A Software Design Specification for a Subnet Calculator

Student: Andressa Pessoa de Araújo Machado
Student number: 040923007
pess0005@algonquinlive.com

Contents

Introduction

This document has the purpose of introduce and describe all functional behaviors requirements for the proper use of the software.

1.1 Goals and objectives

SubnetCalculator is a program that allows the user to get information about an IPv4 Address given its subnet mask.

The user can type in the desired IP, then the desired subnet mask and program automatically shows the network address, the broadcast address, the IP class and other relevant information.

1.2 Statement of scope

The scope of the program is to allow the user to get relevant information of an IPv4 address via terminal interaction.

1.3 Software context

Subnet Calculator was developed as a part of the Network Programing class during the 2018 Fall semester at Algonquin college. Its purpose is to apply the rules about network addresses learned in class and serve as practice and an extra activity for the class.

1.4 Major constraints

The major constraints of this application are:

- A computer capable of running the Java Runtime Environment (namely, Linux, Mac or Windows)
- Interaction via keyboard on the terminal

Data design

2.1 Internal software data structure

The data used between the classes of the program are simple Strings and the data used within a class itself are all Java types such as int, Strings, arrays of Strings, array of ints and Booleans.

2.2 Global data structure

The global structures used in this program is the Math class, used to call the pow (power) method, the Pattern class used to test the user input for validation and System class to use the arrayCopy method.

2.3 Temporary data structure

There are no temporary files being created. The methods create and use variables of the types defined in 2.1

2.4 Database description

This application does not use Databases.

Architectural and component-level design

3.1 System Structure

SubnetCalculator is a simple terminal application. It is composed of just 3 classes. The SubnetCalculator, which holds all the application logic for its goal; the InputUser which deals interaction with the user and the SubnetCalculatorMain which runs the actual application.

