COP 3330 – Object Oriented Programming in Java
Reading Input Line by Line
ArrayList, ArrayDeque classes

Typical Method of Reading Input: Tokenization

In most programming languages, students are taught initially how to read input from the user by reading in token by token. A token is a string of non-white space characters separated by white space (spaces, tab, newlines, carriage returns). In Java, we can read tokens using the following methods from the Scanner class:

```
next()
nextInt()
nextDouble()
nextLong()
```

There are other methods that read tokens, but these are the most typical ones used. A method that does NOT read a token is:

```
nextLine()
```

A critical rule of parsing input is the following (that is not followed by many high school teachers, unfortunately!!!):

NEVER MIX READING IN INPUT BY TOKENS WITH READING INPUT IN BY LINES.

To put this more concretely, if you have a program where you call the nextLine() method on a Scanner object, you must never call **ANY** of the methods that read in a single token, such as next(), nextInt(), nextDouble() or nextLong(). Similarly, if you have a program that calls any method on a Scanner that reads in a token, then you must never call nextLine() in that same program.

Due to this rule and the fact that we haven't taught the use of nextLine() yet, all of the assignments given in the class up until now can be solved via reading only tokens for input.

Thus, if you you didn't get a point deducted on a program yet for calling nextLine(), you should have gotten a point deducted!!!

I won't rehash how to process input where we alternate through the four method calls listed initially to read input token by token. For this to work, we must know what input is coming in each token (is the next token a String? Is it a double?)

But sometimes, we might hit one of the following two situations:

1) An input is given on a line, but has a variable number of tokens (consider someone's full name)
2) Input is separated by something other than a space, obfuscating the real type of input (such as a date or time)

If input for a program has either of these two issues, then one **<u>must read the input line by line only, calling the nextLine() method on the Scanner object and none of the other methods that automatically tokenize.</u>**

Once we read in a line, we need a tool to tokenize all the tokens on that line. The tool we'll use is the StringTokenizer class that is built into Java. Here is the Java doc:

https://docs.oracle.com/javase/8/docs/api/java/util/StringTokenizer.html

We'll talk about two of the constructors:

```
// Creates a new StringTokenizer object from str, with default
// delimiters which are all white space characters.
StringTokenizer(String str)

// Creates a new StringTokenizer object from str, using each
// of the characters listed in delim as delimiters.
StringTokenizer(String str, String delim)
```

Probably the easiest way to think about a StringTokenizer object is that it takes the original string and splits it into a list of separate strings, based on the delimiters. For example,

```
StringTokenizer tok = new StringTokenizer("Jessica Rae Thompson");
```

would return an object that internally stores something like this:

```
{ "Jessica", "Rae", "Thompson" }
   ^
```

The carat symbol is indicating that the next token available is "Jessica".

Once the StringTokenizer object is created, we have several methods we can call on it, but the most important are:

```
// Returns the next token in this StringTokenizer
String nextToken()

// Returns true if there are more tokens left to return from
// this StringTokenizer
boolean hasMoreTokens()

// Returns the number of tokens left to return from this
// StringTokenizer
int countTokens()
```

Each time we call nextToken(), the String that is "up next" gets returned, and gets removed from the list with the pointer/bookmark moving up to the next token in the list. For example, if we called

```
String first = tok.nextToken();
```

Then first would equal "Jessica" and the tok would look like:

```
{ "Rae", "Thompson" }
   ^
```

Once you retrieve a token from a StringTokenizer object, you can't "put it back into" the object.

At this point, we could do something like this:

```
String[] lastNames = new String[tok.countTokens()];
```

At this point in time, countTokens() will return 2 so we create an array of String of size 2, to store the two last names. To store those names in this array, we would do:

```
for (int i=0; i<lastNames.length; i++)
    lastNames[i] = tok.nextToken();
```

Notice that we should NOT run the loop to tok.countTokens(), because that method call will return a different number each time and not operate the way we want the loop to operate. In contract, the expression lastNames.length will not change for the duration of the loop and it properly represents how many times we must retrieve the next token.

