Lab 2: File Input and Output

This lab introduces how to handle files as both input and output. We're coming back to Tracery (which you implemented in Lab 1) with this assignment but instead of always reading the grammar from a file and writing the expanded grammar to the command line, we will also be able to read a grammar from the command line and write the expanded grammar to a file. To do this, you'll be learning about some of the IO (input/output) classes in Java, and how the Java concept of a `Stream` provides an abstraction layer above files and console I/O.

You will be doing the following:

- Import the correct IO packages from the Standard Library
- Process command line arguments that determine whether the program is reading/writing from the console (command line) or from a file
- Load a grammar either from the console or from a file, based on the arguments the program was given
- Write the expanded grammar to either the command line or a file, based on the arguments the program was given

Open the file `TraceryRecursion.java` that came with the lab. This is where you'll make all your changes. Every place in the file where you're supposed to make a change is marked with a comment labeled `To Do`.

Importing packages in Java

Try compiling the code we've given you. If you haven't made any changes it'll tell you it can't find the following symbols:

- `class Hashtable`
- `class PrintStream`
- `class InputStream`

But in Lab 1 we were able to use a `Hashtable`, so what's different? The problem is that we're trying to use something from the Java Standard Library without telling Java that we are. To tell Java that we're using something from the standard library, you use the `import` statement. In Lab 1 we had already included the correct imports for you. For lab 2 the compiler is telling you that you need to import these three classes in order to use them.

As an example of importing a class, to import Hashtable from the standard library so you can use it, you would put this statement at the beginning of your program:

```
import java.util.Hashtable
```

The `java.util` part is the package (a collection of classes) that `Hashtable` belongs to. To figure out the package of a class you want to import from the standard library, you look at its documentation page. The documentation for Java's Hashtable is here. In addition to all the info about where it is in the type hierarchy and its accessible fields and methods, the documentation page says what package the class is part of at the top of the page right above the class name.

To import `PrintStream` and `InputStream` you can look up the package they belong to at their respective documentation pages (here and here). Another trick to know about, though you don't really need to use it in this lab, is how to import a bunch of classes from the same package at once. To import all of the classes from a package, you can use statement:

```
import <package name>.*
```

Command line arguments

This program can take the following command line arguments:

```
-in <input file> -out <output file>
```

The `"-in"` and `"-out"` part of the arguments allow us to specify the input and output file arguments in either order. Below are described the different valid combinations of arguments.

```
$ java TraceryRecursion
```

This reads the grammar from the console and writes the expanded grammar to the console. Reading from the console is called *standard in* and writing to the console is called *standard out*. Standard in is represented by the `InputStream` object stored in `System.in`, while standard out is represented by the `PrintStream` object stored in `System.out`. We will be reading from the console using pipes and the command `cat` (in unix systems like Mac OS and Linux and gitbash for Windows). There is more about this below.

```
$ java TraceryRecursion -in grammar-story.txt
```

This reads a grammar from file `grammar-story.txt` and writes to standard out. This was the behavior you implemented in Lab 1.

```
$ java TraceryRecursion -out grammar-out.txt
```

This reads grammar from standard in and writes the expanded grammar to the text file `grammar-out.txt`.

```
$ java TraceryRecursion -out grammar-out.txt -in grammar-story.txt
```

This reads a grammar from from file `grammar-story.txt` and writes the expanded grammar to file `grammar-out.txt`. Note that since our command line arguments specify the input and output file with `-in` and `-out` you can put the arguments in either order (that is `-out` first like the example above, or `-in` first).Your code doesn't need to handle bad command line arguments. Since `-in` and `-out` are what tell the program to use files, junk arguments will default the program to reading from standard in and writing to standard out.

