CSc 227 — Program Design and Development
Spring 2014 (McCann)

http://www.cs.arizona.edu/classes/cs227/spring14/

# Program #2: Two Programs: Hailstones and One-Month Calendar

*Due Date: February 4$^{th}$, 2014, by 9:00 p.m. MST*

**Overview:** Now that you know a little more Java, you can write much more interesting programs. The Hailstones program is quite simple and should help you get a feel for selection and iteration in Java. Computing and displaying a calendar for a given month is much more involved; it should prove to be a more significant challenge.

**Assignment:** You are to write two distinct Java programs, `Prog2a.java` and `Prog2b.java`.

(a) `Prog2a.java`: **Hailstones**.

There are many problems in mathematics that are easy to describe but which have properties that are difficult to prove. One example is the "hailstone" problem, also known as *The Syracuse Problem*, *The Collatz Problem*, and *The 3n+1 Conjecture*, among others.

"Hailstones" are generated in the following fashion. Begin with a positive integer value $n_0$. If $n_0$ is an even number, the integer value $n_1$ is computed to be $n_0$ divided by 2. If $n_0$ is an odd number, $n_1$ is 3 times $n_0$, plus 1. In more formal notation:

$$n_{i+1} = \begin{cases} n_i/2 & \text{if } n_i \text{ is even} \\ 3n_i + 1 & \text{if } n_i \text{ is odd} \end{cases}$$

So what's so tricky about hailstones? No one has yet been able to prove that the hailstone sequence always terminates at 1 for all positive integers, although that certainly seems to be true, and it has been shown to be true for all values up to $3.458 \times 10^{18}$. About this conjecture, mathematician Paul Erdos was quoted as saying, "Mathematics is not yet ready for such problems."

You are to write a complete and well-documented Java program that computes the hailstones that result from a given positive integer value. Your program should display the members of the sequence to the screen, starting with the given input value, with exactly eight sequence members per line (except the final line, which can have fewer). In addition, your program is to display the quantity of values in the sequence (including the starting value), as well as the largest member in the sequence.

Have your program cease generating members of the sequence after the number 1 is generated and displayed. (Can you see why?)

Here's some sample output. Your program's output doesn't have to look exactly like this, but it does have to be well-organized and contain all of the required elements:

```
Enter the starting value for the hailstone sequence: 52

The sequence of hailstones formed from 52 is:

      52      26      13      40      20      10       5      16
       8       4       2       1

The hailstone sequence contains 12 hailstones, and the largest
value in the sequence is 52.
```

(Continued...)

(b) `Prog2b.java`: **One-Month Calendar**.

America is currently operating under the Gregorian Calendar, adopted in various parts of the world at various times, often centuries apart. In November of 2004, Sohael Babwani published an article in The Mathematical Gazette describing a new way to compute the day of the week (Monday, Tuesday, etc.) on which a particular date falls. Let's call it Babwani's Congruence. It's a bit of a mess, but then so is the Gregorian Calendar!

To use Babwani's Congruence, we need to define some terms. $d$ is the day of the month, $m$ is the number of the month (where 1 is January, 2 is February, etc.), $c$ is the integer formed from the first two digits of the year (the 'century,' you might say), and $y$ is the integer formed from the last two digits of the year (the year of the 'century'). For example, for the date January 29, 2007, $d = 29$, $m = 1$, $c = 20$, and $y = 7$. The day of the week $w$ (where 0 is Saturday, 1 is Sunday, etc.) can be determined by performing this calculation:

$$w = \left( \left\lfloor \frac{5y}{4} \right\rfloor + monthCode + d - 2(c \,\%\, 4) + 7 \right) \,\%\, 7 \ ^{\dagger}$$

where $\lfloor x \rfloor$ is the 'floor' function (see `Math.floor()` in the Java API) and *monthCode* is an integer value that is determined by the given month ($m$). The following table shows the associations. Note that only the month codes of January and February change in leap years:

| | | — *monthCode* — | |
|---|---|---|---|
| Month | $m$ | Non-Leap | Leap |
| January | 1 | 0 | **6** |
| February | 2 | 3 | **2** |
| March | 3 | 3 | 3 |
| April | 4 | 6 | 6 |
| May | 5 | 1 | 1 |
| June | 6 | 4 | 4 |

| | | — *monthCode* — | |
|---|---|---|---|
| Month | $m$ | Non-Leap | Leap |
| July | 7 | 6 | 6 |
| August | 8 | 2 | 2 |
| September | 9 | 5 | 5 |
| October | 10 | 0 | 0 |
| November | 11 | 3 | 3 |
| December | 12 | 5 | 5 |

To complete the January 29, 2007 example, we note that 2007 is not a leap year, and so *monthCode* is 0. The calculation proceeds:

$$\begin{aligned} w &= (\lfloor \tfrac{5(7)}{4} \rfloor + 0 + 29 - 2(20 \,\%\, 4)) \,\%\, 7 \\ &= (8 + 0 + 29 - 2(0)) \,\%\, 7 \\ &= 37 \,\%\, 7 \ = \ 2 \end{aligned}$$

Thus, January 29, 2007 fell on a Monday.