Converting Strings to Numeric Data
One limitation of the StringTokenizer class is that it can only return a String object. It can't return an Integer object or a Double object. Thus, when we retrieve a token that we KNOW must be an integer, long or double, we need a method to convert the String returned by the nextToken() method into the appropriate type. Luckily, each of the wrapper classes for the primitive classes has an appropriate method. Here are the methods:

```
Integer.parseInt(String str)
Long.parseLong(String str)
Double.parseDouble(String str)
```

Imagine a situation where we create a StringTokenizer as follows:

```
StringTokenizer lineToks = new StringTokenizer("candy 3.99");
```

We can extract the two pieces of data we need as follows:

```
String item = lineToks.nextToken();
double price = Double.parseDouble(lineToks.nextToken());
```

Since nextToken() always returns a String, when necessary, we can just pass that string off to the appropriate parse method.

Quick Note on the Wrapper Classes
All the primitive types in Java have a corresponding "wrapper" class provided by Java. A wrapper class is a class, but all it really does is store a value of the corresponding primitive type. In a bit we'll see why this is so useful. Here is a listing of the Integer wrapper class:

https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html

Essentially, all of these methods are static, but they allow us to house some very common methods, such as min and max, and more importantly, Java only allows us to create collections of non-primitive objects, but it may very well be the case that one wants a collection of ints or doubles. Since we can't do that, Java gives us something very close. In addition, Java tries to make the mixing of the primitive type with its wrapper type largely seamless. One common bug is that if you're checking for equality between two Integer objects, you must use the .equals method instead of the == operator. The reason it's such a tricky bug is that == works for many small integer values and only stops working when comparing larger values stored in two different variables. For now, I won't dive into this sidebar, except to note that on occasion, when using the wrapper class, things don't work as they should. So one should be vigilant in testing whenever they use any of the wrapper classes.

One nice thing about the wrapper classes is that a variable of the type of a wrapper class can be set to null, so we have a way to set a variable to indicate that a reference isn't pointing to anything instead.

Strictly speaking, the parse methods don't return variables of the types of the Wrapper class anyway, but this note about Wrappers becomes relevant as soon as we look at Collections in Java.

Java Class: ArrayList
Java Collections comprise of many different classes. The first of which most students learn is ArrayList. An ArrayList object is essentially a growing and shrinking list. Here's the Java doc:

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

The default constructor creates an empty ArrayList and this is typically the one we use. Notice that we have to use a generic to specify the type of item we want in an ArrayList, but that type MUST BE a class, not a primitive. This is why the wrapper classes are so useful!!! For example, to create an array list of integers, we can do:

```
ArrayList<Integer> nums = new ArrayList<Integer>();
```

Once we create an empty ArrayList, we can add items to the end of the list. In this code segment, assume that n is an integer variable storing the number of items in the list and we're going to read in n integers from standard input via the Scanner stdin and add them to the ArrayList nums, one by one. Here's how we would do it:

```
for (int i=0; i<n; i++)
    nums.add(stdin.nextInt());
```

We can retrieve an element at a specified index via the get method.

We can remove a value stored at an index using the remove method making sure we pass it an int index. (If we pass it an Integer object, then the first occurrence of an Integer object equal to that one will be removed.) This issue only arises if we are dealing with an ArrayList<Integer>.

Though rarely used, one can set an element in the list by using the set method.

Far more commonly though, we can retrieve the number of items in an ArrayList by calling the size() method.

You'll note that many of the features are similar to an array, but the key differences are as follows:

1. ArrayList automatically resizes.
2. ArrayList can't store primitives.
3. Both retrieving an item and setting an item work differently in an ArrayList than an array.
4. Some of the ArrayList methods take O(n) time, where n is the list size, but they automatically take care of annoying issues (resizing, moving over items, etc.)

It's critical to just play around with this class a bit to get a good feel for how it works. I'll do this live in class.

Putting it all together: Sorting People by Age

Now, we'll put all of this new information together to solve the following problem:

Given input about several people, including their full name, date of birth (year, month, day) and time of birth (hour, minute), sort the people in order by age. If two people have the identical age (same day and time of birth), break ties by last name, then first name, then middle names, in the order of the middle names. No two people in the input will have fully identical names. If one person has fewer middle names than another but all items are identical, the person with fewer middle names will come first. Make all name comparisons in lexicographical order (so case matters).

The first line of the input will only store a positive integer: the number of people listed below. The information about each person follows, using 2 lines for each person.

The first line of input for each person will be their name. The name will be on a line by itself and will have at least 2 space separated tokens. The first name will be the first of these tokens and the last name will be last of the tokens. The middle names will be in between the first and last name.

On the second line for each person, there will be a date in year/month/day format followed by a space, followed by a time in hour:minute format, where the hour is in military time (0 to 23).