3.1.1 Architecture diagram

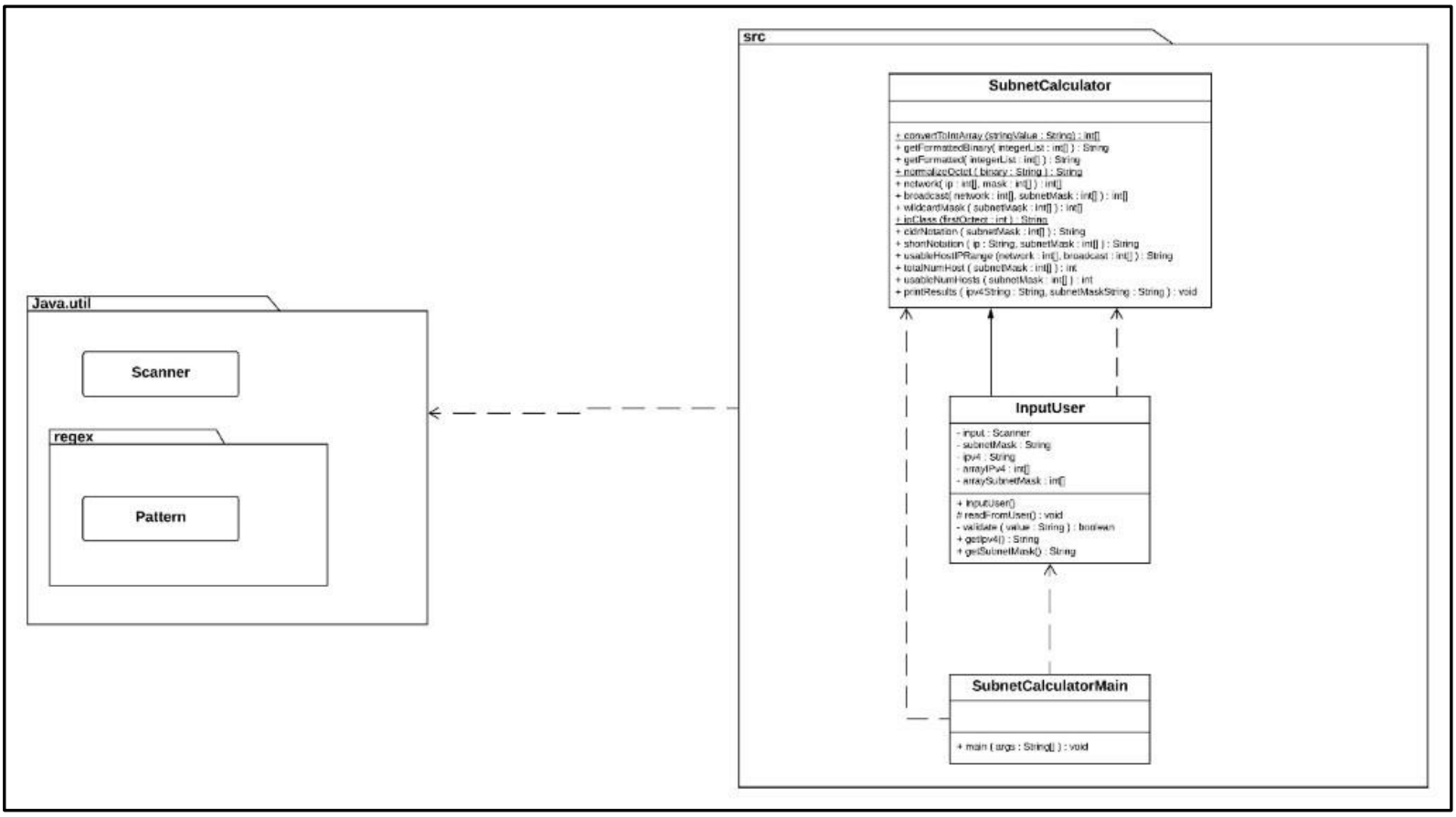

Figure 1 [1] – UML Diagram of the SubnetCalculator Software

The InputUser class is responsible for reading the input from the user via the terminal and validating that input to conform with the expected IPv4 and subnet mask formats. InputUser also has methods to return those validated inputs.

3.2.2 Component InputUser interface description.

InputOut class has a very simple interface.

InputUser():

Constructor for a new user input object.

void **readFromUser**():

It prompts the user via terminal to input the IPv4, calls the validate method to verify if the input is correct and then if that result is true (valid), it prompts the user via terminal to input the subnet mask.

boolean **validate**(String input):

It checks if the input is in the expected ipv4 format and if the values passed are within range. Returns true if format and range is valid, false otherwise.

String **getIpv4() :**

Because the variable ipv4 is private for security reasons, that method was created to return the value outside the class .

String **getSubnetMask() :**

Because the variable subnetMask is private for security reasons, that method was created to return the value outside the class.

3.2.3 Component InputUser processing detail

3.2.3.1 Design Class hierarchy for component InputUser

There is no class hierarchy for InputUser

3.2.3.2 Restrictions/limitations for component InputUser

The class InputUser can only read from user via the terminal (by keyboard). It does not treat for Unicode and special characters.

3.2.3.3 Performance issues for component InputUser

Performance was not tested for this component

3.2.3.4 Design constraints for component InputUser

InputUser had to be designed in such way that it read from the user and stored the values, so that they could be passed to the Calculator.

3.2.3.5 Processing detail for each operation of component InputUser

Constructor **InputUser**():

It initializes the global variables and the arrays created in the class and print a message when the program is initialized.

Method **readFromUser**():

It has a loop where the user is prompted for information. The user will only exit the loop if the input is in a valid format. If it is valid, the method sets its own class variables ipv4 and subnet mask to the values input by the user.

Method **validate**(String address):

Validate uses Regular Expression, Regex, to test if the entry has the format of an ipv4 address and subnet masks:

Regular expressions try to find and match patterns in a text.

This expression tries to find numbers from 0 to 255, separated by a dot, in four groups to match the format: 0-255.0-255.0-255.0-255

If that passes, it returns true.

String **getIpv4() :**

Simply returns the class String variable ipv4

String **getSubnetMask():**

Simply returns the class String variable ipv4

3.2.3.5.1 Processing narrative (PSPEC) for each operation

Since it is an interactive software, some mechanisms to get the information from the user were implemented. An input variable of type Scanner was created to receive the input from the user. A variable called ipv4 was created to store the IPv4 value entered by the user, as well as the subnet mask value, both are Strings.

Arrays were created to store the ipv4 and subnet mask values in a list in a way that each index refers to an octet to make the calculations easier to perform.

The constructor initializes those values and the readFromUser method receive those values entered by the user, call the validate method to check if the input is in a correct format and if it is in a correct format, it calls the convertToIntArray method from the subnetCalculator to convert the methods into arrays to allow the calculations to be made.

The method validate receives the value entered by the user as a parameter and check if the user entered an actual value (The program doesn't accept a null value to work with) and if the ipv4 and subnet mask values entered are in the correct format using Regular Expression. If it is correct, return true.

