Lab 1 — A First Sip of Java

Go to the course web page under *Labs* and download the file **lab01_files.zip**. Unzipping this file will create a folder called lab01_files containing all of the files you will need for this lab.

Part 1: Printing Output

1. The file **PrintingDemo.java** contains various examples of printing and formatting output in Java. Study the code carefully, and then compile and run it. Make sure you understand what each line of code does.

Part 2: BankAccount

2. The files **BankAccount.java** and **TestBankAccount.java** implement a simple BankAccount class. Compile and run the TestBankAccount program, then add a couple more BankAccount objects to the program and test their methods.

3. Add a new method `getID()` to the BankAccount class that returns an account's ID number, and test it.

4. Rewrite the `toString` method using `String.format` instead of the string concatenation operator (+), and rerun the test program to make sure that BankAccount objects are still printed correctly.

5. Add `if/else` statements to the `deposit` and `withdraw` methods to disallow invalid input, such as negative amounts or (in the case of `withdraw`) amounts greater than the current balance. Also have each method print a confirmation message such as "Withdrew $75.00 from account 1001". Make sure to test your methods by adding some additional code to TestBankAccount.

6. Add a method `transfer(double` *`amount`*`, BankAccount` *`other`*`)` to BankAccount, which takes a dollar amount and another BankAccount object as arguments and transfers the specified amount to the other account, as long as the amount is not negative and does not exceed the available funds. Do this by updating the `balance` variables of each object directly (without calling `deposit` or `withdraw`). The `transfer` method is allowed to access the instance variables of other BankAccount objects directly because it is part of the BankAccount class itself. This method does not return a specific value, so its return type should be `void`. Modify TestBankAccount accordingly to test your `transfer` method. Examples:

Method Call	***Sample Output***
`a1.inquiry()`	`Account 1001 balance is $600.00`
`a2.inquiry()`	`Account 1002 balance is $250.00`
`a1.transfer(100, a2)`	`Transferred $100.00 from account 1001 to account 1002`
`a1.inquiry()`	`Account 1001 balance is $500.00`
`a2.inquiry()`	`Account 1002 balance is $350.00`

Part 3: CalendarDate

The files **CalendarDate.java** and **TestCalendarDate.java** contain skeleton code for the CalendarDate class. A CalendarDate object represents a calendar date by storing three internal instance variables of type `int` (integer) to represent the month, day, and year. For example, July 4, 1776 is month 7, day 4, and year 1776 (which happens to be a Thursday). The provided class definition contains a few methods and instance variable declarations to get you started. Your job is to implement each of the CalendarDate methods described below.

To help you out, I will also provide you with a complete Python implementation of the CalendarDate class. You can use the Python code for each method as a guide in implementing the equivalent Java code. If you aren't familiar with Python, just think of the code as a "pseudocode" description of what each method should do.

int getMonth() returns the CalendarDate's month as an integer (e.g., 7).

int getDay() returns the day (e.g., 4).

int getYear() returns the year (e.g., 1776).

boolean equals(CalendarDate *other*) returns true if the month, day, and year of this date are the same as the *other* date.

boolean isLeapYear() returns true if this date occurs in a leap year.

int lastDayOfMonth() returns the last day of this date's month, taking leap years into account.

boolean isValid() returns true if the date is valid, that is, if year ≥ 1 and $1 \leq$ month ≤ 12 and $1 \leq$ day $\leq$ *last*, where *last* is the last day of this month, taking leap years into account. Otherwise it returns false.

String getWeekday() returns the weekday name of this date (e.g., "Thursday"). To compute the weekday, we use the following algorithm:

1. Let D be the day of the month (1 through 31), and let Y be the year.

2. Let M be 1 for March, 2 for April, 3 for May, and so on. Let M be 11 for January and 12 for February, but for these two months, subtract 1 from Y before proceeding to the next step.

3. Let X be the first two digits of Y, and let Z be the last two digits of Y.

4. Compute $F = (26M - 2)/10 + D + Z + Z/4 + X/4 - 2X$
   All divisions are integer divisions in the sense that any remainders are discarded.

5. Now let $W = F$ (modulo 7). That is, divide F by 7 and retain only the remainder.
   But if the remainder is negative, add 7 to it to get W.

W now represents the day of the week: 0 is Sunday, 1 is Monday, 2 is Tuesday, and so on.

String toString() returns a string representation of the date of the form "Thursday, July 4, 1776" including the weekday. If the date is invalid, the string "Bad CalendarDate" is returned instead.

void advanceOneDay() advances the date by one day. If the day is the last day of the month, resets the day to 1, and adds 1 to the month. If the resulting month is greater than 12, resets the month to 1 and adds 1 to the year.

void advanceNumDays(int *numDays*) advances the date by *numDays* days.

The test program **TestCalendarDate.java** can be used to manually test your class definition. For a more rigorous test, you can use the compiled class files **SuperTest.class** and **CalendarDateSolution.class**, which are also provided in the lab01_files folder. Just type the following command at the > prompt in the Interactions panel, located in the bottom part of the DrJava window:

```
> java SuperTest
```

The SuperTest program will thoroughly test each of your CalendarDate methods and report any problems it finds.