CS234: DATA TYPES AND STRUCTURES (F'01)

Assignment 1: Arbitrary precision integer calculator.

Assigned: 09/20

Due: 10/09, electronically by 11:30a.m.

Overview: The assignment develops a *Reverse Polish Notation* (RPN) integer calculator that can manipulate very large integer values. The solution should be developed in two parts: an ADT for manipulating very large integers and the calculator itself (that uses the ADT).

1 Arbitrary Precision Integer Arithmetic

Design and implement an ADT "`APNumber`" that allows representing and manipulating arbitrary-precision (number of digits) integers.

The ADT should support (at least) the following methods:

- `public void APNumber()`
  takes no input, creates a new instance of `APNumber`, and sets it to 0. (in PASCAL, this function returns a reference to the new `APNumber`).
- `public void negate()`
  takes as its input an `APNumber` and complements it.
- `public void add(APNumber operand)`
  adds the value of the argument to its own value (i.e., the value of the `APNumber` object the method is invoked on); the value of the argument `APNumber` must not change.
- `public void subtract(APNumber operand)`
  subtracts the value of the argument from its own value; again, the argument `APNumber` must not change.
- `public void multiply(APNumber operand)`
  multiplies its own value by the the `APNumber` supplied in the argument.
- `public boolean isEqual(APNumber operand)`
  compares the value of its own value with the `APNumber` supplied as argument; returns `true` if they are equal and `false` otherwise.
- `public boolean isGreaterThan(APNumber operand)`
  compares the value of its own value with the `APNumber` supplied as argument; returns `true` if the first one is strictly greater than the second and `false` otherwise.
- `public void input(char digit)`
  add a digit represented by the character supplied in the argument $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ as the next least significant digit to its own value. For example if you have an `APNumber` `v` $= 1234$ then executing `v.input('5')` will change the value stored in `v` to `v` $= 12345$.
- `public char sign()`
  returns the sign of the `APNumber`,

- `public char firstDigit()`
  returns the most significant digit of the `APNumber`,
- `public char nextDigit()`
  returns the next most significant digit of the the `APNumber`.

  For example, for $v = -9876543$ the calls to the above three functions in the order `v.sign()`, `v.firstDigit()`, `v.nextDigit()`, `v.nextDigit()`, `v.firstDigit()`, `v.nextDigit()` return the character values '−', '9', '8', '7', '9', '8', respectively. `nextDigit()` returns a space (`' '`) if there are no more digits in the stored number. `nextDigit` should fail if `firstDigit` hasn't been called.

Extra Credit of up to 20%:

- `public void divide(APNumber divisor)`
  divides (integer division) the `APNumber` by the one supplied in the first argument; again, the argument `APNumber` must not change.
- `public void mod(APNumber modulus)`
  computes the remainder of division of the `APNumber` by the one supplied in the first argument.

The implementation of `divide` and `mod` is for **extra credit** of up to 20%, i.e., you can achieve full credit for this assignment *without* implementing these two functions.

Here is the summary of the ADT functions:

Summary of operations				
Name	Input	output	required time	best time
`APNumber`	none	`APNumber`	$O(1)$	$O(1)$
`negate`	`APNumber`	none	$O(1)$	$O(1)$
`add`	two `APNumbers`	none	$O(n)$	$O(n)$
`subtract`	two `APNumbers`	none	$O(n)$	$O(n)$
`multiply`	two `APNumbers`	none	$O(n^2)$	$O(n \log n \log\log n)$
`divide`	two `APNumbers`	none	$O(n^2)$	$O(n \log n \log\log n)$
`mod`	two `APNumbers`	none	$O(n^2)$	$O(n \log n \log\log n)$
`isEqual`	two `APNumbers`	boolean	$O(n)$	$O(n)$
`isGreaterThan`	two `APNumbers`	boolean	$O(n)$	$O(n)$
`input`	char, `APNumber`	none	$O(1)$	$O(1)$
`sign`	`APNumber`	char	$O(1)$	$O(1)$
`firstDigit`	`APNumber`	char	$O(1)$	$O(1)$
`nextDigit`	`APNumber`	char	$O(1)$	$O(1)$

The column labeled "required time" is the time complexity your solution should achieve, where n refers to the **length** (in digits) of the numbers involved in the operation rather than to their actual value. Also, the operations should run in constant space (though, you may use the space for the result as a working space). In your solution **you must** justify why your solution satisfies these complexity bounds.

