P02 Calendar Printer

Programming II (CS300) Spring 2020	**Due: 9:59PM on February 5, 2020**
Department of Computer Sciences	Pair Programming **is Allowed.**
University of Wisconsin-Madison	Register by Sunday, Feb 2 @ 9:59PM

Overview and Learning Objectives:

One of my first steps (as an instructor) in planning for a new semester is to print out a calendar and begin marking it with important dates: like holidays and the dates of our CS300 Exams. If you have not already, please add our exam dates to your own calendar.

Through the process of completing this assignment, you will brush up your structured programming skills using Java while working with objects of some provided types: Day, Month, and Year. This assignment is meant to encourage you to think about the different ways that data can be represented within a computer program. You will also be developing test methods for your code, as you have seen in Programming Assignment 01: Piggy Bank.

Grading Rubric:

5 points	**Pre-Assignment Quiz:** You should not have access to this write-up until after you have completed the corresponding pre-assignment quiz through Canvas.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is correct. To become more confident in this, you must write and run additional tests of your own.
25 points	**Supplemental Automated Tests:** When your final grade feedback appears on Gradescope, it will include the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but check your code against different requirements within this specification.
50 points	**TOTAL**

Additional Assignment Requirements:

You **CANNOT** make use of java.util.Date. The only classes outside of java.lang and the default package that you are allowed to import and use for this assignment are:

- java.util.ArrayList
- java.util.Scanner

Reminders:

Everyone must complete the pair programming registration form by Sunday, Feb 2 @ 9:59PM.

Ensure that your code for every assignment is styled in conformance to the **Course Style Guide**.

Add **course exam dates** to your calendar and if needed, **request alternate exam times/accommodations**.

0. Project Files and Setup

Create a new Java Project in Eclipse called: P02 Calendar Printer. Then create two Java classes / source files within that project's src folder (both inside the default package) called **CalendarPrinter** and **CalendarTester**. Each of these classes should include their own main method. Next, download this provided P02.jar file to save within your project folder (outside of the src folder). In Eclipse, you may need to right-click on your project folder and click "Refresh" to see the new file appear in the Package Explorer. Then you can right-click on this P02.jar file within the Package Explorer view and choose: Build Path / Add to Build Path.

1. Creating First Test Method

To practice good structured programming, we will be organizing our CalendarPrinter implementation into several easy-to-understand sized methods. We want to test these methods before writing code that makes use of calling them. When we do find bugs in the future, we will add additional tests to demonstrate those defects, to help us reproduce and debug them, and then to monitor whether/when they are resolved and whether they ever return in the future.

Here are the JavaDoc style method header comments and signature for the first method for which we would like to write a test. Remember that we will be implementing this public static method inside our CalendarPrinter class, but only after we finish writing its test method inside CalendarTester.

```java
/**
 * Calculates the number of centuries (rounded down) between year 0 and the
 * specified year.  For example, the year 2034 has the century index of 20 (as do
 * the other years 2000-2099).
 * @param year to compute the century offset for
 * @return number of centuries between year 0 and the specified year
 */
public static int fullCenturiesContained(Year year)
```

Remember that we want to keep our tests short and simple to avoid accumulating too many extra bugs or development hours. You can copy these tests directly into your CalendarTester class as a starting example.

```java
public static boolean testFullCenturiesContained () {
  if(CalendarPrinter.fullCenturiesContained(new Year(2)) != 0) return false;
  if(CalendarPrinter.fullCenturiesContained(new Year(2020)) != 20) return false;
  if(CalendarPrinter.fullCenturiesContained(new Year(44444)) != 444) return false;
  return true;
}
```

If you would like to print out more specific feedback when tests fail (before returning false), that can also be helpful. Notice that this test method does not test the behavior when a null reference is passed to this method as the year. Since this write-up (including the JavaDoc style comments above) does not specify what should happen when a null reference is passed, we will consider any resulting behavior to be correct. Since any resulting behavior is correct, no test should fail under those circumstance, and we will not add such tests to our code.

If you are not familiar with the class Year that is used in the example above, that is probably because it was only recently created by your course staff and imported through the P02.jar file that you added to your build path in step 0. This is one of three classes and two enumerations that are being provided to help you implement this assignment, and to get you more practice working with objects in Java. The JavaDoc specifications for Year, Month, and Day classes are here. They should also appear in Eclipse when you mouse over any methods being called on these types of objects. Year objects are pretty simple, they just hold an int primitive that can be later retrieved. This is similar to an Integer wrapper class object, but is more specific in its intended use: to represent years in the Gregorian Calendar AD.

