Mendicant University Learning Series

#1 Representing Tabular Data in Ruby

Andrea Singh and Felipe Doria

Introduction

Mendicant University (MU) is a free online school aimed at socially minded software developers. It was founded in 2010 by Gregory Brown, author of the "Practicing Ruby" weekly newsletter and original developer of the PDF generation library Prawn.

The on-going MU three-week core courses focus on practical problems, designed to deepen the students' understanding of software programming. To date, over sixty students have successfully completed the course and achieved MU alumni status.

Once a course session is completed, the assignments are posted publicly and are can be used for self-study. The MU Learning Series was conceived as a means of providing an "under the hood" examination of real solutions submitted by alumni. The intention is to allow a glimpse into the type of reasoning that went into coming up with one possible solution.

In this first installment we will look at one of the assignments from the MU session of September 2010.

The Problem

The task was to build a general purpose table structure that could be used in a wide range of data processing scenarios. You can view the exact list of requirements on Github here.

At a minimum, the API of the table structure needed to implement the following features:

- add, insert and delete rows and columns
- support named column headers
- access a particular row, column or cell
- update values of an entire row or column
- filter rows and columns by given criteria

As the end product is likely to be fairly complex, it would be easy to get overwhelmed were we to try to implement all the features at once. If you read through the full list of requirements, you can see that they become progressively more complex. This type of problem lends itself particularly well to a Test Driven Development (TDD) approach, that is, developing the solution in small incremental step. We can start with one basic requirement, design a test for it and then write the code to make it pass. Then repeat the same procedure for the next requirement, and so on.

The first four chapters are a walk-through of this TDD approach and, as we shall see, the resulting solution is somewhat similar to that submitted by the majority of students in this session. In the later chapters, we will discuss how a naive implementation of TDD can result in failing to account for potential errors - in this particular case, situations involving data corruption, bad user input and problems with column names. Finally, in the concluding chapters, we'll take a look at some of the more unique approaches that made a deliberate attempt to incorporate a variety of Ruby best practices.

Test Environment

We will use the code>Test::Unit/code> library from the Ruby stdlib and the contest gem. The latter adds a thin layer on top of the Test::Unit API, allowing us to write for example:

```
test "should do stuff" do
  assert true
end
```

instead of

```
def test_should_do_stuff
  assert true
end
```

The contest library also supports nested context blocks, which make the organization of your test suite easier. Like in RSpec, every context block can have its own setup and teardown methods and nested contexts inherit the setup/teardown from their parents.

CHAPTER 1

Setting Up the Table

The most intuitive way to represent a table in Ruby is using a two dimensional array - where an individual row is an array and the collection of rows is also an array.

In keeping with TDD practices, we'll begin with the simplest possible scenario, in this case, initializing an empty table object.

To verify that the table is indeed empty, we first write a test to check that the table contains no rows:

```
require 'test/unit'
require 'contest'

# This is a Test::Unit test suite
# From now on we will omit this class declaration
class TableTest < Test::Unit::TestCase

  test "can be initialized empty" do
    table = Table.new
    assert_equal [], table.rows
  end

end
```

Making this test pass is as trivial as setting up a Table class with a single instance method - rows() - that returns an empty array. It is standard practice in TDD to "fake" return values - in this case, an empty array - to just satisfy the test requirements. As we progress, these stand-in return values will, of course, be replaced by more meaningful code.

```
class Table
  def rows
    []
  end
end
```

Next, we need the ability to add rows to the empty table. Let's write a test for that:

```
test "row can be appended after empty initialization" do
  table = Table.new
  table.add_row([1,2,3])
  assert_equal [[1,2,3]], table.rows
end
```

To satisfy this requirement, the rows() method can no longer simply return an empty array. Ideally, we want to make this new test pass without causing the first one to fail. This is the most straight forward code that would cause both tests to pass:

```
class Table
  attr_reader :rows

  def initialize
    @rows = []
  end

  def add_row(row)
    @rows << row
  end
end
```

Given a large enough dataset, populating the table row by row would no longer be practical. It would be preferable to have the option to initialize the table with data. To accomplish this, we can pass a two-dimensional array to the initializer method of the Table class.

We need some sample data to work with while testing. In order to have it available to all of our tests, we'll stick it in a setup block:

```
setup do
  @data = [["name",   "age", "occupation"  ],
           ["Tom",     32,   "engineer"    ],
           ["Beth",    12,   "student"     ],
           ["George",  45,   "photographer"],
           ["Laura",   23,   "aviator"     ],
           ["Marilyn", 84,   "retiree"     ]]
end
```

Next we need to hook up our Table class in such a way that it can accept a two dimensional array as an argument upon initialization.

```
test "can be initialized with a two-dimensional array" do
  table = Table.new(@data)
  assert_equal @data, table.rows
end
```

```
class Table
  attr_reader :rows

  def initialize(data = [])
    @rows = data
  end

  def add_row(row)
    @rows << row
  end
end
```

Another common feature of tables is to have named columns. In a two-dimensional array the column names could be represented by the first nested array with the following arrays being the actual data rows.

As things exist now, the first row of @data represents the column names. Therefore, when we try to access what we'd semantically expect to be the first row of data, the column names are returned instead:

```
>> table = Table.new(@data)
>> table.rows[0]
=> ["name", "age", "occupation"]
```

To remedy this, we could extract the first row and assign it to a separate variable, as follows:

```
test "first row represents column names" do
  table = Table.new(@data)
  assert_equal ["name", "age", "occupation"], table.headers
end
```

```
class Table
  attr_reader :rows, :headers

  def  initialize(data = [])
    if !data.empty?
      @headers = data.shift
      @rows = data
    else
      @rows, @headers = [], []
    end
  end

  def add_row(row)
    @rows << row
  end
end
```

Here have reached a first milestone of sorts. We are able to initialize a Table with or without data and add rows to it manually. We have also laid the foundation to support named columns.

