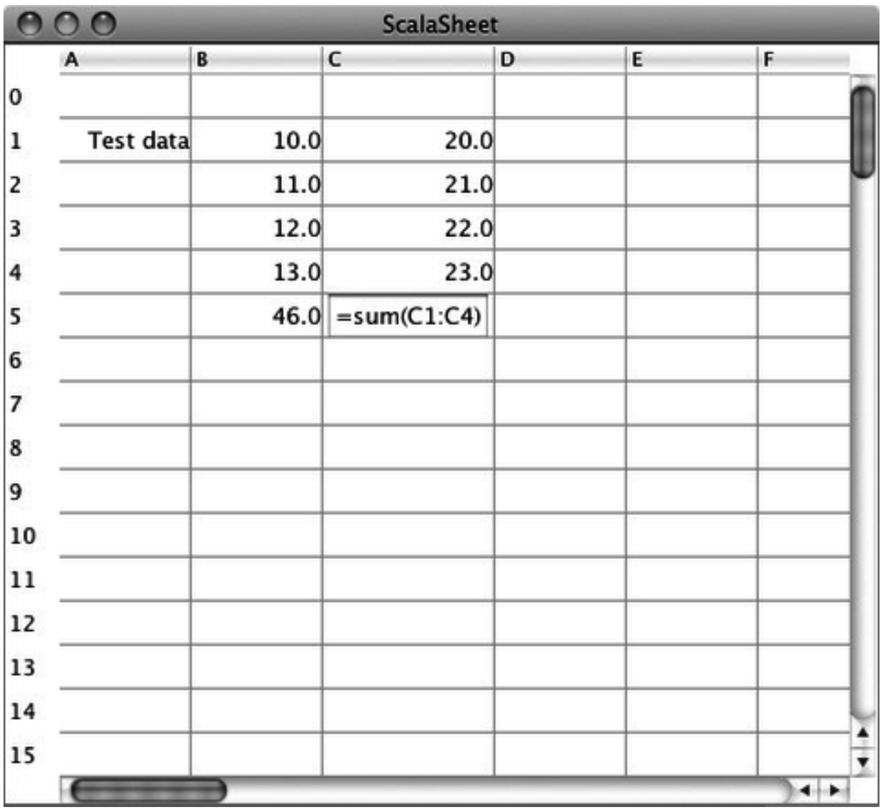
That is, it expects an argument list consisting of two elements x and y and returns the sum of xand y. If the argument list contains a number of elements different from two, a MatchError is thrown. This corresponds to the general "let it crash" philosophy of SCell's evaluation model, where incorrect input is expected to lead to a runtime exception that then gets caught by the try-catch inside the evaluation method.

The last two operations, sum and prod, take a list of arguments of arbitrary length and insert a binary operation between successive elements. So they are instances of the "fold left" schema that's expressed in class List by the /: operation. For instance, to sum a list of numbersList(x, y, z), the operation computes 0 + x + y + z. The first operand, 0, is the result if the list is empty.

You can integrate this operation library into the spreadsheet application by mixing theArithmetic trait into the Model class, like this:

```
  package org.stairwaybook.scells

  class Model(val height: Int, val width: Int)
      extends Evaluator with Arithmetic {

    case class Cell(row: Int, column: Int) {
      var formula: Formula = Empty
      def value = evaluate(formula)

      override def toString = formula match {
        case Textual(s) => s
        case _ => value.toString
      }
    }

    ... // rest as before
  }
```

Another change in the Model class concerns the way cells display themselves. In the new version, the displayed value of a cell depends on its formula. If the formula is a Textual field, the contents of the field are displayed literally. In all other cases, the formula is evaluated and the result value of that evaluation is displayed.



	A	B	C	D	E	F
0						
1	Test data	10.0	20.0			
2		11.0	21.0			
3		12.0	22.0			
4		13.0	23.0			
5		46.0	=sum(C1:C4)			
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

Figure 35.4 - Cells that evaluate.

If you compile the changed traits and classes and relaunch the Main program you get something that starts to resemble a real spreadsheet. Figure 35.4 shows an example. You can enter formulas into cells and get them to evaluate themselves. For instance, once you close the editing focus on cell C5 in Figure 35.4, you should see 86.0, the result of evaluating the formula sum(C1:C4).