One last detail: How do you tell that a year is a leap year? Many people believe that any year evenly divisible by 4 is a leap year. That's only mostly true: Not all years that end in '00' are leap years; '00' years that are evenly divisible by 400 are leap years, the others are not. Thus, 2016 will be a leap year ($2016 \,\%\, 4 = 0$), 2000 was a leap year (it's evenly divisible by both 100 and 400), but 2100 will not be a leap year (it's not evenly divisible by 400).

You are to use Babwani's Congruence to display the calendar for a month specified by the user. For example, if the user asks for the February 2014 calendar, your program would display:

```
February 2014:

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
```

We want the calendar to be formatted exactly like this example. Of course, before you can display this, you'll also have to ask the user for the month and year, but you already know how we like user prompting to look.

(Continued...)

[†]The "+7" is an adjustment to Babwani's expression to prevent negative dividends, which are handled differently by Java.

**Exclusions:** We know that many of you know a lot more Java than we've covered in class so far. For this assignment, you **MAY NOT** use it! That's right: No arrays, no string variables or methods, no user-defined methods (other than `main()`) – none of that fun stuff. You can use methods from the `Math`, `System`, and `Scanner` classes that deal with the primitive types (and you can print string literals), but no methods from any other classes. And, of course, you can use the other basics of Java that we *have* covered, such as variables, constants, math operators, selection and iteration statements, etc. Yes, your `main()` will be rather lengthy; structure it as best you can to make your code readable. We're doing this to make a point (or several), but those are lessons for another time.

**Turn In:** Use the 'turnin' page to electronically submit both of your programs (`Prog2a.java` and `Prog2b.java`) to the `cs227p02` directory at any time before the stated due date and time.

**Grading Criteria:** We included grading criteria on the first program, but probably won't on any other assignment. Why not? It's not that we have anything to hide; it's that criteria creation is the responsibility of the section leaders, and they don't start creating it until after the program handouts are written. That's not much of a problem for you: If you do everything we require, and do it well, you'll be fine. And yes, we do give partial credit.

**Want to Learn More?**

- Why the name "hailstones?" Because the 'up and down' behavior of the sequence of values is reminiscent of how hailstones are formed in a thunderstorm. A description with diagrams, pictures, and a video can be found here:
  `http://islandnet.com/~see/weather/elements/hailform.htm`
- Details of Babwani's Congruence are available from a document linked to this web page:
  `http://www.babwani-congruence.blogspot.com/`
- The origin of leap years is interesting; you can read about that here:
  `http://www.timeanddate.com/date/leapyear.html`

**Other Requirements and Hints:**

- Include rudimentary input error checking in your programs. For instance, if the user enters -6 to start a hailstone sequence, tell the user to enter a positive integer and loop back to let them try again. Similarly, in the calendar program, if a user enters non-positive values, or months outside the 1 - 12 range, catch them. But if they give 56 or 13018 as the year, well, let them, even if the Gregorian Calendar didn't or may not exist then. You also don't have to worry about testing for non-numeric input, such as punctuation and characters. Finally, you don't have to worry about dealing with huge numbers. If a user starts a hailstone sequence with 2000000001 and the integer variables overflow as a result, we don't care.
- Achieving the spacing seen in the sample output of these programs is easy if you use the `System.out.printf()` method that was added to Java in version 1.5. With it, you can specify field sizes into which output values are placed. For example:
  `System.out.printf("%8d",value);` will display `value` right-justified in a field of size 8. See the table on the Primitive Data Types handout for some additional format codes.
- Document your code as you saw/did in the programs of the first assignment: Use meaningful variable names, put the 'external' block comment at the top, insert comments after each variable declaration, add brief comments ahead of potentially confusing portions of the code, etc.
- Be sure to take the time to verify the output of (that is, test) your programs. In particular, there are quite a few 'day of the week' calculators on the web; you should have no trouble finding one against which you can compare your program's output. Keep in mind that being on the web is not the same as being correct!
- Many of the little reminders from the first program apply here (as well as to all future programs!): You may use late days if you still have them to use, this is an individual assignment (do your own coding!), the schedule for section leader lab hours is available from the class web page, etc.
- As always: Start early! Sure, you've got six days, but why waste the first four or five of them? Start early, finish early, and enjoy the feeling that comes with being done early.