As you can see, taking a test driven approach allows us to make sure that the code is behaving as it should by focusing on small tasks. Following TDD principles also enables us to write code that progressively takes us in the right direction without having to worry about the full set of requirements at the outset.

Although it's a good start, as we shall see, this implementation has some undesired consequences. It corrupts the initialization data and fails to vet user input. Rather than cover those issues now, we'll keep building on the requirements and tackle these pitfalls in chapter 5.

CHAPTER 2

Data Access

Currently, the default behavior is for the Table class to siphon off the first row of input data as the column headers :

```
>> table = Table.new [[1, 2, 3], [4,5,6]]
>> table.rows[0]
=> [4,5,6]

>> table.headers
=> [1,2,3]
```

While we want to allow for column names to be set, it would be better if that feature were optional. In other words, the setting of column names needs to be configurable. In this case, we need to edit a previous test:

```
test "can be initialized with a two-dimensional array" do
  table = Table.new(@data)
  assert_equal @data, table.rows
  assert_equal [],    table.headers
end

test "first row considered column names, if indicated" do
  table = Table.new(@data.dup, :headers => true)
  assert_equal @data[1..-1], table.rows
  assert_equal @data[0],     table.headers
end


class Table
  attr_reader :rows, :headers

  def initialize(data = [], options = {})
    @headers = options[:headers] ? data.shift : []
    @rows = data
  end

  def add_row(row)
    @rows << row
  end
end
```

Ideally, we would like the API to be able to access the columns either by name or by index. This would allow for data retrieval, for example, by either asking for "the 'name' field in the second row" or for "the first column in the second row".

This is what the tests and implementation for this feature could look like:

```
test "cell can be referred to by column name and row index" do
  table = Table.new(@data, :headers => true)
  assert_equal "Beth", table[1, "name"]
end

test "cell can be referred to by column index and row index" do
  table = Table.new(@data, :headers => true)
  assert_equal "Beth", table[1, 0]
end


class Table
  attr_reader :rows, :headers

  def initialize(data = [], options = {})
    @headers = options[:headers] ? data.shift : []
    @rows = data
  end

  def [](row, col)
    col = column_index(col)
    rows[row][col]
  end

  def column_index(pos)
    i = @headers.index(pos)
    i.nil? ? pos : i
  end

  def add_row(row)
    @rows << row
  end
end
```

To recap what we have done so far: the table can be initialized with configurable data and individual data cells can be accessed in an intuitive manner. In the next chapter we'll start to implement some of the more tricky requirements.

CHAPTER 3
Row Manipulations

We have already set up a method to retrieve the collection of rows. We can also append a new row at the end (see the add_row() method in Chapter 1). Now we want to be able to retrieve a single row. Following is the code to retrieve the contents of a particular row:

```
# We will omit this context declaration for the remainder of the chapter
context "row manipulations" do
  setup do
    @table = Table.new(@data, :headers => true)
  end

  test "can retrieve a row" do
    assert_equal ["George", 45,"photographer"], @table.row(2)
  end
end


class Table

  def row(i)
    rows[i]
  end
end
```

What if we want to insert a new row in a particular position and not at the end?

```
test "can insert a row at any position" do
  @table.add_row(["Jane", 19, "shop assistant"], 2)
  assert_equal ["Jane", 19, "shop assistant"], @table.row(2)
end
```

Instead of creating a brand new method for inserting a row at a particular position, we simply change the existing add_row() method to take an optional second argument representing the position where the new row should be inserted. If we don't send in a position, the new row will be appended at the end by default.

```
class Table

  def add_row(new_row, pos=nil)
    i = pos.nil? ? rows.length : pos
    rows.insert(i, new_row)
  end
end
```

To proceed with deleting a particular row from our table, we set up a test to check that a deleted row has actually been replaced:

```
test "can delete a row" do
  to_be_deleted = @table.row(2)
  @table.delete_row(2)
  assert_not_equal to_be_deleted, @table.row(2)
end
```

```
class Table

  def delete_row(pos)
    @rows.delete_at(pos)
  end
end
```

Next, we need to write a method to trigger a row-level transformation for altering all the data cells of that row in a user-defined way. In other words, we have to design our API in such a way it can accept the transformation code as an argument. This is where blocks come into play.

```
test "transform row cells" do
  @table.transform_row(0) do |cell|
    cell.is_a?(String) ? cell.upcase : cell
  end

  assert_equal ["TOM", 32, "ENGINEER"], @table.row(0)
end
```

```
class Table

  def transform_row(pos, &block)
    @rows[pos].map!(&block)
  end
end
```

Here we take advantage of the Array#map! method - which can accept a block as an argument - and delegate the heavy lifting over to it.

There are other use cases where the API would need to respond to user-defined blocks. Say we want to reduce our table data to only contain records of people under 30. That would mean that we'd have to check every row for that condition (age under 30) and only keep those rows that meet this criteria.

```
test "reduce the rows to those that meet a particular conditon" do
  @table.select_rows do |row|
    row[1] < 30
  end
  assert !@table.rows.include?(["George", 45,"photographer"])
end
```

Again, there's a handy Array method to accomplish this, namely Array#select!

```
class Table

  def select_rows(&block)
    @rows.select!(&block)
  end
end
```

So far so good. Our table implementation is starting to take shape. Let's implement the final set of requirements in the next chapter.