However, there's a crucial element still missing. If you change the value of cell C1 in Figure 35.4 from 20 to 100, the sum in cell C5 will not be automatically updated to 166. You'll have to click on C5 manually to see a change in its value. What's still missing is a way to have cells recompute their values automatically after a change.

35.7 CHANGE PROPAGATION

If a cell's value has changed, all cells that depend on that value should have their results recomputed and redisplayed. The simplest way to achieve this would be to recompute the value of every cell in the spreadsheet after each change. However such an approach does not scale well as the spreadsheet grows in size.

A better approach is to recompute the values of only those cells that refer to a changed cell in their formula. The idea is to use an event-based publish/subscribe framework for change propagation: once a cell gets assigned a formula, it will subscribe to be notified of all value changes in cells to which the formula refers. A value change in one of these cells will trigger a re-evaluation of the subscriber cell. If such a re-evaluation causes a change in the value of the cell, it will in turn notify all cells that depend on it. The process continues until all cell values have stabilized, *i.e.*, there are no more changes in the values of any cell.[3]

The publish/subscribe framework is implemented in class Model using the standard event mechanism of Scala's Swing framework. Here's a new (and final) version of this class:

```
package org.stairwaybook.scells
import swing._

class Model(val height: Int, val width: Int)
extends Evaluator with Arithmetic {
```

Compared to the previous version of Model, this version adds a new import of swing._, which makes Swing's event abstractions directly available.

The main modifications of class Model concern the nested class Cell. Class Cell now inherits from Publisher, so that it can publish events. The event-handling logic is completely contained in the setters of two properties: value and formula. Here is Cell's new version:

```
case class Cell(row: Int, column: Int) extends Publisher {
  override def toString = formula match {
    case Textual(s) => s
    case _ => value.toString
  }
```

To the outside, it looks like value and formula are two variables in class Cell. Their actual implementation is in terms of two private fields that are equipped with public getters, valueand formula, and setters, value_= and formula_=. Here are the definitions implementing the valueproperty:

```
private var v: Double = 0
def value: Double = v
def value_=(w: Double) = {
  if (!(v == w || v.isNaN && w.isNaN)) {
    v = w
```

```
      publish(ValueChanged(this))
    }
  }
```

The value_= setter assigns a new value w to the private field v. If the new value is different from the old one, it also publishes a ValueChanged event with the cell itself as argument. Note that the test whether the value has changed is a bit tricky because it involves the value NaN. The Java spec says that NaN is different from every other value, including itself! Therefore, a test whether two values are the same has to treat NaN specially: two values v, w are the same if they are equal with respect to ==, or they are both the value NaN, *i.e.*, v.isNaN and w.isNaN both yieldtrue.

Whereas the value_= setter does the publishing in the publish/subscribe framework, theformula_= setter does the subscribing:

```
  private var f: Formula = Empty
  def formula: Formula = f
  def formula_=(f: Formula) = {
    for (c <- references(formula)) deafTo(c)
    this.f = f
    for (c <- references(formula)) listenTo(c)
    value = evaluate(f)
  }
```

If a cell is assigned a new formula, it first unsubscribes with deafTo from all cells referenced by the previous formula value. It then stores the new formula in the private variable f and subscribes with listenTo to all cells referenced by it. Finally, it recomputes its value using the new formula.

The last piece of code in the revised class Cell specifies how to react to a ValueChanged event:

```
    reactions += {
      case ValueChanged(_) => value = evaluate(formula)
    }
  } // end class Cell
```

The ValueChanged class is also contained in class Model:

```
  case class ValueChanged(cell: Cell) extends event.Event
```

