Deadline

There is no deadline. This is a warm up exercise to familiarize you with some Java classes that would be useful for your Assignments 1 and 2.

Objectives

In this assignment, you will learn how to (i) read from, and write to, a file, (ii) call an external program, and (iii) schedule a task to be executed later in Java.

Pre-requisites

You are expected to know (i) how to write simple Java programs with branches, loops, classes (including inheritence), methods, and arguments and (ii) how to compile and run a Java program in a UNIX environment. You should also be comfortable referring to Java API documents for information.

File I/O in Java

In this section, you will learn about how to read and write a file in Java.

Let's consider reading first. There are several important Java classes for this purpose. The first, is the `InputStream` class. `InputStream` is an abstract class that represents a sequence of bytes that can be read. The `FileInputStream` is a special type of `InputStream` class that represents a sequence of bytes from a file.

An `InputStreamReader` is a class that provides methods to read from an `InputStream`. When reading, the `InputStreamReader` also converts the input streams from bytes to characters. Note that this conversion is necessary since, unlike in C, each character in Java is represented by two bytes (due to Unicode encoding).

The `InputStreamReader` class performs raw input from files, and is less efficient that a *buffered* reader. A buffered reader would read a bunch of data from the disk at one time and keep them in memory. When `read` is called, a buffered reader returns the data from memory if available, or read another batch if the amount of data in memory is low. The class that corresponds to a buffered reader is `BufferedReader`.

With these three classes, you can now read from a file. The following is a typical code snippet for this purpose:

```
FileInputStream in = new FileInputStream("input.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(in));
String line;
while((line = br.readLine())!= null) {
   // process line
}
```

For writing, there is an output-counterpart for `InputStream`, called, well, `OutputStream`. You can guess that there are classes called `FileOutputStream`, `OutputStreamWriter`, and `BufferedWriter`, and you are right! The typical code to write something to a file is:

```
FileOutputStream out = new FileOutputStream("output.txt");
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(out));
bw.write("Hello World!");
```

If you write the three lines of code above in a program, however, you will find that output.txt will be created, but is empty! To understand why this happens, you need to understand how `BufferedWriter` works. Just like buffered reader, a buffered writer does not write immediately to disk, but store the content to write in memory. Thus, the string "Hello World!" would have been stored only in memory, not to the file on disk. There are two things we can do to make sure that the content in the buffer is written properly: (i) we can call the `flush()` of `BufferedWriter`. This would cause whatever content buffered to be written out to disk, (ii) if we have no use for the `BufferedWriter` object anymore, we can call the `close()` method of `BufferedWriter`. This call would clean up the internals of `BufferedWriter`, including flushing the content to disk. Try adding either one of the following two lines into your code, and you should see that output.txt will contain "Hello World!" correctly after this.

```
bw.close();
```

```
bw.flush();
```

So, remember to flush. Past experience indicates that students in CS2105 have spent countless sleepless hours debugging their programs, because they forgot to flush after they are done writing.

As an exercise, try to write a program called Copier that copies one file to another. Your program should take in two arguments: `Copier.java src deist`, and it should copy the content from `src` and write it to a new file `dest`. Your program should work for both text and binary files, and for both small files and large files (hundreds of MBs).

Calling an External Program

If you do not like to do the hard work of copying data from one file to another, you can make use of existing system command `cp` in UNIX-based OS to do it for you. Java provides an easy way to invoke an external command/program, using the `Runtime` class. The `Runtime` class provides an interface with the environment in which the Java application is running, and it allows us to do many cool things, such as finding out the amount of free memory and the number of processors. We are interested in the `exec()` method of `Runtime`, which allows us to execute an external program. For example, if I wish to invoke the `cp` command, I can use the following:

```
Runtime.getRuntime().exec("cp output.txt backup.txt");
```

There are variations of the `exec()` commands. Read the `Runtime` API and play with different variations to see their differences.

As an exercise, rewrite the Copier program to use the system command `cp` to make copies of your files.

Timer

Now, let's see how we can write a method that can be scheduled to be invoked "in the future", perhaps periodically. This effect can be achieved using the classes `Timer` and `TimerTask`.

The `Timer` class schedules a task to be executed periodically after a certain delay, while the `TimerTask` class implements the code that you want to execute periodically. For example, to print a given number after 5 seconds, and subsequently every 5 seconds, you create the following task,

```
class NumberPrinter extends TimerTask {
  int x;
  NumberPrinter(int toPrint) {
    x = toPrint;
  }
  public void run() {
    System.out.println(x);
  }
}
```

and schedule the timer elsewhere, like this.

```
  Timer timer = new Timer();
  timer.schedule(new NumberPrinter(10), 5000, 5000);
```

When you want to stop printing,

```
  timer.cancel();
```

As an exercise, extend the Copier.java program you wrote earlier to be a crude backup program that takes in the file f to back up and the location L to be back up to, and periodically (say every 10 minutes) make a copy of f into L.

THE END