CHAPTER 4

Column Manipulations

By modeling the table data as a two-dimensional array, we were able to easily access and manipulate rows using common Array methods. However, as we turn our attention to column functionality, we shall see that this "row-centric" representation is not without cost. For example, retrieving the contents of a particular column would necessitate iterating through each row, extracting the data cell of that column and then mapping it to a new array.

To this effect, one might be tempted to use a convenient array method, such as Array#transpose to temporarily remap the rows as columns. Here is a quick demonstration of what this method accomplishes:

```
>> @data.transpose
=> [["name", "Tom", "Beth", "George", "Laura", "Marilyn"],
    ["age", 32, 12, 45, 23, 84],
    ["occupation", "engineer", "student", "photographer", "aviator", "retiree"]]
```

This would allow us to essentially duplicate all the row methods, re-purposed for column manipulation. But herein lies the rub. Transposing the data in this fashion necessitates the creation a new array, effectively doubling the data held in memory. Consider for example the use of tranpose() to conveniently insert a column. Something like:

```
def insert_column(pos)
  columns = @rows.transpose
  columns.insert_at(pos)
  @rows = columns.transpose
end
```

As you can see from the code above, we have to use transpose() twice, once to map the rows to columns and a second time to update the rows. Extra steps and processing time are required.

Another, less obvious drawback is that transpose is quite picky about its input. Consider the following:

```
>> table = [[1, 2, 3], [4,  5], [6]] # rows with different lengths
>> table.transpose
=> IndexError: element size differs (2 should be 3)
```

Now that we have seen that there is viable alternative to remapping the rows to columns, we have two choices: resign ourselves to extract columns from rows as needed or re-think the entire approach to this problem altogether. We'll continue exploring the code for the first option now and afterwards look at alternate design patterns for table representation.

Here then is test that demonstrates retrieving a column (both by name and index):

```
# We will omit this context declaration for the remainder of the chapter
context "column manipulations" do
  setup do
    @table = Table.new(@data, :headers => true)
  end

  test "can access a column by its name" do
    assert_equal ["Tom", "Beth", "George", "Laura", "Marilyn"],
                 @table.column("name")
  end

  test "can access a column by its index" do
    assert_equal ["Tom", "Beth", "George", "Laura", "Marilyn"],
                 @table.column(0)
  end
end
```

To implement this, we map the rows by the index of the column in question:

```
class Table

  def column_index(pos)
    i = @headers.index(pos)
    i.nil? ? pos : i
  end

  def column(pos)
    i = column_index(pos)
    @rows.map { |row| row[i] }
  end
end
```

Since we have already implemented column names and are saving a reference to them in a separate variable - @headers -, adding support for renaming a column simply requires replacing one item in the @headers array:

```
test "can rename a column" do
  @table.rename_column("name", "first name")
  assert_equal ["first name", "age", "occupation"], @table.headers
end
```

```
class Table

  def rename_column(old_name, new_name)
    i = @headers.index(old_name)
    @headers[i] = new_name
  end
end
```

As with rows, we want to be able to expand our data set by appending or inserting a new column. We will implement adding a column in similar fashion to the equivalent row method. Again, we want to support a position argument for insertion and a default behavior of appending the column at the end. However, we have to remember to take into account the column names along with the fact that they are optional.

```
test "can append a column" do
  to_append = ["location", "Italy", "Mexico", "USA", "Finland", "China"]
  @table.add_column(to_append)
  assert_equal ["name", "age", "occupation", "location"], @table.headers
  assert_equal 4, @table.rows.first.length
end

test "can insert a column at any position" do
  to_append = ["last name", "Brown", "Crimson", "Denim", "Ecru", "Fawn"]
  @table.add_column(to_append, 1)
  assert_equal ["name", "last name", "age", "occupation"], @table.headers
  assert_equal "Brown", @table[0,1]
end
```

The add_column() method needs to know whether headers are being used or not. As such, we store that option in a boolean variable, @header_support. Of course this is a matter of taste. We could just as easily have checked whether the @headers array is empty.

```
class Table
  attr_reader :rows, :headers, :header_support

  def initialize(data = [], options = {})
    @header_support = options[:headers]
    @headers = @header_support ? data.shift : []
    @rows = data
  end

  def add_column(col, pos=nil)
    i = pos.nil? ? rows.first.length : pos
    if header_support
      @headers.insert(i, col.shift)
    end
    @rows.each do |row|
      row.insert(i, col.shift)
    end
  end
end
```

A similar procedure is needed for deleting a column. That is, in each row we need to delete the item belonging to that column.

```
test "can delete a column from any position" do
  @table.delete_column(1)
  assert_equal ["name", "occupation"], @table.headers
  assert_equal ["Tom", "engineer"],    @table.row(0)
end


class  Table

  def delete_column(pos)
    pos = column_index(pos)
    if header_support
      @headers.delete_at(pos)
    end
    @rows.map {|row| row.delete_at(pos) }
  end
end
```

Finally, we want to be able to transform the values of a column one by one. For example, we would like to "age" everyone in our table by five years.

```
test "can run a transformation on a column which changes its content" do
  expected_ages = @table.column("age").map {|a| a+= 5 }
  @table.transform_columns("age") do |col|
    col += 5
  end
  assert_equal expected_ages, @table.column("age")
end
```

In this case we are yielding each cell in the column to the block defined by the user of the API.

```
class Table

  def transform_columns(pos, &block)
    pos = column_index(pos)
    @rows.each do |row|
      row[pos] = yield row[pos]
    end
  end
end
```