Notes:

- If you use a linked list implementation, you MUST implement the linked list on your own (in particular, you must not use Vectors or similar objects in JAVA).
- The number of digits in an `APNumber` should only be limited by the availability of memory.

2 The Calculator

To finish the assignment use the `APNumber` ADT to develop a *RPN* (postfix notation; cf. pages 81–84 in the textbook) calculator that reads a postfix arithmetic expression terminated by a ";", evaluates the expression using the `APNumber` ADT, and prints out the result(s).

Here are the operations supported by the calculator:

Input	Action
`nnnn`	enter a new number `nnnn`
`~`	replace last entry by its complement
`+`	replace last two entries by their sum
`-`	subtract the last entry from the second last one
`*`	replace last two entries by their product
`/`	divide the second last entry by the last entry
`%`	compute the remainder of "second last entry/last entry"
`=`	replace last two entries by 1 if they are equal and by 0 otherwise
`>`	replace last two entries by 1 if the last one is the greater one, by 0 otherwise
`D`	duplicate the last entry
`S`	swap the last two entries
`;`	print out all (remaining) entries

where `nnnn` stands for a (potentially very long) sequence of digits. The calculator should report an error if an operation is requested but have been too few values present to complete the operation.

The calculator reads its input from *standard input* and prints the results to *standard output*. The output *must* look as in the following example (further samples will be posted on the class web page). In particular, the calculator must be able to read consecutive RPN expressions and should terminate only when it encounters the end-of-file on the input.

The symbols on the input may not be separated by spaces (except for consecutive numbers) and the number of spaces must not matter. For example the inputs "`1 2+ ~;`" and "`1 2 +~  ;`" must be treated as the same expression.

Example: A sample session with the calculator:

```
% java Calc
12 34 56 * ~ 78 - + ;
Results:
-1970
```

```
12DD*;
Results:
12
144

34 * ;
Error: not enough operands for '*'
```

Notes:

- The numbers on the input may be quite large
- You must not use ADTs from JAVA libraries
- You can omit the '`/`' and '`%`' operations if you didn't attempt the extra credit in part 1; implementing these two operations is part of the extra 20%.

Assignment submission. On the assignment due date you are to submit a completed implementation of the integer calculator. In particular, this requires that you write a function in JAVA (PASCAL) for each of the operations of the `APNumber` ADT.

Your submission should include the following three files:

1. `APNumber.java` (`APNumber.p`) that contains a *Java Class* (*Pascal unit*) implementation of the `APNumber` ADT.
2. `Calc.java` (`Calc.p`) containing the implementation of the calculator, and
3. `Output.txt` that contains test cases that exercise your implementation (the test cases should demonstrate that your implementation is correct and part of the grade will depend on the quality of your test cases).
4. If you are using JAVA, you may submit additional `xxx.java` files that implement additional classes needed in your solution. However, you must make sure that the complete assignment (i.e., the arithmetic ADT, the calculator, and any additional files can be compiled using a single `javac *.java` command.
5. you are **responsible** for reading any clarifications for the assignment posted to the newsgroup, in particular for additional submission instructions posted there.

Final **important** note: you must include a comment at the start of each function indicating the complexity of the worst case running time of the function in the terms of the size of its inputs. In particular, for your implementations of the arithmetic operations for the `APNumber` ADT you must indicate the complexity of the operation in the sizes of both operands (independently). Your comment must clearly indicate how you have measured problem size.