If you haven't already, finish implementing your CalendarPrinter.fullCenturiesContained() method, ensure that it passes your tests, and upload your progress to Gradescope before moving on.

2. Additional Helper Methods to Implement and to Test

In addition to the provided classes Year, Month, and Day, your P02.jar file contains definitions for two enumerations: DayOfWeek and MonthOfYear. If you have not worked with enumerations in Java recently, it may be helpful to review zyBook Ch 16.11. Some additional methods that are useful when working with enumerations are demonstrated below. You are welcome but not required to copy and trace through this code in a spare Java Project in Eclipse, or in the Java Visualizer.

```
DayOfWeek x = DayOfWeek.Wednesday;
System.out.println(x); // print out the enum value
DayOfWeek[] allValues = DayOfWeek.values(); // retrieve array of all values
System.out.println(java.util.Arrays.toString(allValues)); // print all values
System.out.println(x.ordinal()); // find the index of x within array of all values
System.out.println(allValues[x.ordinal()]); // look up x's value within this array
```

Below are the JavaDoc method headers and signatures for the helper methods that you must implement and test, similar to the fullCenturiesContained() method from the previous step.

```
/**
 * Calculates the number of years between the specified year and the first year of
 * its century.  For example, the year 2034 has the offset within its century of 34
 * (as do 1234 and 9999934).
 * @param year to compute the offset within century for
 * @return number of years between the specified year and the first year of century
 */
public static int yearOffsetWithinCentury(Year year)

/**
 * This method computes whether the specified year is a leap year or not.
 * @param year is the year is being checked for leap-year-ness
 * @return true when the specified year is a leap year, and false otherwise
 */
public static boolean isLeapYear(Year year)

/**
 * Calculates the number of days in the specified month, while taking into
 * consideration whether or not the specified month is in a leap year.
 * Note: that this is calculated based on the month's monthOfYear and year, and
 * is NOT retrieved from the month's getDayCount() method.  This is because this
 * method is used to generate the day objects that are stored within each month.
 * @param month to determine the number of days within (within its year)
 * @return the number of days in the specified month (between 28-31)
 */
```

```java
public static int numberOfDaysInMonth(Month month)

/**
 * Calculates which day of the week the first day of the specified month falls on.
 * Note: that this is calculated based on the month's monthOfYear and year, and
 * is NOT retrieved from the month's getDayByDate(1) day.  This is because this
 * method is used to generate the day objects that are stored within each month.
 * @param month within which we are calculate the day of week for the first day
 * @return DayOfWeek corresponding to the first day within the specified month
 */
public static DayOfWeek calcFirstDayOfWeekInMonth(Month month)

/**
 * Calculates the day of the week that follows the specified day of week.
 * For example, Thursday comes after Wednesday, and Monday comes after Sunday.
 * @param dayBefore is the day of week that comes before the day of week returned
 * @return day of week that comes after dayBefore
 */
public static DayOfWeek dayOfWeekAfter(DayOfWeek dayBefore)

/**
 * Calculates the month of the year that follows the specified month.  For example,
 * July comes after June, and January comes after December.
 * @param monthBefore is the month that comes before the month that is returned
 * @return month of year that comes after monthBefore
 */
public static MonthOfYear monthOfYearAfter(MonthOfYear monthBefore)

/**
 * Creates a new month object and fully initializes with its corresponding days.
 * @param monthOfYear which month of the year this new month represents
 * @param year in which this month is a part
 * @return reference to the newly created month object
 */
public static Month createNewMonth(MonthOfYear monthOfYear, Year year)
```

In addition to implementing each of these methods, be sure to create a different test method for each. Each of these tests should call the specific method that they are testing **AT LEAST THREE TIMES** with different inputs. In order for our automated grading tests to find and call these methods, be sure that each of these methods are named testX where X is the UpperCamelCase name of the method being tested (as you saw between fullCenturiesContained and testFullCenturiesContained). It is also important that all seven of these test methods are defined within your CalendarTester class as public static methods that take no arguments and return a boolean: true when a test passes and false otherwise.

Note that the Month returned by createNewMonth contains a lot of data. Your test may just check a few of the expected Days: like the first, last, and something in the middle, rather than checking for every single day within a given month. By the end of this step, your CalendarTester class should contain 8 test methods + the main method which calls each of those test methods.