Last but not least, we need to filter out columns that don't meet a particular condition. As a somewhat contrived example, let's select only those items for which the column sum is lower than 10:

```
test "can select columns by some criteria" do
  table = Table.new([["item1", "item2", "item3", "item4", "item5"],
                     [3, 7, 4, 9, 2],
                     [4, 8, 2, 3, 1],
                     [0, 9, 3, 4, 6]
                     ],
                    :headers => true)

  table.select_columns do |col|
    col.inject(0, &:+) < 10
  end

  assert_equal 3, table.rows[0].length
  assert_equal(["item1", "item3", "item5"], table.headers)
end
```

Given our current, row-biased approach there is simply no easy way to do this. We have to temporarily create each column and check it against the condition defined in the block sent to the select_columns() method. Since we expect the block to return true or false, we can use that to determine whether we should keep or delete the column. The execution of the latter we will delegate to the existing delete_column() method.

```
class Table

  def max_y
    rows[0].length
  end

  def select_columns
    selected = []

    (0...max_y).each do |i|
      col = @rows.map { |row| row[i] }
      selected.unshift(i) unless yield col
    end

    selected.each do |pos|
      delete_column(pos)
    end
  end
end
```

We've completed all the requirements for the assignment and you may find this simple implementation complete with tests here.

This implementation is somewhat similar to that submitted by the majority of students in this session but it's not without its weaknesses. These drawbacks are the subject of the next chapter.

CHAPTER 5

Handling Exceptions and Bad Input

In a perfect world, we could probably be done at this point. However, we have so far completely ignored an important aspect, namely dealing with potential errors.

We start with mimicking errors that could occur when accessing table data if non-existent rows or columns are being referenced. We can call them "out of bounds" errors. Consider the following scenario:

```
>> table = Table.new(@data)
>> table[99, 99]
=> NoMethodError: undefined method `[]' for nil:NilClass
```

As a reminder, here is the code that is presently unprepared to handle such a test case:

```
class Table

  def [](row, col)
    col = column_index(col)
    rows[row][col]
  end

end
```

If the referenced row (rows[row]) is out of bounds, then nil gets returned. This is in itself something that you might want to handle with either an error or a warning (or else accept that nil gets returned in such cases). If we attempt to retrieve a particular column in that non-existent row, however, we have an unhandled NoMethodError on our hands.

A simple fix could involve checking the requested index before attempting to retrieve the cell. The code below also demonstrates how one might set up and use custom error classes.

```
class Table

  NoRowError = Class.new(StandardError)

  def [](row, col)
    check_row_index(row)
    col = column_index(col)
    rows[row][col]
  end

  def max_x
    @rows.length
  end

  def check_row_index(pos)
    unless (-max_x..max_x).include?(pos)
      raise NoRowError, "The row index is out of range"
    end
  end
end
```

This leaves Table#[] with mixed behavior when a row or column is out of bounds. For a row it will return an error and for a column it will return nil. Dealing with columns means that we need to take column names into account. In the next section, we take care of making the Table#[] behavior consistent.

Column names

Since columns can be referenced either by their index or by their name, we could encounter nonexistent column names, as in:

```
>> table[2, "bad_column"]
=> TypeError: can't convert String into Integer
```

Or, what if we were to insert or rename a column using a name that is already taken:

```
>> table.add_column(["age", 10, 11, 12, 13, 14])
>> table.headers
=> ["name", "age", "occupation", "age"]
>> table[2, "age"]
=> 45

>> table.rename_column("name", "age")
>> table.headers
=> ["age", "age", "occupation", "age"]
>> table[2, "age"]
=> George
```

To handle the case of referencing a column that doesn't exist, we can employ a similar strategy as the one we used for the rows. We can intercept the method that determines the numeric index of the column based on it being either a string or an integer.

To avoid duplicate column names, we could just opt to check the @headers array when appropriate. Ruby, however, provides a more elegant way to deal with this problem. Changing the internal representation of @headers from an array to a hash facilitates avoiding duplicate names, since hashes don't allow for duplicate keys.

Here then is the code to guard against both potential sources of error:

```
class Table

  def initialize(data = [], options = {})
    @header_support = options[:headers]
    set_headers(data.shift) if @header_support
    @rows = data
  end

  def set_headers(header_names)
    @headers = {}
    header_names.each_with_index do |item, index|
      @headers[item] = index
    end
  end

  def max_y
    rows[0].length
  end

  def column_index(pos)
    pos = @headers[pos] || pos
    check_column_index(pos)
    return pos
  end

  def rename_column(old_name, new_name)
    check_header_names(new_name)
    @headers[new_name] = @headers[old_name]
    @headers.delete(old_name)
  end

  def add_column(col, pos=nil)
    i = pos.nil? ? rows.first.length : pos

    if header_support
      header_name = col.shift
      check_header_names(header_name)
      @headers.each { |k,v| @headers[k] += 1 if v >= i }
      @headers[header_name]  = i
    end

    @rows.each do |row|
      row.insert(i, col.shift)
    end
  end
```

```
    def delete_column(pos)
      pos = column_index(pos)

      if header_support
        header_name = @headers.key(pos)
        @headers.each { |k,v| @headers[k] -= 1 if v > pos }
        @headers.delete(header_name)
      end

      @rows.map {|row| row.delete_at(pos) }
    end

  private

    def check_header_names(name)
      raise ArgumentError, "Name already taken" if @headers[name]
    end

    def check_column_index(pos)
      unless (-max_y..max_y).include?(pos)
        raise NoColumnError,
              "The column does not exist or the index is out of range"
      end
    end

  end
```

Handling bad input

In a table we expect all rows to have the same length and also all columns to have the same number of elements. Now, we turn our attention to input that does not conform to this basic expectation.

Generally speaking, there are two main courses of action in dealing with bad input: either reject it by raising an error or adjust the input so as to bring it back into the fold.

