National University of Singapore
School of Computing

CS2105 **Assignment 0** Semester 2 AY15/16

Submission Deadline

30 January 2016 (Saturday), 6pm sharp. 1 point penalty will be imposed on late submission (Late submission refers to submission or re-submission after the deadline for whatever reason).

Objectives

This is a warm up assignment to familiarize you with some Java classes and programming skills that will be useful for your later assignments.

This programming assignment is worth 4 marks. All the work in this assignment shall be completed **individually**.

Testing Your Programs

You are free to write your programs on any platform/IDE that you are familiar with.

However, you are responsible to ensure that your programs run properly on **`sunfire` server** because **we will test and grade your programs on `sunfire`.**

To test your programs, you may log on to **`sunfire`** using your **SoC UNIX id and password**.

a. If you don't have your SoC UNIX account, please create it here: https://mysoc.nus.edu.sg/~newacct.

b. If you forget your password, please reset it here: https://mysoc.nus.edu.sg/~myacct/iforgot.cgi.

c. If you are using a UNIX-based machine (e.g. Mac), SSH should be available from the command line and you may simply type **`ssh <SoC UNIX id>@ sunfire.comp.nus.edu.sg`**.

d. If you are using a Window-based machine, you may need to install an SSH client (e.g., "SSH Secure Shell Client", downloadable from IVLE Workbin Assignments folder) if your machine does not already have one installed. You may use the "SSH Secure File Transfer Client" software (bundled with "SSH Secure Shell Client") to upload your programs to the **`sunfire`** server for testing. A brief guide on how to connect to **`sunfire`** and upload programs can be found in IVLE Workbin Assignments folder.

If you have any question or encounter any problem with the steps discussed above, please post your message on IVLE forum or consult the teaching team in face.

Program Submission

Please submit your programs to **`CodeCrunch`** website: https://codecrunch.comp.nus.edu.sg. A briefing guide on how to use **`CodeCrunch`** can be found in IVLE Workbin Assignments folder.

Note that **`CodeCrunch`** is just used for program submission. No test case has been mounted on **`CodeCrunch`**. We will collect your programs and test them on **`sunfire`** server when the deadline is over.

You are not allowed to post your solutions in any public domain in the Internet.

Grading

Each program is worth 1 mark. Your programs (except the program of the last question) will be graded according to their correctness using grading programs.

In addition, we will deduct 1 mark for every type of failure to follow instructions (e.g. wrong program name, wrong class name).

Plagiarism Warning

You are free to discuss this assignment with your friends. But, ultimately, you should write your own code. We employ zero-tolerance policy against plagiarism. If a suspicious case is found, student would be asked to explain his/her code to the evaluator in face. Confirmed breach may result in zero mark for the assignment and further disciplinary action from the school.

Exercise 1

You are given an IP address which is a 32 characters long bit sequence (a bit is either 1 or 0). Read the IP address (from interactive user input) as a string and then convert it to a dotted decimal format. A dotted decimal format for an IP address is formed by grouping 8 bits at a time and converting the binary representation into decimal representation.

For example, IP address 00000011100000001111111111111110 will be converted into dotted decimal format as: 3.128.255.254. This is because

1. the 1st 8 bits 00000011 will be converted to 3,
2. the 2nd 8 bits 10000000 will be converted to 128,
3. the 3rd 8 bits 11111111 will be converted to 255 and
4. the last 8 bits 11111110 will be converted to 254.

To convert binary numbers to decimal numbers, remember that both are positional numerical systems, whereby the first 8 positions of the binary systems are:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Therefore, 00000011 = (0*128) + (0*64) + (0*32) + (0*16) + (0*8) + (0*4) + (1*2) + (1*1)

= 3

Name your program as **IPAddress.java** which contains only one class called **IPAddress**.

Two sample runs are shown below, with user input highlighted in **bold**.

Sample run #1:

```
$java IPAddress
00000011100000001111111111111110
3.128.255.254
```

Sample run #2:

```
$java IPAddress
11001011100001001110010110000000
203.132.229.128
```

Exercise 2

Before you start, you may read the following online article on how to use command-line argument: http://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html.

Write a program **Calculator.java**, which contains only one class called **Calculator**, to implement a simple calculator of 4 basic operations: *addition* (+), *subtraction* (-), *multiplication* (*) and *division* (/).

