Exercise 8:
Building HTML Documents

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering Design & Construction
WS 2016/17 - Dr. Michael Eichberg, M.Sc. Matthias Eichholz

Although this exercise is not graded, it is highly recommended to also do them on your own. Just looking at a solution is much easier in comparison to actually coming up with it. Support can be found in the office hours of our tutors and in the forum at `https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=234`.

Building HTML Documents

In this exercise, you will develop a small library to build simple HTML documents. You will use different design patterns and build documents that render images either as embedded PNGs, as linked PNGs, or in HTML 5 canvas elements. In the project template, you will find partial implementations (they will not compile) and a test application that creates a few test pages. Your implementation must adhere to the interface used in the given `HTMLTests` object, i.e., once you have implemented your library, `HTMLTests` will compile and open a few generated pages.

Task 1 Source Code Builder

Eventually we want to emit correctly indented HTML and Javascript code. Implement a builder `SourceCodeBuilder` that helps you do that. You will find a template in the project that you simply need to complete.
The builder maintains a current level of indentation, which can be changed with methods `indent()` and `unindent()`. Note that the builder needs to recognise when a new line is started in order to correctly indent the code. A new line is started either by a call to `newline()` or by appending a string ending in `'\n'`.
Most methods in `SourceCodeBuilder` have return type **`this.type`**, which simply means you have to return **`this`** at the end of each method. This will allow you to do method call chaining such as in:

```
val age = 99
val b = new SourceCodeBuilder("␣␣")
b += "Hi,␣I␣am␣" += age += "years␣old."
```

Use a `StringBuilder` to implement `SourceCodeBuilder`, which is more efficient than composing strings.

Task 2 Document Builder

Implement a builder that lets clients gradually build HTML documents. Your implementation needs to be able to emit HTML 5 as well as HTML 4 documents. Moreover, you should be able to generate single HTML documents and documents that are split across multiple files.
Use the bridge pattern to implement these two axes of variability. When using the bridge pattern, one has two class hierarchies, the abstraction and the implementation hierarchy. In this case, the implementation side should care about emitting code for different HTML versions. The abstraction side cares about how to logically arrange the HTML document. You will find the base traits of both sides in `DocumentsBuilders` and `HTMLBuilders`.
For this task, you need to do two things:

- Create two subclasses for `HTMLBuilder` as indicated in the project template. The only difference between them for now is that they emit different DOCTYPE declarations (DTD). For HTML 5 it should simply be `<!DOCTYPE html>`. See `http://www.w3schools.com/tags/tag_doctype.asp` for the HTML 4.01 Strict DTD. Use your `SourceCodeBuilder` for the implementation side.
- Create two subclasses for `DocumentBuilder` as indicated in the project template. The `SimpleDocumentBuilder` will emit HTML is a straightforward way, eventually creating a simple string containing a valid HTML document including a DTD, head and body elements. The `PaginatedDocumentBuilder` will create a new HTML document each time a client adds a `<h1>` headline. The new document will again be a valid HTML document including a DTD, head and body elements, where the body will contain the headline and everything that follows until the next `<h1>` headline. The result of `PaginatedDocumentBuilder` should be a `Seq[String]` and it should use a (fresh) `SimpleDocumentBuilder` to build each page.

Task 3 Image Hierarchy

We now want to be able to add images to HTML documents. In order to do so, we first need a representation of them. Since images can be either embedded into HTML (in Data URIs using Base64 encoding[1]), or linked via a URL, we want to represent these two alternatives. It is important to notice that in the latter case, we don't need to load the image data into memory.
Therefore, develop a minimal image hierarchy to represent pixel images. In `ex10.Images`, you will find the base trait of the hierarchy, which is as follows (excluding comments):

```
trait Image {
  def width: Int
  def height: Int
  def toPNG: Array[Byte]
}
```

The only way to obtain image data is via method `toPNG`, which returns the image in the PNG format. Create the following two subclasses:

- Use the adapter pattern to adapt a `java.awt.image.BufferedImage` to your `Image` trait. You can obtain a PNG byte array from a `BufferedImage image` as follows:

  ```
  val out = new ByteArrayOutputStream
  ImageIO.write(image, "png", out)
  out.toByteArray()
  ```

- Use the proxy pattern to represent an image at a given URL. The proxy gets the width and height as constructor arguments but the image data is loaded from the URL (using `ImageIO.read()`) only when needed. You will find utility method `Image.resizeImage()` to scale a given image.

Create factory methods in the companion object of the `Image` trait to create images from a `java.io.URL` (using your proxy) and a `java.io.File` (using your adapter and `ImageIO.read()`).

Task 4 HTML Images

Add a method `image(ing: Image)` to `DocumentsBuilder` and its implementations which adds an image element to the document under construction. It should add a linked image as in `<img src="http://upload.wikimedia.org/wikipedia/en/7/78/Design_Patterns_cover.jpg">` if the given image is an instance of your image proxy. Otherwise, it should use a Data URI with base64 encoding. You will find a utility method in the project template to convert PNG byte arrays to base64 strings.

Task 5 Graphics

We now want to be able to add procedurally rendered images into an HTML document, i.e., draw lines and rectangles into an HTML 5 canvas element or fall back to pre rendered PNG images for HTML 4. For this purpose, we need a canvas to draw on. This canvas is in fact again an instance of the builder pattern.
Your tasks are:

- You will find the base class `Canvas` of the canvas builder in the project template. Implement two subclasses, `HTML5Canvas` and `PNGCanvas`. Class `HTML5Canvas` should eventually append HTML as follows:

  ```
  <canvas id="canvas1" width="300" height="200"></canvas>
  <script type="text/javascript">
    function draw_canvas1() {
      var canvas = document.getElementById('canvas1');
      if (canvas.getContext) {
        var c = canvas.getContext('2d');
        c.fillStyle = 'rgba(255,0,0,0.5019607843137255)';
        c.fillRect(10,10,50,50);
        c.fillStyle = 'rgba(0,0,255,0.5019607843137255)';
        c.fillRect(30,30,50,50);
      };
    };
    draw_canvas1();
  </script>
  ```

[1] See the HTML example on `http://en.wikipedia.org/wiki/Data_URI_scheme#Examples`

Class `PNGCanvas` should render the image into an in-memory `BufferedImage` and add a Data URI with base64 encoding.

- Add a method `canvas(width: Int, height: Int)(c: Canvas => Unit)` to your document builders. It accepts a width and height of the canvas and a function that takes a handle to the canvas to construct. This function can then draw shapes to the canvas, e.g.,:

```
builder.canvas(400, 300) { c =>
  c.setColor(Color.Blue)
  c.fillRect(50, 50, 100, 100)
  c.setColor(Color.White)
  c.drawLine(10, 10, 140, 140)
}
```

- For the previous point, you also need to modify your HTML builders to choose the right canvas instance.

The code template already contains a few utility methods and partial implementations for this task.