Because of the nature of the variables ipv4 and subnetMask, the methods getIpv4 and getSubnetMask are necessary so we can have access to the values entered.

Both variables are private for security reasons. To not allow the change of them in during run time.

3.2.3.5.2 Algorithmic model (e.g., PDL) for each operation

No complex algorithms were used, hence the Processing narrative for each component displayed below should suffice for the operations comprehension.

3.2 Description for Component SubnetCalculator

3.2.1 Processing narrative (PSPEC) for component SubnetCalculator

SubnetCalculator is the class responsible for calculating all the information needed based on the input. It can calculate network addresses, broadcast addresses, among other given the correctly formatted ipv4 and subnet values.

The SubnetCalculator does not get input for user and does not validate data. It converts a valid String to an array of ints since that data structures are easier to manipulate. Using that data structure all the other methods can be run.

Finally, the calculator can print all the information in a table format.

3.2.2 Component SubnetCalculator interface description.

The subnet calculator has the following interface:

static int[] **convertToIntArray**(String stringValue)

Given a correctly formatted ip address in a string, this method converts and returns an array of integers such as *[192, 168, 15, 1]*

int[] **network**(int[] ip, int[] mask)

Given two array of integers - the ip and subnet mask - this method calculates and returns the network value as an array of integers

int[] **broadcast**(int[] network, int[] subnetMask)

Given two array of integers - the network and subnet mask - this method calculates and returns the broadcast value as an array of integers.

int[] **wildcardMask**(int[] subnetMask)

Given the subnetMask integer array, this method will calculate the mirror of the subnetMask

static String **ipClass**(int firstOctect)

Given the first the firstOctect value as integer of the IP address, this method returns a String with that IP's class

String **cidrNotation**(int[] subnetMask)

Given the subnet mask, this method returns a String containing the slash notation for the subnet mask. (e.g. /24).

String **usableHostIPRange**(int[] network, int[] broadcast)

Given the array containing the network address and the array containing the broadcast, this method returns a String containing the first host address and the last one.

int **totalNumHosts**(int[] subnetMask)

Given the array containing the subnet mask, this method returns the total number of hosts in that network. This method includes the network address and the broadcast address in the operation.

int **usableNumHosts**(int[] subnetMask)

Given the array containing the subnet mask, this method returns the total number of available hosts in that network. This method does not include the network nor even the broadcast.

void **printResults**(String ipv4String, String subnetMaskString)

Given the ipv4 and the subnet that are been used in the operation, this method calls all the methods to perform the operation and print then in a table format to the user .

String **getFormattedBinary**(int[] integerList)

Given the integer array representing the ip address, this method will return a dot-separated, String representation of the binary values in the array. This method uses the **normalizeOctet** method to correctly build the return value.

String **getFormatted**(int[] integerList)

Given the integer array representing the ip address, this method will return a dot-separated, String representation of the values in the array.

String **normalizeOctet**(String binary)

Given a string representation of a binary number, this method will make it so that the return value has always 8 bits present, to match the format of the IP outputs. Eg: a String with "10100" passed to this method will return "00010100".

int[] **getDefaultMaskByClass** (String ipClass)

Given a string containing the class name of an IP address this method will test with if statements each class and return the array of integer representing the default subnet mask for that IP class.

int **lastFullOctetPos** (int[] ip)

Given an array of integer with an IP address this method will use a for loop to check each entry of the array and return the last index of the array which had the full octet value of 255.

int **borrowedBits** (int[] ip, int[] subnetMask)

Given an array of integer with an IP address and the array of integers with the subnet mask this method will calculate how many bits were borrowed and return an int value.

int **subnetIndex** (int[] ip, int[] subnetMask)

Given an array of integer with an IP and an array of integers with the subnetMask this method will calculate and return an int representing the subnet Index.

3.2.3 Component SubnetCalculator processing detail

3.2.3.1 Design Class hierarchy for component SubnetCalculator

There is no class hierarchy for SubnetCalculator.

3.2.3.2 Restrictions/limitations for component SubnetCalculator

The calculator does not work with IP version 6. The method **convertToIntArray** had to be made static so that InputOut could use it.

3.2.3.3 Performance issues for component SubnetCalculator

No performance tests were made for this software.

3.2.3.4 Design constraints for component SubnetCalculator

No design constraints.