One way to change rows and columns so that they have the expected length might be to pad the short ones and truncate the ones that are too long. When it comes to rows, we might get away with this strategy, but padding short columns could have some undesired side effects. In tables with header support, for instance, we assume the first element to be the header name. Should the header not be included in the new column, then the padding approach will cause a missing header to "fail silently": the first element will be appointed as the header, while the last element will be supplied by padding.

Truncating rows/columns that are too long leaves one with an uneasy feeling, even if the truncation is accompanied by a warning.

Considering all these issues with truncation/padding, we will instead raise an exceptions upon encountering rows or columns with unexpected lengths.

```
class Table

  def initialize(data = [], options = {})
    data.each do |row|
      check_length(row, data.first.length, "Inconsistent rows length")
    end

    @header_support = options[:headers]
    set_headers(data.shift) if @header_support

    @rows = data
  end

  def add_row(new_row, pos=nil)
    check_length(new_row, max_y, "Inconsistent row length") unless @rows.empty?

    i = pos.nil? ? rows.length : pos
    rows.insert(i, new_row)
  end


  def add_column(col, pos=nil)
    check_length(col, max_x+1, "Inconsistent column length")

    i = pos.nil? ? rows.first.length : pos

    if header_support
      header_name = col.shift
      check_header_names(header_name)
      @headers.each { |k,v| @headers[k] += 1 if v >= i }
      @headers[header_name]  = i
    end

    @rows.each do |row|
      row.insert(i, col.shift)
    end
  end

  private

  def check_length(data, expected, msg="Input length is inconsistent")
    raise(ArgumentError, msg) unless data.length == expected
  end

end
```

Data corruption

The user of our API can at any time inadvertently corrupt the table internals by tinkering with the provided data. Conversely, changes initiated through the table will also affect the provided data.

The source of this issue lies in the fact that, in Ruby, variables hold references to objects, rather than being the objects themselves. As such, if the same object is referenced by more than one variable, we need to keep in mind that changes made to the object will be exposed regardless of which variable we chose to reference the object by. Here's a simple example to illustrate the point:

```
>> person1 = "Tim"
>> person2 = person1
>> person1[0] = 'J'

>> puts "person1 is #{person1}"
=> person1 is Jim
>> puts "person2 is #{person2}"
=> person2 is Jim
```

So, in our case, when we initialize a new Table we are setting the @rows variable to reference the same two-dimensional array as @data does. Here you can see that when we remove the first array as headers, we are also altering @data:

```
>> @data = [["name",  "age", "occupation"],
            ["Tom", 32,"engineer"],
            ["Beth", 12,"student"],
            ["George", 45,"photographer"],
            ["Laura", 23, "aviator"],
            ["Marilyn", 84, "retiree"]]
>> table = Table.new(@data, :headers => true)
>> @data
=> [["Tom", 32,"engineer"],
    ["Beth", 12,"student"],
    ["George", 45,"photographer"],
    ["Laura", 23, "aviator"],
    ["Marilyn", 84, "retiree"]]
```

We can change the initialize method in order to avoid damaging the data provided by the user:

```
class Table

  def initialize(data = [], options = {})
    # code omitted
    if options[:headers]
      @headers = data[0]
      @rows    = data[1..-1]
    else
      @headers = []
      @rows    = data
    end
  end

end
```

While this approach doesn't change the provided data, data corruption can still happen. Here is another example of how altering @data will be reflected when we access the Table instance:

```
>> table = Table.new(@data)
>> @data[2][2] = "king of the world"
>> table[2, 2]
=> "king of the world"
```

Let's say that to prevent accidental data corruption we want to somehow duplicate the provided data when we initialize the table. We might try to use Object#dup to accomplish this.

```
class Table

  def initialize(data = [], options = {})
    # code omitted
    @rows = data.dup
  end

end
```

Simple, isn't it? Except that this doesn't solve our problem. The behavior from the previous example remains.

```
>> table = Table.new(@data)
>> @data[2][2] = "king of the world"
>> table[2, 2]
=> "king of the world"
```

Let's examine the dup() method a little closer:

From the Pickaxe book: dup()

dup()

Produces a shallow copy of obj — the instance variables of obj are copied, but not the objects they reference. dup copies the tainted state of obj. See also the discussion under Object#clone.

In general, dup duplicates just the state of an object, while clone also copies the state, any associated singleton class, and any internal flags (such as whether the object is frozen). The taint status is copied by both dup and clone.

So if we dup() the data that's passed to our initialize method before we assign it to Table#rows, the outer array that @rows points to is indeed a different object from the outer array held in @data. However, Table#rows and @data are still both referencing the same internal arrays, meaning the actual rows. To elucidate:

```
>> table = Table.new(@data)
>> table.rows.object_id    # note that the output is run specific
=> 2151823700
>> @data.object_id
=> 2151823780

>> table.row(0).object_id
=> 2151825160
>> @data[0].object_id
=> 2151825160
```

The only really reliable way to create a brand new object when we assign it to another variable is marshaling.

From the Pickaxe book: Marshaling

Marshaling is the ability to serialize objects, letting you store them somewhere and reconstitute them when needed.

[.....]

Saving an object and some or all of its components is done using the method Marshal.dump. Typically, you will dump an entire object tree starting with some given object. Later, you can reconstitute the object using Marshal.load.

```
class Table

  def initialize(data = [], options = {})
    data.each do |row|
      check_length(row, data.first.length, "Inconsistent rows length")
    end

    @header_support = options[:headers]

    @rows = Marshal.load(Marshal.dump(data))
    set_headers(@rows.shift) if @header_support
  end

end
```

We use Marshal.dump to output a string representation of the object tree referenced by "data" and then use this string as the input for Marshal.load which reconstructs the full object tree. Now there is no cross-reference between @data and the table @rows. We can also say that the Table.new method is side-effect free:

```
>> table = Table.new(@data, :headers => true)
>> @data[2][2] = "king of the world"
>> table[2, 2]
=> "photographer"
>> @data[0]
=> ["name", "age", "occupation"]
```

Unfortunately this method is also not without drawbacks. Apart from the time that it takes to marshall and unmarshall the seed data, we are temporarily storing two copies of the same nested array.

