A `SimpleDate` Class - How to tackle a large program

Assignment: The **`SimpleDate`** Class

Many applications, such as calendars and appointment managers, must be able to keep track of and manipulate dates. Java provides the java.util.Date and java.util.GregorianCalendar classes that might be useful for these applications. However, instead of using those classes this lab asks you to create and test a similar class named `SimpleDate`. The `SimpleDate` class does many of the same things that Java's `Date` and `GregorianCalendar` classes do, but it is somewhat simpler. The production of the `SimpleDate` class requires a substantial amount of conditional processing and is thus a good exercise to provide practice using Java's if and if/else statements as well as the logical operators.

For this problem you will need to create two files, `SimpleDate.java` and `SimpleDateTester.java`. In the `SimpleDate.java` file you will provide the definition and implementation of the `SimpleDate` class. A complete description of the public interface (API) to the `SimpleDate` class can be found at the end of this assignment. **USE IT** to make sure you have all the variables and methods you need. The method names need to match the API.

The `SimpleDateTest.java` file should contain a main method that, when executed, creates several `SimpleDate` objects and thoroughly tests each of the methods in the the `SimpleDate` class.

You should develop the `SimpleDate` class and the `SimpleDateTester` program incrementally. In other words, do not try to write the entire `SimpleDate` class and then write the `SimpleDateTester` program. Instead, start out by writing and testing small parts of the `SimpleDate` class at a time. The following paragraphs will walk you through one possible way carry out this incremental implementation and testing. Some suggested dates to test in your program:

```
SimpleDate d1 = new SimpleDate(2002);
SimpleDate d2 = new SimpleDate(8, 2001);
SimpleDate d3 = new SimpleDate(13,2001);
SimpleDate d4 = new SimpleDate(0, 2001);
SimpleDate d5 = new SimpleDate(30,3, 1988);
SimpleDate d6 = new SimpleDate(-9,6, 2006);
SimpleDate d7 = new SimpleDate(31,6, 2011);
SimpleDate d8 = new SimpleDate(29,2, 2011);
SimpleDate d9 = new SimpleDate(29,2, 2012);
SimpleDate d10 = new SimpleDate(30, 4, 2015);
SimpleDate d11 = new SimpleDate(31, 5, 2015);
SimpleDate d12 = new SimpleDate(4, 7, 1776);
SimpleDate d13 = new SimpleDate(3, 9, 1999);
SimpleDate d14 = new SimpleDate(31, 10, 2000);
SimpleDate d15 = new SimpleDate(25, 11, 1999);
SimpleDate d16 = new SimpleDate(25, 12, 2016);
```

You might begin implementing the `SimpleDate` class by writing just the declarations of the instance data, the `SimpleDate(int initYear)` constructor and the `getYear()` method. Once you have those written, you should write code in `SimpleDateTester` to test just the part of `SimpleDate` that you have written. So at this point the `SimpleDateTester` program might contain:

```
public class SimpleDateTester {
```

```
    public static void main(String[] args) {

      // Test the one argument constructor.
      // Expected output: date1Year: 2002
         SimpleDate date1 = new SimpleDate(2002);
         System.out.println("date1 Year: " + date1.getYear());
    }
}
```

Once you are convinced that the `SimpleDate(int initYear)` constructor and the `getYear()` method are correct you might add the `SimpleDate(int initMonth, int initYear)` constructor and the `getMonth()` method to the `SimpleDate` class. You should also add some additional tests to the `SimpleDateTest` program. Do not delete your old tests. Instead simply add new tests to them to test the newly added methods. For example the `SimpleDateTester` program might now contain:

```
// This is only a partial list.  You must add to it.
public class SimpleDateTester {
    public static void main(String[] args) {

      // Test the one argument constructor.
      // Expected output: date1 Year: 2002
      //                  date1 Month: 1
     SimpleDate date1 = new SimpleDate(2002);
      System.out.println("date1 Year: " + date1.getYear());
      System.out.println("date1 Month: " + date1.getMonth());

      // Test the two argument constructor.
      // Expected output: date2 Year: 2001
      //                  date2 Month: 8
      SimpleDate date2 = new SimpleDate(8,2001);
      System.out.println("date2 Year: " + date2.getYear());
      System.out.println("date2 Month: " + date2.getMonth());

      // Test the two argument constructor with invalid months.
      // Expected output: date3 Month: 1
      //                  date4 Month: 1
      SimpleDate date3 = new SimpleDate(13,2001);
      SimpleDate date4 = new SimpleDate(0,2001);
      System.out.println("date3 Month: " + date3.getMonth());
      System.out.println("date4 Month: " + date4.getMonth());
    }
}
```

Notice that you should be careful to test not only the straight forward use of the `SimpleDate` class but also the special cases that come up. **(Poor testing will lower your grade)** Also, you should include comments in the `SimpleDateTest` class describing the purpose of each test and the expected results of the test.

You might next implement and test the `isLeapYear()` method. (Use your leapYear program from the first unit as a model.) There are 3 other rules. When you test the `isLeapYear()` method you should be sure to include at least one test for each of the possible cases. For example you will need to include tests for:

- A year that is not divisible by 4.
- A year that is divisible by 4 but not divisible by 100 or 400.
- A year that is divisible by 4 and 100 but not 400.
- A year that is divisible by 4 and 100 and 400.

- A year prior to 1582 that is a leap year.
- A year prior to 1582 that is not a leap year.