Processing command line arguments

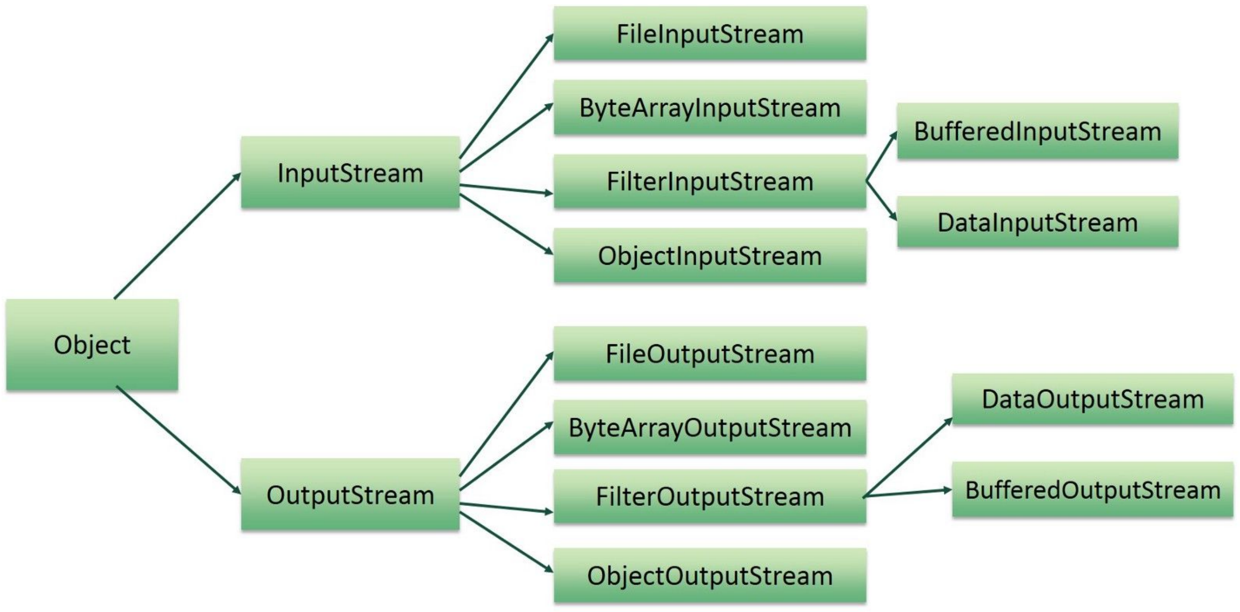
We use the command line arguments to figure out where we're reading the grammar from and where we're writing the grammar to. This is done with two helper functions that you need to write:

```
public static InputStream getInputStream(String[] args);
public static PrintStream getOutputStream(String[] args);
```

There are comments marked with `To Do` in the file to remind you to define these methods. You'll be calling these methods in `main`, passing them the command line arguments, in order to determine the `InputStream` and `PrintStream` to use.

The nice thing about these methods is that they abstract away whether you're reading/writing from standard IO or a file. The rest of the code can just read from the returned `InputStream` and write to the returned `OutputStream` without having to know or care whether its a file or the console.

To understand this, here's a bit more information about how Java abstracts input and output. You'll notice that `getInputStream()` returns an `InputStream` which is an *abstract* class (an `abstract` class, like an `interface`, can't be directly instantiated as an object). This means you will actually be returning a *subclass* of `InputStream`, where which subclass depends on whether it's the console or a file. If reading from a file, you'll be returning something called a `FileInputStream`. The diagram below shows where it sits in the class hierarchy for IO.



A `FileInputStream` is the subclass of `InputStream` for reading from a `File`. A `File` is Java's representation of a file in the computer's file system. The field `System.in` stores an `InputStream` for standard input.

To return the correct input location you'll:

- Look for a `"-in"` in `args[]`
- If there is an `"-in"` then you'll make a new `FileInputStream` that will take a new `File` named whatever is in `args[]` in the index after the location you found `"-in"`. Return the `FileInputStream`.
- Otherwise return `System.in` which is also an `InputStream`

You can assume that there will be 0, 2 or 4 command line arguments.

Now since we're dealing with user input and we don't know what exactly is in the folder with our program (from the program's point of view), we want to handle the possibility that the input file we were told to open doesn't exist. We'll want to use a `try-catch` block to catch a `FileNotFoundException` within `getInputStream()`. If you catch the exception, print the error message `"Input grammar file does not exist."` and return `System.in` (as if no `-in` argument was specified).

Our function `getOutputStream()` is very similar to `getInputStream()` but instead of returning an `InputStream` we're returning a `PrintStream` (`System.out` is a `PrintStream`).

To return the correct output location you'll:

- Look for a `"-out"` in `args[]`
- If there is a `"-out"` then you'll make a new `PrintStream` that will take a new `File` named whatever is in `args[]` in the index after the location you found `"-out"`. Return the `PrintStream`
- Otherwise return `System.out` which is also a `PrintStream`

As above, you'll want to catch a `FileNotFoundException`, print the error message `"Output file can not be created."`, and return `System.out` (as if no `-out` argument was specified).

Piping on the command line

Another part of this lab is giving you practice using pipes on the command line to use the standard output of one program as the standard input of another. For our purposes we'll be using pipes to pipe the contents of a file into `TraceryRecursion`, but in general it's possible to hook lots of programs together on the command line using pipes. The general form of using pipes looks like

```
<program 1> | <program 2>
```

where the `|` character is the pipe. The line above would takes the standard output of `<program 1>` (which would normally just print to the console) and feeds it into the standard input of `<program 2>` (which would normally read interactively from the console).

To feed a file into the standard input of `TraceryRecursion`, we'll use the the unix command `cat` (short for concatenate). This command simply reads a file and writes the content to standard out. For example, if we type:

```
cat grammar-story.txt
```

this will print out the file contents to the console:

```
animal:cat,emu,okapi
Emotion:happy,sad,elated,curious,sleepy
Color:red,green,blue
Name:emily,luis,otavio,anna,charlie
character:#name# the #adjective# #animal#
place:school,the beach,the zoo,Burning Man
adjective:#color#,#emotion#,
origin:once #character# and #character# went to #place#
```

To take this output and feed it into `TraceryRecursion` we can pipe the output of `cat` into `TraceryRecursion`:

```
cat grammar-story.txt | java TraceryRecursion
```

More generally this looks like:

```
cat <file name> | java TraceryRecursion <arguments>
```

Try running `TraceryRecursion` without a `-out` argument using pipes to feed grammar files into `TraceryRecursion`.