This technique can also be applied when adding rows and columns. You may find this improved implementation complete with tests here.

While this implementation is more robust than the one from the last chapter there's still room for improvement. In the following chapters we'll take a look at some interesting student submissions.

CHAPTER 6

Submission by Lucas Florio

Although it works and has a fairly clean API, the solution presented above is by no means perfect. We have already mentioned the somewhat awkward access to columns. Another area of concern is that all the code for the entire table lives in a single class - a potential violation of the Single Responsibility Principle.

Single Responsibility Principle (SRP)

A "responsibility", in this sense of the word, is a description of what a particular piece of software is designed to accomplish. Ideally, different responsibilities should be handled by different parts of the software. This more often than not boils down to a class having a single responsibility.

Without this well-considered separation, the responsibilities that a class is tasked with become coupled. This is an undesirable scenario as it necessitates the extra step of taking into account how the change to one responsibility affects another.

Therefore, by organizing the code in small, tightly focused sections, we make it both more readable and easier to maintain.

The Table class has some separation built-in by sporting short, focused methods. While this is a first step in the right direction, there are other ways to break out functionality into separate, non-overlapping classes. This is, however, not as easy as it sounds. Obvious attempts might include separating classes for rows and columns, but this will prove difficult because of the tight coupling between them. The coupling in question derives from the fact that they are essentially just different perspectives on the same data set.

Now we're going to consider a solution by Lucas Florio which manages to treat rows and columns as wholly separate objects, but without duplicatingthe data set. The trick is that they don't hold the actual data values, but delegate that responsibility to a Cell class.

The complete code can be found at: https://github.com/lucasefe/s1-final.

Rows, Columns and Cells

How exactly is this behavior accomplished? In Lucas' own words:

> *The object Bricks::Table stores two Bricks::Index objects. One for vertical access and one for horizontal allowing data access by column or row. For each cell that gets added to the table, an Bricks::Cell object is created. This Cell is also added to both indexes, so that when you retrieve data by column or row, you always get what you want.*
>
> *Since the cells are created and then added to the index, there is no data duplication. If you modify a table cell, as in the following example, you are only modifying one object, the cell itself, but not the indexes.*

Take a look at the initialize method of the Bricks::Table object:

```
module Bricks
  class Table
    # code omitted

    attr_reader :rows, :columns
    def initialize(*args)
      @rows    ||= Bricks::Index.new
      @columns ||= Bricks::Index.new

      # code to extract options and to set column headers omitted
      data = args.first.dup   # to avoid alter original data

      data.each_with_index do |row, index|
        add_row row
      end
    end

    def add_row(new_row)
      add_and_update(new_row, rows, columns)
    end

    # code omitted
  end
end
```

As you can see, the individual rows are immediately sent on to a method called add_and_update() that also manages to account for columns. A little further code diving reveals that the add_and_update() method can handle both adding rows or adding columns just by switching out which of the two is considered to be the primary or the secondary target of the operation. At the same time, the common Cell objects are being created.

```
module Bricks
  class Table
    # code omitted
    private

    def add_and_update(array, primary, secondary)
      new_array = []
      with_cells_in(array) do |cell, index|
        new_array << cell
        secondary[index] ||= []
        secondary[index] << cell
      end
      primary << new_array
    end

    def with_cells_in(array)
      array.each_with_index do |value, index|
        yield(build_cell(value), index)
      end
    end

    def build_cell(value)
      value.kind_of?(Bricks::Cell) ? value : Bricks::Cell.new(value)
    end
  end
end
```

Similarly, all pertinent operations (adding, inserting and deleting) are mirrored in both storage options and handled by the same private helper method, i.e. delete_row_at() and delete_column_at() both forward calls to delete_at_and_update().

Let's play around with this on irb to demonstrate some of the advantages of this approach:

```
>> data = [[1,2,3], [4,5,6], [7,8,9]]
>> table = Bricks::Table.new(data)
=> #<Bricks::Table:0x191388 @options={}, @rows=[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
@columns=[[1, 4, 7], [2, 5, 8], [3, 6, 9]]>
```

The mirroring of rows and columns affords very easy access to columns by simple array methods, such as:

```
>> table.columns[0]
=> [2,4,7]
```

Also, changing the Cell value through rows will automatically change the value in the columns and vice versa:

```
>> table.rows[0].map {|i| i.value = i.value*2}
=> [2, 4, 6]
>> table.rows
=> [[2, 4, 6], [4, 5, 6], [7, 8, 9]]
>> table.columns
=> [[2, 4, 7], [4, 5, 8], [6, 6, 9]]
```

To better understand how the Bricks::Index class is constructed, consider the following snippet from its initializer:

```
module Bricks
  class Index < DelegateClass(Array)

    def initialize
      super(Array.new)
    end

    # code omitted
  end
end
```

The most obvious feature is the use of DelegateClass(Array). Simply speaking, this lets Bricks::Index instances delegate methods not defined by the class itself to an internal array object. For a detailed explanation on how this works, see the following aside.

The DelegateClass method

Under the hood, DelegateClass is a method call and just like the class keyword returns a class object.

The class returned by DelegateClass(Array) knows to delegate all methods defined by Array to an internal array object that should be defined in the initialize method. This object is created in the constructor of Bricks::Index and passed to the superclass via super(Array.new).

