Assignment 1: Printing a Calendar in Scala

Assigned: March 31
Due: April 14

For this assignment we are interested in printing a calendar. Specifically, we want to print an overview of a given month that shows which date falls on which day of the week. For example, the month of April 2011 should be printed as follows:

```
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

For this assignment we want to flex our functional programming muscles, **therefore vars, loops, and mutable data structures are not allowed**. Instead, try using pattern matching and recursion. Also, Scala has a number of convenience methods defined on `List` that will likely prove useful in writing concise implementations. More points for better style.

Leap years, the first of January, etc.:

In order to print a monthly overview, we first have to determine the weekday of the first of the month. We provide the following function definitions to simply this task:

```
/** The weekday of January 1st in year y, represented
* as an Int. 0 is Sunday, 1 is Monday etc. */
def firstOfJan(y: Int): Int = {
  val x = y - 1
  (365*x + x/4 - x/100 + x/400 + 1) % 7
}

def isLeapYear(y: Int) =
  if (y % 100 == 0) (y % 400 == 0) else (y % 4 == 0)

def mlengths(y: Int): List[Int] = {
  val feb = if (isLeapYear(y)) 29 else 28
  List(31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
}
```

With the help of these functions, define a function `firstDay` that calculates the weekday of the first day of a given month.

```
def firstDay(month: Int, year: Int): Int = ...
```

Making a picture:

Picturing data with a non-trivial layout can be tricky. Therefore, we want to use a compositional approach where larger, more complex pictures are composed of smaller, simpler pictures. In our design, pictures are represented as instances of the `Picture` case class (recall that, among other features, defining a case class allows us to extract the constructor arguments during pattern matching).

```
case class Picture(height: Int, width: Int, pxx: List[List[Char]]) {
  def showIt: String { ... }
}
```

As we can see, a picture has a height and width, along with contents `pxx`, which is character data represented as a list of rows, where each row is a list of characters. The `showIt` method is provided for you and turns the picture into a formatted `String`.

The following function `pixel` creates a simple picture of height and width `1` that contains a given character.

```
def pixel(c: Char) = Picture(1, 1, List(List(c)))
```

From pictures as simple as that, we want to compose larger ones using composition operators.

1. Define a method `above` for class `Picture` that returns a new picture where the argument picture is placed below `this`:

   ```
   case class Picture(...) {
     def above(q: Picture): Picture = ...
   }
   ```

   For instance, the following code

   ```
   println((pixel('a') above pixel('b')).showIt)
   ```

   should print

   a
   b

   Give an error message (using the predefined `error` function) when the pictures do not have the same width.

2. Define a method `beside` for class `Picture` that returns a new picture where the argument picture is placed on the right side of `this`:

   ```
   case class Picture(...) {
     def beside(q: Picture): Picture = ...
   }
   ```

   Give an error message when the pictures do not have the same height.

3. Define functions `stack` and `spread` that arrange a list of pictures above and beside each other, respectively, producing a single resulting picture. For `stack`, the picture at the head of the

argument list should be the topmost picture in the result. For `spread`, the head of the list should be the leftmost picture in the result.

```
def stack(pics: List[Picture]): Picture = ...
def spread(pics: List[Picture]): Picture = ...
```

4. Define a function `tile` that arranges a list of rows of pictures in a rectangular way using the `stack` and `spread` functions:

   ```
   def tile(pxx: List[List[Picture]]): Picture = ...
   ```

5. Define a function `rightJustify` that takes a width `w` and a list of characters, and produces a picture of height `1` and width `w` where the given characters are justified on the right border:

   ```
   def rightJustify(w: Int)(chars: List[Char]): Picture = ...
   ```

   Give an error message if `chars.length > w`.

6. Define a function `group` that splits a list into sublists. The function takes an integer argument that indicates the split indices (e.g., split every 7 elements). We intend to use this function to split a list representing a whole month into a list of weeks. Note that this function is parameterized which means that it can be used with lists of any element type.

   ```
   def group[T](n: Int, xs: List[T]): List[List[T]] = ...
   ```

7. Define a function `dayPics` that takes the weekday number of the first day of a month and the number of days in that month and produces a list of 42 pictures. In this list the first `d` pictures are empty (i.e., the character data is a list of spaces) if the number of the first day is `d` (`d==0`: Sunday, `d==1`: Monday, etc.). The trailing pictures that correspond to the days of the next month should be empty as well. Using this function, we can produce a picture of a calendar by grouping and tiling the result of `dayPics`.

   ```
   def dayPics(d: Int, s: Int): List[Picture] = ...
   ```

8. Using the functions defined in the previous steps, define a function `calendar` that produces a picture of a calendar that corresponds to the given year and month:

   ```
   def calendar(year: Int, month: Int): Picture = ...
   ```

Customizing printing:

Scala traits allow us to create modules with stackable modifications through mixin composition. As a simple example, let's first define a `CalendarPrinter` class with a single method `print`, which writes our calendar picture to the standard output.

```
class CalendarPrinter(val year: Int, val month: Int) {
  def print() = println(calendar(year, month).showIt)
}
```

Depending on your previous implementation, you may or may not need to modify the implementation of `print` above to include the names of the days of the week at the top of the calendar.

Next create traits `PrintMonth` and `PrintYear`, which override the `print` method to print the month and year, respectively, at the top of the calendar. The methods should call their superclass in order to print the rest of the calendar.

```
trait PrintMonth extends CalendarPrinter {
  override def print() { ... }
}

trait PrintYear extends CalendarPrinter {
  override def print() { ... }
}
```

We now want to print a calendar that displays both the month and the year at the top of the calendar. We can achieve this by creating a `CalendarPrinter` object that mixes in both `PrintMonth` and `PrintYear`. Observe and add a comment in the code explaining the difference in the output resulting from calling `print` on the following two objects:

```
new CalendarPrinter(year, month) with PrintYear with PrintMonth

new CalendarPrinter(year, month) with PrintMonth with PrintYear
```