3.2.3.5 Processing detail for each operation of component SubnetCalculator

static int[] **convertToIntArray**(String stringValue)

The method takes the string and since it is guaranteed that the format is correct, it first splits the String on the points and puts the results into an array of string.

Then it created an empty array of integers and for each item in the array of string, it converts each to integer and sets it in the array.

The array of integers is then returned.

String **getFormattedBinary**(int[] integerList)

The method creates a return array of integers first called *broadcast.*

To calculate the broadcast, it loops through the network array and first tests for octets with the value 255. In case we find one, we simply copy those to the return broadcast array.

When the octet value is not 255, we need to perform a calculation. First, we get the difference between 255 and the current octet value in the network array. The broadcast value is the current octet value plus the difference found in the previous step.

Then, the broadcast array is returned.

String **getFormatted**(int[] integerList)

This method creates a temporary String array variable *valueArray* to hold the String representation of each integer in the integerList param.

A loop sets each integer in integerList as a String representation (E.g. 10 becomes "10") into *valueArray*.

Then using the method String.join, it creates a String with the values on *valueArray* separated by ".", this value is then returned.

static String **normalizeOctet**(String binary)

This method first creates a temporary integer variable *numZeros* to hold the amount of zeros that will need to be added to the parameter. The variable is assigned by subtracting 8 from the length of the parameter.

Using that variable, it creates a loop and it adds a "0" to the beginning of the binary parameter for a total of *numZeros* of 0's being added. E.g.: a binary with 11111 (5 in length) will receive 8 - 5 = 3, zeros to it resulting in 00011111.

The binary param is then return.

int[] **network**(int[] ip, int[] mask)

The method creates a return array of integers first called *network*. With the two given arrays of integers it loops through them applying the Java bitwise operator & to perform the AND operation on the bits of each , such as:

result[i] = ip[i] & mask[i]

The return array will contain the correct octets for the network value.

int[] **broadcast**(int[] network, int[] subnetMask)

The method creates a return array of integers first called *broadcast.*

To calculate the broadcast it loops through the network array and first tests for octets with the value 255. In case we find one, we simply copy those to the return broadcast array.

When the octet value is not 255 we need to perform a calculation. First we get the difference between 255 and the current octet value in the network array. The broadcast value is the current octet value plus the difference found in the previous step.

Then, the broadcast array is returned.

int[] **wildcardMask**(int[] subnetMask)

The method creates a return array of integers first called *wildcard.*

To calculate the wildcard the method loops over the subnetMask parameters and subtracts the current octet of subnetMask from 255 and then sets it to the wildcard array.

The wildcard array is then return

static String **ipClass**(int firstOctect)

Given the first the firstOctect value as integer of the IP address, this method returns a String with that IP's class based in the following table:

Table 1 [2]

Class	Address range	Supports
Class A	1.0.0.1 to 126.255.255.254	Supports 16 million hosts on each of 127 networks.
Class B	128.1.0.1 to 191.255.255.254	Supports 65,000 hosts on each of 16,000 networks.
Class C	192.0.1.1 to 223.255.254.254	Supports 254 hosts on each of 2 million networks.
Class D	224.0.0.0 to 239.255.255.255	Reserved for multicast groups.
Class E	240.0.0.0 to 254.255.255.254	Reserved for future use, or research and development purposes.

It was used if to check the classes for the respective ip entered by the user based in the first octet.

String **cidrNotation**(int[] subnetMask)

The CIDR Notation give us the network ID plus the subnet mask slash notation. This method returns the slash notation for the subnet mask based on the number of zeros in the subnet mask. The method creates an int *count* to hold the return value. Then the method loops over the subnetMask parameter and it tests for the value of the octets. If an octet value is 255, the counter is increased by 8.

After the loop, the method return a String with the counter value preceded by a "/".

String **shortNotation**(String ip, int[] subnetMask)

This method simple returns the IP address value plus the subnet mask short notation.

String **usableHostIPRange**(int[] network, int[] broadcast)

This method receives an array for the network value and one array for the broadcast value. Because the address of the first host available in a network is the network address plus one, the method simply copies the network array, calling it *firstHost*, and add one to the last index of the *firstHost* array. The same have been done to the *lastHost* array which is a copy of the broadcast param, but, in this case, subtracting one from the last octet value.

