CSE 142, Summer 2010
Programming Assignment #8: Birthday/Date (20 points)
Due Wednesday, August 18, 2010, 11:30 PM

This program focuses on classes and objects. Turn in two files named `DateClient.java` and `Date.java`. You will also need the support file `Date.class` from the course web site; place it in the same folder as your program.

The assignment has two parts: a client program that uses `Date` objects, and a `Date` class of your own whose objects represent calendar dates.

Part A (`DateClient.java`, client program):

The first part of this assignment asks you to write a client program that uses an existing `Date` class written by the instructor. The purpose of Part A is to give you a bit of practice creating and using `Date` objects from a client's perspective and to give you an appreciation for the usefulness `Date` objects in general.

Begin by prompting the user for today's date and for his/her birthday in *month day year* format (see examples below). Use this information to print the user's birthday in year/month/day format, the day of the week the user was born, and how old the user is in days. If the user was born in a leap year, you print an additional message letting them know their birth year was a leap year.

Below are several example logs of execution from the program; user input is bold and underlined. Your program's output should match these examples exactly given the same input. See the course web site for more logs with other input.

```
What is today's date (month day year)? 8 8 2010
What is your birthday (month day year)? 1 26 1971
You were born on 1971/1/26, which was a Tuesday.
You are 14439 days old.
```

```
What is today's date (month day year)? 8 8 2010
What is your birthday (month day year)? 2 2 1996
You were born on 1996/2/2, which was a Friday.
1996 was a leap year.
You are 5301 days old.
```

```
What is today's date (month day year)? 8 8 2010
What is your birthday (month day year)? 11 8 1900
You were born on 1900/11/8, which was a Thursday.
You are 40085 days old.
```

```
What is today's date (month day year)? 7 11 1856
What is your birthday (month day year)? 10 2 1800
You were born on 1800/10/2, which was a Thursday.
You are 20371 days old.
```

Solve this problem using `Date` objects. The methods and behavior of each `Date` object are described on the next page. For Part A you can use an instructor-provided version of `Date` by downloading the file `Date.class` from the web site and saving it to the same folder as your `DateClient.java` file. You can construct a `Date` object as follows:

```
Date name = new Date(year, month, day);
```

To figure out the number of days old the user is, represent today and the birthday as `Date` objects and use the methods found in the Date objects to figure out how many days are between them. Note: Some of the methods provided modify the state of the object on which they are called. See the next page for method descriptions.

You do need to take leap years into account for this assignment. A leap year occurs roughly every 4 years and adds a 29th day to February, making the year 366 days long. (Leap day babies usually celebrate their birthdays on February 28 or March 1 on non-leap years.) `Date` objects have various methods that can help you to detect and handle the leap year case. You may assume that neither today nor the user's birthday is the rare "leap day" of February 29.

Assume valid input (that the user will always type a year between 1753 – 9999, a month between 1-12, and a day between 1 and the end of that month).

Part B (`Date.java`, class of objects):

The second part of this assignment asks you to implement a class named `Date`, stored in a second file named `Date.java`. In the descriptions below the phrase "this `Date` object" refers to the object on which the method was called. Assume that all parameters passed to all methods are valid. Your `Date` class should implement the following behavior:

- `public Date(int year, int month, int day)`
  Constructs a new `Date` representing the given year, month, and day.
- `public int getYear()`
  This method should return this `Date` object's year, between 1753 and 9999. For example, if this Date object represents August 17, 2010, this method should return 2010.
- `public int getMonth()`
  This method should return this `Date` object's month of the year, between 1 and 12. For example, if this Date object represents August 17, 2010, this method should return 8.
- `public int getDay()`
  This method should return this `Date` object's day of the month, between 1 and the number of days in that month (which will be between 28 and 31). For example, if this Date object represents August 17, 2010, this method should return 17.
- `public String toString()`
  This method should return a `String` representation of this `Date` in a *year/month/day* format. For example, if this `Date` object represents May 24, 2010, return `"2010/5/24"`. If this `Date` object represents December 3, 1973, return `"1973/12/3"`. Note that this method *returns* the string; it does not print any output.
- `public boolean equals(Date d)`
  Returns `true` when this `Date` object represents the same date as the given `Date` parameter. Returns `false` otherwise.
- `public boolean isLeapYear()`
  Returns whether this `Date`'s year is a leap year. Leap years are all years that are divisible by 4, except for years that are divisible by 100 but not by 400. For example, 1756, 1952, 2004, 1600, and 2000 are all leap years, but 1753, 2005, 1700, and 1900 are not.
- `public void nextDay()`
  Modifies the state of this `Date` object by advancing it 1 day in time. For example, if this `Date` object represents September 19, a call to this method should modify this `Date` object's state so that it represents September 20. Note that depending on the date, a call to this method might advance this `Date` object into the next month or year. For example, the next day after August 17, 2010 is August 18, 2010; the next day after February 28, 1997 is March 1, 1997; and the next day after December 31, 1978 is January 1, 1979.
- `public int advanceTo(Date endDay)`
  Modifies the state of this `Date` object by advancing it to the given end `Date`. This method returns the number of days it took to advance this `Date` object to the given end `Date`. For example, if this `Date` object represents September 19, 2010 and the `endDay` represents September 29, 2010, a call to this method modifies this `Date` object's state so it also represents September 29, 2010 and returns `10`. Depending on the date, a call to this method might advance this `Date` object into a different month or year. You may assume the given `Date` object always comes after this `Date` object.