Parameters will be entered as command-line arguments where there should be two operands separated by an operator. The two operands must be integers. Examples of valid operands are 23, 0, -123, but not 23.5 (not an integer) or 51.0 (not an integer). The allowed operators are '+', '-', '*' and '/' only.

If the inputs are valid, the respective operation is carried out, and the result displayed. Otherwise, an error message will be displayed and calculation is skipped. You also need to handle division-by-zero error so that your program does not crash when it happens. Your program should always output the same error message **"Error in expression"** no matter what kind of error is encountered.

You may assume that the operands and the result are within the range of value of the **int** type.

Sample run #1:

```
$java Calculator 30 / 7
30 / 7 = 4
```

Sample run #2:

```
$java Calculator 30 '*' 7
30 * 7 = 210
```

(Note: to test '**java Calculator 30 * 5**' on sunfire, you need to type your command as '**java Calculator 30 '*' 5**'. Otherwise, * will be interpreted by UNIX as current location.)

Sample run #3:

```
$java Calculator -10 *
Error in expression
```

Sample run #4:

```
$java Calculator 333 / 0
Error in expression
```

Sample run #5:

```
$java Calculator 30.5 / 7
Error in expression
```

Exercise 3

Checksum can be used to detect if data is corrupted during network transmission (e.g., a bit flips from 0 to 1). Write a program **Checksum.java** that contains only one class called **Checksum** to calculate the checksum for a file **`src`** entered as command-line argument. File **`src`** should be placed in the same folder as **Checksum.java**.

You may use the **`CRC32`** class in **`java.util.zip`** package to calculate the CRC-32 checksum. Firstly, you will need to read all the bytes from a file and store them into a byte array. **`Paths`** and **`Files`** classes from **`java.nio.file`** package can help you achieve that easily. Subsequently, invoke the **`update()`** method of **`CRC32`** class giving the byte array as a parameter. Finally, call the **`getValue()`** method of **`CRC32`** class to obtain the checksum.

An example is shown below.

```
byte[] bytes = Files.readAllBytes(Paths.get("input.txt"));
CRC32 crc = new CRC32();
crc.update(bytes);
System.out.println(crc.getValue());
```

Sample run:

```
$java Checksum doge.jpg
4052859698
```

Exercise 4

Write a program **Copier.java** that contains only one class called **Copier** to make a copy of an existing file. Your program should take in two command-line arguments: **src** and **dest**, and then copy the content of **src** into the new file **dest**. Both **src** and **dest** should be placed in the same folder as **Copier.java**. You need to ensure **dest** has exactly the same content as **src**. On **sunfire**, you may use the following command to compare the content of two files - if their contents are identical, nothing will be reported:

```
cmp src dest
```

There are several important Java classes for reading and writing files in Java. Let's consider reading first. The **InputStream** class is an abstract class that represents a sequence of bytes that can be read. **FileInputStream** is a special type of **InputStream**, which represents a sequence of bytes from a file.

FileInputStream performs raw input from files. To make the disk reading more efficient, we may wrap it up using **BufferedInputStream**. The **BufferedInputStream** class reads a bunch of data from the hard disk each time and keeps the data in memory for later use. Below is an example code snippet.

```
byte[] buffer = new byte[1000];
FileInputStream fis = new FileInputStream("input.txt")
BufferedInputStream bis = new BufferedInputStream(fis);
int numBytes = bis.read(buffer); //return num of bytes read
```

For writing, there is an output-counterpart for **InputStream**, called **OutputStream**. Likewise, classes **FileOutputStream** and **BufferedOutputStream** exist and shall be used. The typical code to write something to a file is:

```
FileOutputStream fos = new FileOutputStream("output.txt");
BufferedOutputStream bos = new BufferedOutputStream(fos);
bos.write("Hello World");
```

Remember to close **bis** and **bos** at the end of your program. Your program should work for both text and binary files, and for both small files and large files (hundreds of MBs).

Sample run #1:

```
$java Copier doge.jpg mime.jpg
doge.jpg is successfully copied to mime.jpg
```

Sample run #2:

```
$java Copier Calculator.java Backup.java
Calculator.java is successfully copied to Backup.java
```