COSC 1P03 Lab 1
Jan. 13-17, 2014

Exercise 1
Intro to Character Reading

Estimated Time: 20 min

For the first task, we'll simply be reading lines from an ASCII data file, and handling each line as a separate character array. We'll get to 2D arrays in the next part, so we're starting simple.
For this part, simply write a program that can read the `labOneA.dat` file, one line at a time.
Refer here for documentation:
http://www.cosc.brocku.ca/sites/all/files/documentation/Brock_packages/index.html?BasicIO/ASCIIDataFile.html
Note that each returned line will be a `String` type, but also remember from lecture that `String`s have a function to return their character information as an array. Use this, and for each line (i.e. for each char array) simply print out the characters (one String per line), with each character separated by a tab (`'\t'`). Use an `ASCIIDisplayer` or `System.out`, per your own preference.

Sample Output:

```
c	o	s	c
c	o	m	p	u	t	e	r
s	c	i	e	n	c	e
l	i	n	k	e	d
l	i	s	t	s
a	r	r	a	y
h	e	a	p	s
m	a	i	n	m	e	m	o	r	y
h	a	r	d	d	i	s	k
p	o	i	n	t	e	r
r	e	f	e	r	e	n	c	e
p	a	c	k	a	g	e
c	l	a	s	s
e	n	t	e	r
i	n	t	e	r	f	a	c	e
p	r	o	t	o	t	y	p	e
p	r	i	m	a	t	i	v	e
o	b	j	e	c	t
c	o	l	l	e	c	t	i	o	n
s	u	b	t	y	p	e
k	e	y	b	o	a	r	d
```

Exercise 2
Guarded Data Access

Estimated Time: 20 min

For this task, you'll be reading information into a 2D array, and then displaying a random line of it (either `System.out` or `ASCIIDisplayer`). The catch is that you'll be selecting a random line, and a random location from which to start that line (*wrapping* off the end and back to the beginning).

To make this easier (and to separate the data access from the main algorithm), you'll be writing a *filter function*. A filter is simply a tool that should always provide you with a valid return, even if your request wasn't valid (in this case, even if you ask for an index that's a negative value, or too high). To that end, your filter function will take negative values as wrapping off of the left or top edges and back from the right or bottom (respectively), and will take excessively high indices as wrapping off the right or bottom edges and back from the left or top (respectively).

Wrapping an index that's too high is easy. The modulus operator does all of the work for you. e.g. for an array of length 13, `index%13` will always be valid for positive `index` values.

However, what about negative values? In Java, modulus does not convert negative values into positive. One could simply add the length of the array, which would work for small negative values, but -327, for example, would still not be valid for the previously mentioned array. However, there's a solution: `(index%length+length)%length` will *always* return a valid value.

Write your program to read the contents of labOneB.dat into a 2D array (Assume it will always be 25×25). Write your filter function to receive a row index, column index, and a reference to the 2D array. It should return the value at the corrected indices.
Demonstrate by randomly selecting a row from -100 to +100, and some random starting column, and then displaying 25 characters starting at that index. Then, repeat at the same row/column, but display 25 characters going to the left.

Sample Output:

```
P M R P E L E R C E S R C T A G A P O E A D T R R
P R R T D A E O P A G A T C R S E C R E L E P R M
```

Exercise 3
Guarded Access with Ragged Arrays

Estimated Time: 20 min

This task will extend the idea of the previous somewhat. Instead of accessing the contents of a regular matrix, we'll be accessing the elements of a ragged 2D array.
Wrapping wouldn't really make as much sense here, so instead we'll say that, if the provided indices are *valid* (i.e. they correlate to an actual position within the ragged array), then the appropriate character will be returned. However, if the indices are outside the bounds of the array, a *null character* (`'\0'`) will instead be returned.
Your main algorithm, depending on whether or not it receives a null character, should identify cases where the indices were invalid, and handle it accordingly.
Specifically, for this task, in addition to writing a new filter function to perform the duties specified above, you'll be writing a row-major traversal and a column-major traversal to make use of that filter function.
Instead of using a data file, hardcode your 2D array as follows:

```
char[][] data={{'A','B','C','D','E'},
     {'F','G'},
     {'H','I','J'},
     {'K','L','M','N'},
     {'O'},
     {'P','Q','R','S','T'}
     };
```

For the row-major traversal, you don't need to bother with the filter function (as a row-major traversal is *always* possible for 2D arrays in Java). So, for practice, use *for-each* loops for the row-major traversal.
For the column-major traversal, treat it as a 5×5 matrix, and use your filter function. Include control logic (i.e. use an `if` statement) to display an underscore character for invalid indices.

Sample Output:

```
Row-Major Traversal, with For-Each:
ABCDE
FG
HIJ
KLMN
O
PQRST

Column-Major Traversal, with Filter:
AFHKO
BGIL_
C_JM_
D__N_
E____
```

Exercise 4
Variable-Sized 2D Arrays

Estimated Time: 20 min

For this final task, we're going to revisit the idea of *variable-sized arrays*. Variable-sized arrays are arrays in which, because the necessary allotted capacity is not known *a priori*, you need to allow for extra space, and then keep track of how much you used. Variable-sized 2D arrays follow the exact same idea, except you need a separate counter for each dimension.

All you need to do is to read the contents of `labOneD.dat` into a 2D array, *except* you may not assume it will be any specific dimension in advance. There are different ways to approach this (the simplest is to only consider the width of the matrix at the first line, as you're allowed to assume a regular matrix). Feel free to have fun with this, and even consider reading each character in individually (with `ASCIIDataFile`'s `.readC()` function), so you can explicitly handle line endings.

Sample Output:

```
Matrix is 25 rows by 25 columns:
IKESEALEMETNIYIBNCTICYDEC
KNKEOKORIDRUMREPECNEREFER
OATNCRAKAIRTCCOEEINTPEKIT
BCEECEEARNCSDIAYTNIQCMICA
KLNJRLMTCEONNNDCFEJETTCPU
RLOENFIIUCETUNFTNTTCESCCS
LEEIAEANCPEVIRCEVNILROACL
IPCIAYRCKRMUEETYTIIELONLP
SKPCOOLEEECOJNFCACDLAEPSR
TECIAIAOFRDBCCHMPKENTERPS
SKDHFEINREOLPESTICPRSRNOC
TRRPMRPELERCESRCTAGAPOEAD
CAAITRCCCTILKPKINSSTCONSU
PROLEEYAKKURPIOSLCGDCKSMT
CIBLEPCRCEOTONTPISBSRAARF
ESYJEMCPOHYRYOIYEDTCLAEGS
DAECYTCMKMCBANYBIVDCGPREE
TOKPRIYCRDESPERLEEIRRAASC
KNUCYHCAORYMPAEACCPTATLPT
OERECTELRYLRNCREEONYAHKSH
SYFSAYOAERCOBIODCJTETMLOS
EJESNEHTPCATEEAJADTAIBIDA
BDRAROLMUSFRPMOMCYRBKCURI
CPRNJDHOIRIAIHHRNCSIFPSSP
ONTRMINEORPROTOTYPENPEDIF
```

As a final thought, consider how you could extend this idea to variable-sized *ragged* arrays. You could still use a single counter to keep track of the number of rows, but you'd need a separate counter to keep track of the number of elements within each row! The simplest way to keep track of this would be with a separate array of counters, but a dynamic data structure (covered later in the course) would usually be a better alternative entirely.