Loading the grammar

For this lab, the `loadGrammar()` function will be very similar to what it was in Lab 1 with one major difference: **instead of giving it a string as an argument we're passing it the `InputStream` we returned from `getInputStream()`**.

We're going to use an instance of `BufferedReader` to read from this `InputStream`. The main reason for using this class instead of the raw `InputStream` is for the function `readLine()` which allows us to read strings that correspond to a line of the file, automatically handling the fact that different operating systems using different characters to indicate the end of a line. You can read more about `BufferedReaders` here. To make a new `BufferedReader` you need to give it a new `InputStreamReader` which requires an `InputStream` as part of its constructor. You can see the details of an `InputStreamReader` here. So you'll be creating a new `BufferedReader` from a new `InputStreamReader` which is created from the `InputStream`.

Note: The constructor for `BufferedReader` wants an instance of the abstract class `Reader` as the argument. An `InputStreamReader` is a subclass of `Reader`, which is why this works.

We'll use the `readLine()` method on `BufferedReader` to read each line of the grammar and add it to the `Hashtable`. The loop where you're calling `readLine()` will have to know when to stop. Fortunately, it returns `null` when it gets to the end of the stream, so we can use this to know when the loop should stop.

Specifying that loadGrammar() throws an exception

You've probably seen at least a few `RuntimeExceptions` being thrown in your experience as a Java programmer. Among the most common are the `ArrayOutOfBoundsException` (when you're trying to access an index that isn't in an array) and a `NullPointerException` (when you try to access a field or method on a reference variable that hasn't been set equal to anything). These are unchecked exceptions, and programmers don't usually try to catch them.

The `BufferedReader` methods in `loadGrammar()` throw the *checked* exception `IOException`. Since it's checked, the compiler flags it as an error, requiring the programmer to either catch the exception or declare that the method can throw the exception. For example, to declare that `main()` can throw an IOException, you'd change the signature to:

```
public static void main(String[] args) throws IOException
```

You'll need to do this for `loadGrammar()`. When you declare that a method throws a checked exception, the compiler is happy, because now it can check that anyone who calls the method that can throw an exception either catches the exception or declares that they throw the exception in turn.

Changes in `main()`

There are two things you'll need to change in `main()`:

- Calling the methods `getInputStream()` and `getPrintStream()` to get the `InputStream` and `PrintStream` based on the arguments
- Changing the `println()` in `main()` to be called on the `PrintStream` that `getOutputStream()` returned

Outputting the grammar

Unlike in Lab 1, the method `outputGrammar()` has two arguments. Like the previous lab, the first one is the grammar we're expanding. The second is the `PrintStream` we're going to print the grammar to. All we have to do here is to call `println()` on the PrintStream argument rather than on standard output.

Example arguments and output

```
$ java TraceryRecursion -in grammar-story.txt -out output.txt
```

```
Running TraceryRecursion...
Set seed 1
```

The contents of output.txt are:

```
GRAMMAR:
adjective:              "#color#","#emotion#",
place:                  "school","the beach","the zoo","Burning Man",
emotion:                "happy","sad","elated","curious","sleepy",
origin:                 "once #character# and #character# went to
#place#",
color:                  "red","green","blue",
name:                   "emily","luis","otavio","anna","charlie",
character:              "#name# the #adjective# #animal#",
animal:                 "cat","emu","okapi",
once otavio the blue emu and luis the curious emu went to school
once otavio the elated cat and charlie the blue emu went to school
once anna the elated okapi and charlie the sad okapi went to Burning
Man
once emily the happy okapi and otavio the red cat went to school
once otavio the blue cat and emily the sleepy okapi went to the beach
once luis the green cat and charlie the blue emu went to school
once otavio the red emu and otavio the red okapi went to the zoo
once charlie the green cat and luis the green okapi went to the zoo
once luis the red emu and emily the red cat went to school
once otavio the sad okapi and luis the happy emu went to school
```

Turning the code in

- Create a directory with the following name: `<student ID>_lab2` where you replace `<student ID>` with your actual student ID. For example, if your student ID is 1234567, then the directory name is `1234567_lab2`.
- Put a copy of your edited `TraceryRecursion.java` file in the directory.
- Compress the folder using zip. Zip is a compression utility available on mac, linux and windows that can compress a directory into a single file. This should result in a file named `<student ID>_lab2.zip` (with `<student ID>` replaced with your real ID of course).
- Upload the zip file through the page for Lab 2 in canvas (https://canvas.ucsc.edu/courses/12730/assignments/39627).

Helpful tips:

Packages:

- Overview of using package members: https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html

Try-Catch:

- Overview of catch blocks in Java: https://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html
- Throwables in Java: https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html

I/O Streams

- Relationship of the various Input and Output Streams: https://stackoverflow.com/questions/22563986/understanding-getinputstream-and-getoutputstream