Here is a sample input for the program to process:

```
5
Brenda Smith
2006/9/27 2:06
Jason Ryan Byrd
2006/9/27 2:07
April Cheyenne Miller Smith
2006/9/27 2:06
Gemma Dunne
1998/1/16 21:34
April Cheyenne Smith
2006/9/27 2:06
```

The program should simply output the full names in order of birth, breaking ties as described. Thus, for this input, the program should output:

```
Gemma Dunne
April Cheyenne Smith
April Cheyenne Miller Smith
Brenda Smith
Jason Ryan Byrd
```

We'll create a Person class, which will contain a Date object and a Time object. We'll build a different Time class than the one we previously built. All three of these classes will implement the Comparable interface. Since we don't know how many tokens each name will have, we must read ALL the input using the .nextLine method. When necessary, we'll use StringTokenizer objects.

While the full program is fairly lengthy, it's included in the sample programs. In these notes we'll focus on the following:

1) Use of nextLine() method in the Scanner class and use of StringTokenizer objects.

2) Sorting mechanism

Code for SortPeople.java

In the file SortPeople.java, we just have the main program that does the key processing. It reads in the input, passes the information to the Person constructor, and adds each person into an ArrayList. This list is then sorted (via Collections.sort since an ArrayList is a Collection.) Finally, the desired output is printed out.

Here is the main method of that class:

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

	Scanner stdin = new Scanner(System.in);
	int n = Integer.parseInt(stdin.nextLine().trim());

	ArrayList<Person> list = new ArrayList<Person>();
	for (int i=0; i<n; i++) {

		// Get the next line.
		String line = stdin.nextLine();

		// We'll have to split these tokens here by white space.
		StringTokenizer tok = new StringTokenizer(stdin.nextLine());
		String mydate = tok.nextToken();
		String mytime = tok.nextToken();

		// Now create the person object.
		Person tmp = new Person(line, mydate, mytime);

		// Add them to the list.
		list.add(tmp);
	}

	Collections.sort(list);

	// Print out the sorted list.
	for (int i=0; i<list.size(); i++)
		System.out.println(list.get(i));
}
```

A few notes: The ArrayList just stores references, so index 0 of list points to the first Person object created (then the variable tmp gets destroyed), then index 1 of list points to the second Person object created, and so forth.

Also, notice that we only use one StringTokenizer here, just to split the date and time apart. The three strings that the Person constructor takes in are all multi-part strings.

Otherwise, this looks very similar to other examples we previously saw that utilize custom sorting.

Now, to really dig into the user of StringTokenizer, let's look at the Person constructor on the next page:

```
public Person(String fullname, String dob, String tob) {

      // Just for printing!
      name = fullname;

      // Create a default String Tokenizer with the whole name.
      StringTokenizer tok = new StringTokenizer(fullname);

      // Grab this.
      firstName = tok.nextToken();

      // Make this the right size and copy these in.
      middleNames = new String[tok.countTokens()-1];
      for (int i=0; i<middleNames.length; i++)
            middleNames[i] = tok.nextToken();

      // This is the last name,
      lastName = tok.nextToken();

      // The delimiter for the date is the forward slash.
      tok = new StringTokenizer(dob, "/");
      int yr = Integer.parseInt(tok.nextToken());
      int mon = Integer.parseInt(tok.nextToken());
      int day = Integer.parseInt(tok.nextToken());
      dateOfBirth = new Date(yr, mon, day);

      // The delimiter for the time is the colon.
      tok = new StringTokenizer(tob, ":");
      int hr = Integer.parseInt(tok.nextToken());
      int min = Integer.parseInt(tok.nextToken());
      timeOfBirth = new Time(hr, min);
}
```

We end up creating three separate StringTokenizer objects (but just use the same StringTokenizer reference). The first one is used to split up the full name into all of its parts and is fairly similar to the previous example shown. Notice that the other two objects have different delimiters, exploiting the formats for both a date and a time. Since the individual tokens are numeric, we have to convert these from Strings to integers.

Now, let's analyze the compareTo method in the Person class. This one's the most complicated, but it does farm off work to the compareTo for Date, Time and String, respectively.