Bricks::Index can override array methods and also implement additional methods to define different behaviors.

Note that even though we seem to be using inheritance, we don't actually "inherit" directly from Array. Rather than stating that Bricks::Index is a special form of Array, it would be more precise to say that Bricks::Index **is a** Delegator and **has an** Array.

Reading vs. Writing Operations

One of the major gripes we had with the solution described in the earlier chapters was the difficult access to columns. Every time we wished to retrieve a column, we needed to iterate through the rows and fetch the row item corresponding to the column index in question.

Lucas found a neat workaround by storing the columns separately (but without duplicating the data set). However, this also comes with a cost of its own. Any operation that modifies the table - adding, inserting, deleting, filtering - is process intensive in that rows and columns need to be manually synched to each other effectively producing a "writing overhead".

One potential drawback to this approach is that it requires the creation of x*y Cell objects, where x is the number of rows and y the number of columns. This would lead to a high memory load when dealing with very large tables.

Data Corruption

If you study the code closely, you would notice that filtering rows using the Bricks::Index#select! method leaves the columns unchanged. This unimplemented feature could easily be introduced by devising a method to safely execute the select!() operation that makes the appropriate changes in the alternate data view. However, this omission points to a bigger issue altogether. Since the Brick::Index class exposes all the methods defined by Array, the user could cause accidental data corruption, for example by bypassing the intended API methods and working with array methods directly.

CHAPTER 7

Other Interesting Submissions

This chapter deals with two more alumni submissions that are noteworthy for different reasons.

Eric Gjertsen tried to optimize memory usage, which is a valid concern when dealing with large data sets. He also had a unique idea regarding data storage.

Wojciech Piekutowski attempted to cleanly separate concerns in an object oriented way. We will discuss the particulars of his solution in a bit.

Eric Gjertsen's Submission

The complete code can be found at: https://github.com/ericgj/s1-final.

From the data storage point of view, we've considered the options of storing an array of rows as provided by the user and Lucas Florio's approach which introduces a collection of Cell objects "indexed" by rows and columns. Turns out that there are even more ways of representing the data.

Eric's idea was to store the data as a single flattened array instead of as a two-dimensional array. The data is not contained in traditional "rows" or "columns" which, in the previous two solutions, were represented by the internal arrays.

If you stop and think about the implications of using a flattened array for a moment, you'll realize that quite a bit would have to change in the way we think about rows and columns.

A "row" now becomes merely a virtual sequence in the array. It can be sliced out of the main array by calculating the indexes at which it begins and ends - calculations which are based on the row index and length. Similarly, a column would need to be assembled based on its calculated indexes throughout the main array.

Here are some of Eric's thoughts on this way of storing the data:

> *The path I took was initially motivated by two related issues of encapsulation. First of all, you're going to be doing operations on both rows and columns which have some state in common with each other and with the table as a whole. If rows and columns are basically arrays, you immediately run into the problem that state can't really be shared between them. If you slice columns out of rows, how are you going to be able to "Run a transformation on a column which changes its content based on the return value of a block"? You'd be changing the sliced array elements but not the row array elements.*

Of course this could be addressed by making the appropriate updates in the other storage option - something we have seen in the two previous solutions -, but Eric was opposed to what he called "*a lot of messy double changes*". He continues:

> *That was one problem. The second issue was that even as it looked like you need row, column, and column header arrays (or enumerables of some kind) to manipulate, you don't want to actually expose these as arrays with direct access to table data, because then you could easily corrupt the table.*

He also mentions a third and relatively serious concern:

> *The other thing running through my mind when I started thinking about the problem was that the memory overhead should be kept as low as possible. I know the Ruby community tends to downplay this kind of concern upfront, but I think it's legitimate here given that we are already loading an arbitrary sized file into memory. That is -- you have n rows * m cols objects before you even talk about a Table class and whatever else you need.*

Mulling over potential memory issues, he came up with a way to "lazily load" rows and columns. Check out his ScopedCollection class to get a better picture:

> *Row and Column collections are 'lazy-loaded' arrays of rows and columns. That is, you can specify a range of rows/columns and conditions, and the array isn't actually populated until you enumerate it in some way -- e.g. by calling table.rows.map.*

In retrospect, though, he voiced some doubts regarding his initial choice of storage, the flattened array.

> *Storing everything in a single array made things like inserting and deleting a column quite complicated. I found myself wanting some variant of Array#zip that inserts from one array into every nth element of another. In the end, my col insert and delete methods end up rebuilding the entire table - not very efficient. On the other hand, having one array saves memory compared to an array of arrays, particularly for large numbers of rows.*

This is how he would change things for future incarnations of the Table class:

> *It would be relatively easy to change to storing the data as an array of arrays. I would just have to re-implement the row/cell access and manipulation methods in Table -- there would be no changes needed to Row and Column and Cell classes. And the problem of sharing references between row and column arrays would not come up, since cells would always access the same underlying element, whether they belong to a column or to a row.*

Wojciech Piekutowski's (W.P.) submission

The complete code can be found at: https://github.com/wpiekutowski/s1-final.

What makes W.P.'s solution interesting is the way the code is organized. He identified five classes that together make up the functionality necessary to have a working Ruby table implementation. Besides the Table class, there are classes representing the collection of rows and columns (Table::RowsProxy and Table::ColumnsProxy), as well as a class for an individual row and an individual column (Table::Row and Table::Column).