Once you have implemented and tested the `isLeapYear()` method, the `daysInMonth()` method is a good next choice. This is because you will probably find it useful to call `isLeapYear()` from within your `daysInMonth()` method. For example, if the month is February (i.e. 2) then your `daysInMonth()` method will need to check to see if the current year is a leap year before returning the number of days in the month. So the `daysInMonth()` method might contain a fragment of code like this:

```
if (month == 2) {
  if (isLeapYear()) {
    ...
  } else {
    ...
  }
}
```

Notice that no object variable is required to call one instance method from another instance method. Java will automatically invoke the method on the object referred to by the implicit first parameter. So the above code is equivalent to writing:

```
if (month == 2) {
  if (this.isLeapYear()) {
    ...
  } else {
    ...
  }
}
```

Once you test the `daysInMonth()` method you will find it to be useful for implementing the `SimpleDate(int initDay, int initMonth, int initYear)` constructor. At this point you should have a good feel for what is involved in adding and testing each additional method of the `SimpleDate` class. Thus, you will need to make you own decisions about the order in which to implement and test the remaining methods of the `SimpleDate` class. One final word of caution; when you begin to implement the methods such

Grading

This lab will be worth 25 points. The `SimpleDateTester` program will be worth 5 points and your grade will be based on how completely your test cases test the `SimpleDate` class. The `SimpleDate` class will be worth 20 points and your grade will be based on how well your `SimpleDate` class meets my grading rubric

You must hand in a printout of the following files. Please use the CSD to indent your programs before you hand them in. You don't have to leave the blue marks. Put them in a word document.

`SimpleDate.java`
`SimpleDateTester.java`

Please be sure to pay attention to the following:

- Use blank lines in your code as clues to the reader that indicate logically distinct parts of your program.
- Choose descriptive names for your variables.
- Use comments to describe the role of statements in solving the problem if it is not obvious from the Java code itself.
- Prompt for each piece of input making it clear to the user what input is expected.
- Label all output making the meaning of the output clear to the user.
- Display results in an easy to read format.

as `advanceDay()`, `advanceMonth()` and `advanceYear()` you will need to **think very carefully about the test cases that are necessary to show that they are correct.**

Class SimpleDate

```
java.lang.Object
  |
  +--SimpleDate
```

---

public class **SimpleDate**
extends java.lang.Object

This class describes a SimpleDate object. SimpleDate objects represent a date including the day, month and year. The months of the year are represented as an integer ranging from 1 to 12 with Januaray=1 and December=12. The day of the month is represented as an integer with the first day of the month being 1.

Constructor Summary

SimpleDate(int initYear) Construct a new SimpleDate object with the day=1, month=1 and year=initYear.
SimpleDate(int initMonth, int initYear) Construct a new SimpleDate object with the day=1, month=initMonth and year=initYear.
SimpleDate(int initDay, int initMonth, int initYear) Construct a new SimpleDate object with the day=initDay, month=initMonth and year=initYear.

Method Summary

void	**advanceDay**() Advance this SimpleDate to the next day.
void	**advanceMonth**() Advance this SimpleDate to the same day next month.
void	**advanceYear**() Advance this SimpleDate to the same day next year.
int	**daysInMonth**() Return the number of days in the month represented by this SimpleDate.
int	**getDay**() Return the day represented by this SimpleDate.
java.lang.String	**getLongDate**() Return a long string representation of the SimpleDate object.
int	**getMonth**()

	Return the month represented by this SimpleDate.
java.lang.String	**getShortDate**() Return a short string representation of this SimpleDate object.
int	**getYear**() Return the year represented by this SimpleDate.
boolean	**isLeapYear**() Return true if the year represented by this SimpleDate is a leap year, otherwise return false.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SimpleDate

```
public SimpleDate(int initYear)
```

Construct a new SimpleDate object with the day=1, month=1 and year=initYear.

SimpleDate

```
public SimpleDate(int initMonth,
                  int initYear)
```

Construct a new SimpleDate object with the day=1, month=initMonth and year=initYear. If an invalid value is provided for the month its value should default to 1.

SimpleDate

```
public SimpleDate(int initDay,
                  int initMonth,
                  int initYear)
```

Construct a new SimpleDate object with the day=initDay, month=initMonth and year=initYear. If invalid values are provided for the day and/or the month the invalid value(s) should default to 1.

Method Detail

advanceDay

```
public void advanceDay()
```

Advance this SimpleDate to the next day. This method will also advance the month and year when appropriate.

advanceMonth

`public void **advanceMonth**()`

Advance this SimpleDate to the same day next month. If the current day exceeds the number of days in the next month the day should be set to the final day of the next month. This method will also advance the year when appropriate.

advanceYear

`public void **advanceYear**()`

Advance this SimpleDate to the same day next year. If the current day does not exist in the next year, the day should be set to the final day of the same month of the next year.

daysInMonth

`public int **daysInMonth**()`

Return the number of days in the month represented by this SimpleDate.

getDay

`public int **getDay**()`

Return the day represented by this SimpleDate.

getLongDate

`public java.lang.String **getLongDate**()`

Return a long string representation of the SimpleDate object. A long string representation will have the format: Month DD, YYYY. For example March 15, 2002 or October 5, 1987.

getMonth

`public int **getMonth**()`

Return the month represented by this SimpleDate.

getShortDate

`public java.lang.String **getShortDate**()`

Return a short string representation of this SimpleDate object. A short string representation will have the format: MM/DD/YYYY. For example 3/15/2002 or 10/5/1987.

getYear

```
public int getYear()
```

Return the year represented by this SimpleDate.

---

isLeapYear

```
public boolean isLeapYear()
```

Return true if the year represented by this SimpleDate is a leap year, otherwise return false.