The rest of class Model is as before:

```
    val cells = Array.ofDim[Cell](height, width)

    for (i <- 0 until height; j <- 0 until width)
      cells(i)(j) = new Cell(i, j)
  } // end class Model

    package org.stairwaybook.scells
    import swing._, event._

    class Spreadsheet(val height: Int, val width: Int)
        extends ScrollPane {

      val cellModel = new Model(height, width)
      import cellModel._
      val table = new Table(height, width) {
        ... // settings as in Listing ???

        override def rendererComponent(
             isSelected: Boolean, hasFocus: Boolean,
             row: Int, column: Int) =
          ... // as in Listing ???

        def userData(row: Int, column: Int): String =
          ... // as in Listing ???

        reactions += {
          case TableUpdated(table, rows, column) =>
            for (row <- rows)
              cells(row)(column).formula =
                FormulaParsers.parse(userData(row, column))
          case ValueChanged(cell) =>
            updateCell(cell.row, cell.column)
        }

        for (row <- cells; cell <- row) listenTo(cell)
      }

      val rowHeader = new ListView(0 until height) {
        fixedCellWidth = 30
        fixedCellHeight = table.rowHeight
      }

      viewportView = table
      rowHeaderView = rowHeader
    }
```

Listing 35.9 - The finished spreadsheet component.

The spreadsheet code is now almost complete. The final piece missing is the re-display of modified cells. So far, all value propagation concerned the internal Cell values only; the visible table was not affected. One way to change this would be to add a redraw command to the value_= setter. However, this would undermine the strict separation between model and view that you have seen so far. A more modular solution is to notify the table of allValueChanged events and let it do the redrawing itself. Listing 35.9 shows the final spreadsheet component, which implements this scheme.

Class Spreadsheet of Listing 35.9 has only two new revisions. First, the table component now subscribes with listenTo to all cells in the model. Second, there's a new case in the table's reactions: if it is notified of a ValueChanged(cell) event, it demands a redraw of the corresponding cell with a call of updateCell(cell.row, cell.column).

35.8 CONCLUSION

The spreadsheet developed in this chapter is fully functional, even though at some points it adopts the simplest solution to implement rather than the most convenient one for the user. That way, it could be written in just under 200 lines of code. Nevertheless, the architecture of the spreadsheet makes modifications and extensions easy. In case you would like to experiment with the code a bit further, here are some suggestions of what you could change or add:

1. You could make the spreadsheet resizable, so that the number of rows and columns can be changed interactively.
2. You could add new kinds of formulas, for instance binary operations, or other functions.
3. You might think about what to do when cells refer recursively to themselves. For instance, if cell A1 holds the formula add(B1, 1) and cell B1 holds the formula mul(A1, 2), a re-evaluation of either cell will trigger a stack-overflow. Clearly, that's not a very good solution. As alternatives, you could either disallow such a situation, or just compute one iteration each time one of the cells is touched.
4. You could enhance error handling, giving more detailed messages describing what went wrong.
5. You could add a formula entry field at the top of the spreadsheet, so that long formulas could be entered more conveniently.

At the beginning of this book we stressed the scalability aspect of Scala. We claimed that the combination of Scala's object-oriented and functional constructs makes it suitable for programs ranging from small scripts to very large systems. The spreadsheet presented here is clearly still a small system, even though it would probably take up much more than 200 lines in most other languages. Nevertheless, you can see many of the details that make Scala scalable at play in this application.

The spreadsheet uses Scala's classes and traits with their mixin composition to combine its components in flexible ways. Recursive dependencies between components are expressed using self types. The need for static state is completely eliminated—the only top-level components that are not classes are formula trees and formula parsers, and both of these are purely functional. The application also uses higher-order functions and pattern matching extensively, both for accessing formulas and for event handling. So it is a good showcase of how functional and object-oriented programming can be combined smoothly.

One important reason why the spreadsheet application is so concise is that it can base itself on powerful libraries. The parser combinator library provides in effect an internal domain-specific language for writing parsers. Without it, parsing formulas would have been much more difficult. The event handling in Scala's Swing libraries is a good example of the power of control abstractions. If you know Java's Swing libraries, you probably appreciate the conciseness of Scala's reactions concept, particularly when compared to the tedium of writing notify methods and implementing listener interfaces in the classical publish/subscribe design pattern. So the spreadsheet demonstrates the benefits of extensibility, where high-level libraries can be made to look just like language extensions.

Footnotes for Chapter 35:

[1] Although "this(row, column)" may look similar to a constructor invocation, it is in this case an invocation of the apply method on the current Table instance.

[2] Range[Int] is also the type of a Scala expression such as "1 to N".

[3] This assumes that there are no cyclic dependencies between cells. We discuss dropping this assumption at the end of this chapter.