Here is an overview of the responsibility and and features of each:

Table:

- Accepts 2-dimensional arrays as input and assigns that data to an @matrix instance variable
- Has a rows() and columns() method that return the respective proxy objects
- doesn't actually manipulate the data in any way

Here's a snippet of the Table class:

```
class Table
  # code omitted

  # Returns ColumnsProxy object capable of columns operations
  def columns
    @columns_proxy ||= Table::ColumnsProxy.new(@matrix, self)
  end

  # Returns RowsProxy object capable of rows operations
  def rows
    @rows_proxy ||= Table::RowsProxy.new(@matrix, self)
  end

end
```

Table::RowsProxy and Table::ColumnsProxy :

- have an instance of @matrix, which holds the data and @table, their "parent" object
- include the Enumerable module
- take care of inserting and appending a new row or column
- index method (#[]) returns Table::Row or Table::Column object

Table::Row and Table::Column :

- references the Table's @matrix and the respective collection (rows or columns) it belongs to in @proxy
- keeps track of its own @index within the collection
- takes care of operations within an individual row or column, such as:
    - mapping the row or column values
    - deleting the row or column
    - accessing row or column elements (cells)
    - renaming or accessing the column name

You might have noticed that all classes reference and manipulate the same instance variable @matrix, which is essentially a two-dimensional array representing the rows. If you peel back the layers of class separation, then this solution is very similar to the first one we walked through.

View for example how you would change the values of all cells in a particular column:

This is the API of W.P.'s solution:

```
table.columns['PROCEDURE_DATE'].map! do |date|
  parse_date(date).to_s
end
```

And this is the equivalent method class in the first, single-class solution:

```
table.transform_columns("PROCEDURE_DATE") do |col|
  parse_date(col).to_s
end
```

Here is the transform_columns() method from first solution:

```
class Table

  def transform_columns(pos, &block)
    pos = column_index(pos)
    @rows.each do |row|
      row[pos] = yield row[pos]
    end
  end

end
```

And this is how W.P.'s classes work together to accomplish the same:

```
class Table

  def columns
    @columns_proxy ||= Table::ColumnsProxy.new(@matrix, self)
  end

end

class Table::ColumsProxy

  def [](column)
    index = position_to_index(column)
    return unless index

    Table::Column.new(@matrix, self, index)
  end

end

class Table::Column

  def map!
    @matrix.each do |row|
      row[@index] = yield(row[@index])
    end
  end

end
```

The solutions both iterate through the row data in order to yield the element at the column index to an arbitrary block. However, the API that W.P.'s solution exposes is in some ways cleaner and more attractive. To be able to write table.columns["some_col"].map!(&block) is much more familiar than table.transform_columns("some_col", &block).

Wojciech describes how he developed this API:

> *I was experimenting with an ideal API - I wanted it to read easily and be well suited for this kind of table operations. That gave me a good understanding on what I want to achieve and some initial ideas about implementation details. It was also a top-level guide about what I should test during the development. Next step was writing some initial specs and initial code. After that I followed red-green-refactor route. I had my general architecture done at the beginning, but as the project went on I did some changes - for example how different parts of the code should communicate or what data really needs to be shared.*

A significant feature of the code is the including of the Enumerable module in the proxy objects. By overriding the each() method in the classes that include Enumerable, we essentially (re)define what we consider to be the unit that we would like to be handled by iterator methods. This is an incredibly powerful feature, since many other methods, that rely on the particular implementation of each() in the background (e.g. select(), map() and inject()), will automatically work as expected.

From the Practicing Ruby Newsletter

(....), there is surprising power in having a primitive built into your programming language which trivializes the implementation of the Template Method design pattern. If you look at Ruby's Enumerable module and the powerful features it offers, you might think it would be a much more complicated example to study. But it too hinges on Template Method and requires only an each() method to give you all sorts of complex functionality including things like select(), map(), and inject(). If you haven't tried it before, you should certainly try to roll your own Enumerable module to get a sense of just how useful mixins can be.

Practicing Ruby Issue #9: Uses For Modules, Part 2 of 4 by Gregory Brown

CHAPTER 8

Conclusion

This problem of representing a table structure in Ruby which, at first glance, seemed straightforward turned out to be quite tricky in practice. This is borne out by the fact that many of the students ended up taking very different approaches.

So why did we see this variety in the solutions? One factor is the native complexity of the problem itself. It requires finding a way of treating rows and columns as separate entities on the one hand, and on the other, maintaining their status as simply different views of the same data set.

Although we have looked at what we can say are good and better solutions, it is evident that there is no absolutely "right" solution.

The main objective here in this exercise, though, was not to come up with "the" perfect solution - in ruby, as in other programming languages, there are often myriad accomplished ways to solve a particular software issue - but rather to develop the ability to think out problems comprehensively by learning to ask the right questions.

The great benefit of the environment that has been created here at Mendicant University is that students have ample opportunity to tackle difficult and multifaceted problems such as this one on their own. This independent self-paced study is essential in order to develop a personal feel for problem solving in Ruby. However, there is also much to be gained by considering how one's peers approach similar types of challenges.

We hope that you have enjoyed this paper as much as we've had writing it.

Special Thanks

Special thanks to our lead teacher, Gregory Brown for being the driving force behind Mendicant University and to the MU alumni Lucas Florio, Wojciech Piekutowski and Eric Gjertsen (now part of the MU staff) for their helpful

feedback and for allowing us to feature their solutions. Eric also deserves credit for helping us with some of the chapter 5 snippets.

Getting Involved

If you want to get involved in Mendicant University, there is no shortage of options:

The official website at http://university.rubymendicant.com is the first place you should look if you'd like to know more about us. Keep an eye on the news page to learn about the latest developments.

On the IRC channel #rmu at Freenode you can chat with the students and staff. The channel is also the place for our monthly Q&A session with special guests.

On Twitter you can start following http://twitter.com/mendicant_news to receive firsthand updates about new activities.

And if you'd like to join us, keep checking news page where the next round of admissions will be announced.