```
public int compareTo(Person other) {

	// First tie breaker.
	int tmp = this.dateOfBirth.compareTo(other.dateOfBirth);
	if (tmp != 0) return tmp;

	// Now time.
	tmp = this.timeOfBirth.compareTo(other.timeOfBirth);
	if (tmp != 0) return tmp;

	// Last name.
	tmp = this.lastName.compareTo(other.lastName);
	if (tmp != 0) return tmp;

	// First name.
	tmp = this.firstName.compareTo(other.firstName);
	if (tmp != 0) return tmp;

	// Now go through the middle names.
	for (int i=0; i<this.middleNames.length && i<other.middleNames.length;
i++){
		tmp = this.middleNames[i].compareTo(other.middleNames[i]);
		if (tmp != 0) return tmp;
	}

	// Last and final tie-breaker!
	return this.middleNames.length - other.middleNames.length;
}
```

Real life code is often like what is written above; a bit tedious and caseworky. Here, we just had to follow the given definition and work through each of our tie-breakers, one by one. The code is still lengthy even though we make use of the natural subtraction property able to handle all three cases (greater, equal, less than) in 1 line.

Java Class: ArrayDeque

The ArrayDeque class is a restricted version of the ArrayList class. It allows for inserting and deleting items from either the front or the end of the list (and no where else). It turns out that these four methods can be offered with O(1) time for each (very fast) and that we can use this class to implement a Stack, Queue or Double-Ended Queue. All three data structures are very important in solving all sorts of problems.

Here is the Java Doc for the class:

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html

Most of what you need from this class is:

addFirst()
addLast()
pollFirst()
pollLast()
getFirst()
getLast()

(poll means to remove...)

To illustrate the use of an ArrayDeque and highlight the use of the website open.kattis.com, we'll use an ArrayDeque to solve the following problem:

https://open.kattis.com/problems/circuitmath

To solve this problem, we need to use a Stack and utilize the algorithm for evaluating postfix expressions that is typically taught in COP 3502 here at UCF. The gist of the algorithm is as follows:

Read the expression to evaluate, left to right. Whenever you read in an operand, push it to the stack. Whenever you read an operator, pop off the last two items from the stack and apply the operator to those items (in reverse order). Then, push the result back to the stack. If the given expression is valid, then at the end of the algorithm a single value will be left on the stack, which is the evaluation of the expression.

To solve this problem, we must first store the true/false values of each variable in an array. Index 0 represents A, index 1 represents B and so on. We can use subtraction of Ascii values to get to the correct array index. After we do this, then whenever we get an operand, we use the array to get its value and push it onto the stack. Whenever we get an operator, we pop the last two items off the stack and push the appropriate result back onto the stack.

The full solution to the problem using an ArrayDeque (we just call addLast and pollLast to do a push and pop from our stack. You can also just call push and pop...) One final note: since the number of tokens on the last line is unknown, we have to use nextLine() only to process input.

__Circuit Math Solution__

```
import java.util.*;

public class circuitmath {
	public static void main(String[] args) {

		Scanner stdin = new Scanner(System.in);
		int n = Integer.parseInt(stdin.nextLine());
		boolean[] values = new boolean[n];

		StringTokenizer tok = new StringTokenizer(stdin.nextLine());
		for (int i=0; i<n; i++)
			values[i] = tok.nextToken().equals("T");

		// Get the expression, set up stack.
		tok = new StringTokenizer(stdin.nextLine());
		ArrayDeque<Boolean> stack = new ArrayDeque<Boolean>();

		// Go through the expression.
		while (tok.hasMoreTokens()) {

			// Get character input.
			char tmp = tok.nextToken().charAt(0);

			// Variable add to stack.
			if (tmp >= 'A' && tmp <= 'Z')
				stack.addLast(values[tmp-'A']);

			else {

				// Special case, pop 1, push its negation.
				if (tmp == '-') {
					boolean top = stack.pollLast();
					stack.addLast(!top);
				}

				else {
					boolean op2 = stack.pollLast();
					boolean op1 = stack.pollLast();

					// Push back the appropriate answer.
					if (tmp == '+')
						stack.addLast(op1 || op2);
					else
						stack.addLast(op1 && op2);
				}
			}
		} // end while.

		// Print accordingly.
		if (stack.pollLast())
			System.out.println("T");
		else
			System.out.println("F");
	}
}
```

There are three lines to read in, so we read in all three with a single nextLine() method call. Whenever we want to add something to the stack, we call addLast. We just have to do the index math to access the value of the appropriate variable. Depending on the operator, we call pollLast() once or twice, so we split those cases, and in each case use addLast() to add the appropriate item onto the top of the stack. At the end, there will be one value left in the stack, which is the answer.