3. Creating a Driver Application

These final methods will help drive your CalendarPrinter application. Since they are responsible for printing output to the Console and reading input from System.in, we will not worry about writing tests for these methods. Here are the JavaDoc header specifications for these last two methods that you must implement for this assignment:

```
/**
 * Prints the contents of the specified month object in calendar form.  This
 * printout should begin with the Month an year of the month.  The next line should
 * contain the three letter abbreviations for the seven days of the week.  And then
 * the dates of each day of that month should follow: one week per line, with
 * periods in positions of days that are a part of the month before or after.  For
 * example, January 2020 should be printed as follows:
 *
 * January 2020
 * MON TUE WED THU FRI SAT SUN
 *  .   .   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  31  .   .
 *
 * @param month which is to be printed by this method
 */
public static void printMonth(Month month)

/**
 * Creates an array list of months that are initialized with their full complement
 * of days.  Prints out each of these months in calendar form, and then returns the
 * resulting ArrayList.
 * @param month of the year for the first month that is created and printed
 * @param year that the first month created and printed is a part of
 * @param count is the total number of sequential months created and printed
 * @return the array list of months that this method generates and prints.
 */
public static ArrayList<Month> createAndPrintMonths(MonthOfYear month,
    Year year, int count)
```

To use and further test your implementations for the above methods, we are providing the following main method for your CalendarPrinter class.

```
/**
 * Driver for the CalendarPrinter Application.  This program asks the user to enter
 * an initial month, an initial year, and the total number of months to create and
 * display in calendar form.
 * @param args is not used by this program
 */
public static void main(String [] args) {
  // print welcome message
  Scanner in = new Scanner(System.in);
  System.out.println("Welcome to the Calendar Printer.");
  System.out.println("~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~");

  // read input from the user
  System.out.print("Enter the month to begin calendar: ");
  String monthString = in.nextLine().trim();
  System.out.print("Enter the year to begin calendar: ");
  String yearString = in.nextLine().trim();
  System.out.print("Enter the number of months to include in this calendar: ");
  String countString = in.nextLine().trim();

  // convert user input into usable form
  monthString = monthString.substring(0,3).toUpperCase();
  MonthOfYear month = null;
  for(int i=0;i<MonthOfYear.values().length;i++)

if(monthString.equals(MonthOfYear.values()[i].name().substring(0,3).toUpperCase()))
      month = MonthOfYear.values()[i];
  Year year = new Year(Integer.parseInt(yearString.trim()));
  int count = Integer.parseInt(countString.trim());

  // create months and display them in calendar form
  System.out.println();
  createAndPrintMonths(month,year,count);

  // display thank you message
  System.out.println("~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~");
  System.out.println("Thanks, and have a nice day.");
  in.close();
}
```

4. Assignment Submission

Congratulations on completing your second CS300 assignment! The only two files that you should submit through gradescope.com for this assignment are CalendarPrinter.java and CalendarTester.java. Your score for this assignment is based on your submission marked "active" and made prior to the hard deadline. If you are working with a partner, be sure to review and follow the course Pair Programming Policy for linking your final submission to both you and your partner's account.

5. Appendix: Implementation Tips for isLeapYear, and firstDayOfWeekInMonth

5.1. Tips for calculating isLeapYear

The following excerpt from Wikipedia is preserved here for use in this assignment (https://en.wikipedia.org/wiki/Leap_year#Algorithm):

The following pseudocode determines whether a year is a leap year or a common year in the Gregorian calendar (and in the proleptic Gregorian calendar before 1582). The year variable being tested is the integer representing the number of the year in the Gregorian calendar.

```
if (year is not divisible by 4) then (it is a common year)
else if (year is not divisible by 100) then (it is a leap year)
else if (year is not divisible by 400) then (it is a common year)
else (it is a leap year)
```

5.2. Tips for calculating calcFirstDayOfWeekInMonth

The following excerpt from Wikipedia is preserved here for use in this assignment (https://en.wikipedia.org/wiki/Zeller%27s_congruence#Implementation_in_software). Note that this formula is more general than is needed for this assignment, because it calculates the day of week for any day of any month, whereas your program only ever needs to compute the first day of the month:

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor + 5J\right) \bmod 7,$$

where

- h is the day of the week (0 = Saturday, 1 = Sunday, 2 = Monday, ..., 6 = Friday)
- q is the day of the month
- m is the month (3 = March, 4 = April, 5 = May, ..., 14 = February)
- K the year of the century (year mod 100).
- J is the zero-based century (the integer part of year/100) For example, the zero-based centuries for 1995 and 2000 are 19 and 20 respectively (to not be confused with the common ordinal century enumeration which indicates 20th for both cases).
- $\lfloor \ldots \rfloor$ is the floor function or integer part
- mod is the modulo operation or remainder after division

NOTE: In this algorithm January and February are counted as months 13 and 14 of the previous year. E.g. if it is 2 February 2010, the algorithm counts the date as the second day of the fourteenth month of 2009 (02/14/2009 in DD/MM/YYYY format)