Continued on the next page…

- `public String getDayOfWeek()`
  Returns a `String` representing what day of the week this `Date` falls on. The `String` will be "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", or "Saturday". For example, August 8, 2010 fell on a Sunday, so the return value for a `new Date(2010, 8, 8)` object would be "Sunday". At the end of a call to this method, the state of this `Date` object should be the same as it was at the start of the call. You should base your calculations on the fact that January 1, 1753 was a Monday.

None of the methods listed above should print any output to the console. You may not utilize any behavior from the instructor-provided `Date` class to help implement your own `Date` class, nor use any of Java's date-related classes such as `java.util.GregorianCalendar` or `java.util.Date`.

You can test your `Date` program by running your `DateClient.java` program from Part A with it. By compiling your `Date.java` file you will overwrite our provided `Date.class` with your own. (If necessary you can always revert to our `Date.class` by re-downloading it or backing it up.) `DateClient.java` is not a great testing program; it might not call all of your `Date` methods or may not call them in a very exhaustive way that tests all cases and combinations. Therefore you may want to create another small client program of your own to help test other aspects of your `Date` class's behavior.

You may put additional behavior in your `Date` class if you like, but we will still test your `DateClient` program with the instructor-provided `Date` class, so it should still run correctly with that class and not only when used with your `Date`.

Development Strategy and Hints:

Complete Part A before Part B, to get a good understanding of how `Date` objects work from the client's perspective.

Write Part B in phases:

- Write the constructor and `getYear`, `getMonth`, and `getDay` and methods first.
- Then implement `toString`, `equals`, and `isLeapYear`.
- Next, try to write `nextDay`. (You may wish to write a helping method that returns the number of days in this `Date`'s month, to help you implement `nextDay`. If you decide to write this method, please remember that if it is a leap year, February will have 29 days.)
- Next, write `advanceTo`. You should be able to use methods you have already written to write this.
- Lastly, write `getDayOfWeek`. You might want to construct a `Date` object for January 1, 1753 to help you. Again, you should use the other methods you have already written to help you write this method. Remember that the `Date` represented by the object should be the same as it was after a call to `getDayOfWeek` (this is different from `nextDay` and `advanceTo` which modify the state of the object).

We encourage you to build your `Date` class incrementally, writing a small amount of code at a time and testing it. It is possible to test an incomplete `Date` class by writing some of its methods and then creating a small client program to call just those methods.

Recall that code in one of an object's methods is able to call any of the object's other methods if so desired. Specifically, when implementing `advanceTo` and `getDayOfWeek` you may want to consider calling other methods within the `Date` object to help you.

Since objects can be difficult to visualize and understand, we strongly recommend that you use the jGRASP debugger to step through your code to understand each method's behavior, especially in Part B. If you are stuck on a particular method, we also recommend using temporary debugging `println` statements from inside the `Date` class to see what is going on. For example, printing the state of the current `Date` object from inside the `nextDay`, `advanceTo`, and `getDayOfWeek` method can help you find bugs.

Style Guidelines:

For Part A, you are to solve the problem by creating and using `Date` objects as much as possible. This is because a major goal of this assignment is to demonstrate understanding of using objects and defining new classes of objects. In Part A, you should have at least **2 methods other than** `main` to solve the problem. No one method should be overly long, and each method should perform a coherent task. Your main method should still contain the overall control flow of the program.

For Part B, implement your `Date` as a new type of object, using non-static methods, non-static data fields, constructors, etc. as appropriate. You should also properly encapsulate your `Date` objects by making their methods and constructors `public` and their data fields `private`. As much as possible you should avoid redundancy and repeated logic within the `Date` class. Avoid unnecessary fields: Use fields to store the important data of your `Date` objects but not to store temporary values that are only used within a single call to one method.

On both Parts of the assignment, you should follow general past style guidelines such as: appropriately using control structures like loops and if/else statements; avoiding redundancy using techniques such as methods, loops, and `if/else` factoring; properly using indentation, names, types, variables; and not having lines longer than 100 characters. You should properly comment your code with a proper heading in each file, a description on top of each method, and on any complex sections of your code. Specifically, place a comment heading at the top of each method of the `Date` class, written in your own words, describing that method's behavior, parameters, and return values if any.

You are limited to features in Chapters 1 through 8. For reference, our `DateClient.java` solution is around 50-60 lines including comments, and our `Date.java` solution is around 120-130 lines.