Then the method builds a String with *firstHost* and *lastHost* using the **getFormatted** method with the word "to" in between.

int **totalNumHosts**(int[] subnetMask)

Because the total number of hosts is counted by taking all the hosts available bits and using then to the power of two, this method receive a subnetMask int array, create another array of strings to store the binary version (calling the **getFormattedBinary** method), it creates a local variable int to counter the number of zeros (the host available bits in a subnet mask) and uses a for loop to pass through the array counting the number of zeros. After that the method calls the pow (power) method from the Math class to do the power operation.

int **usableNumHosts**(int[] subnetMask)

Since the first host is always the network and the last host is always the broadcast, those two are not available to be used. Because of that, this method simple returns the total number of hosts already calculated by the totalNumHosts method, minus 2.

void **printResults**(String ipv4String, String subnetMaskString)

To perform its operations, this method receives two strings containing the ipv4 and subnet mask to be used in the calculations. With those values, the method calls the other methods to perform the expected behavior and print it the results in a table format for readability of the user.

To print in a table format was used System.out.format

int[] **getDefaultMaskByClass** (String ipClass)

Given a string containing the class name of an IP address this method will test with if statements each class and return the array of integer representing the default subnet mask for that IP class.

int **lastFullOctetPos** (int[] ip)

Given an array of integer with an IP address this method will use a for loop to check each entry of the array and return the last index of the array which had the full octet value of 255.

int **borrowedBits** (int[] ip, int[] subnetMask)

Given an array of integer with an IP address and the array of integers with the subnet mask this method will calculate how many bits were borrowed following the steps ahead. First it uses the **getDefaultMaskByClass** using the ip param, then that result is used to find the last full octet using the method **lastFullOctetPos.**

Knowing those two pieces of information it will then loop starting in the following octet from the last full octet (if last full octet was 1, it will start at 2) in the subnetMask array. For each octet in the subnetMask the loop will count the number of ones in the subnetMask in excess of the defaultMask to know how many bits were borrowed. That value is then returned.

int **subnetIndex** (int[] ip, int[] subnetMask)

Given an array of integer with an IP address and the array of integers with the subnet mask this method will perform many steps to find the correct subnetIndex. First it will find the IP class using the ip param first octet and the **ipClass** method; it will find the default mask using the **getDefaultMaskByClass;** it finds the network using the **network** method passing the params ip and subnetMask; it finds the last full octet position with **lastFullOctetPos** method and finds the number of borrowed bits with the **borrowedBits** method.

To start the calculation it gets the network address value in a string as the binary representation (Eg 10011000.11100010. 10011000.11100010) but removing the dots so all calculations can be done directly. Then it finds the initial position to be used to calculate , this value is equal to the number last full octet + 1, multiplied by 8. This means that it needs to start counting right after the last full octet position. Eg If there is only 1 full octet in the default mask (255.0.0.0 or 11111111.0.0.0), it should start at position 8, or exactly the first 0 of the second octet.

With that it then proceeds to copy the bits from the network binary representation starting from the initial position and counting until a number of borrowed bits is counted. That sequence of bits is converted to an integer using Integer.parseInt, using the base 2 parameter to convert from binary representation string to int and then it is returned

3.3 Dynamic Behavior for Component SubnetCalculator

This class has the method **printResults** which receives as parameters, the ipv4 and subnet mask entered by the user. To do that, in the main method, it was created a new SubnetCalculator object and a new **InputUser** object, then the method **readFromUser** is called and the Ipv4 and the SubnetMask are stored int the class variables of the **InputUser** object. Using the getters for those two class variables those values are passed to the **printResults** method.

User interface design

The interaction with the user is made using the terminal and a table format was used to make the visualization better.

To make that possible it was used just simple strings and the System.out.Format.

4.1 Description of the user interface

The terminal prompts the user for the IP and Subnet values and then display the result.

4.1.1 Screen images

The initial screen:

```
"C:\Program Files\Java\jdk1.8.0_161\bin\java.exe" ...
********** Welcome to the IP Subnet Calculator **********
This program returns a good range of values using your Internet Protocol version 4(Ipv4)

Enter your IPv4 address:
* please, include dots between different sections
```

After the user enters the IP value and Subnet Value:

```
"C:\Program Files\Java\jdk1.8.0_161\bin\java.exe" ...
********** Welcome to the IP Subnet Calculator **********
This program returns a good range of values using your Internet Protocol version 4(Ipv4)

Enter your IPv4 address:
* please, include dots between different sections
32.20.130.0
Enter your Subnet Mask:
* please, include dots between different sections
255.255.128.0
```

After the user entered valid values, the results are shown:

```
"C:\Program Files\Java\jdk1.8.0_161\bin\java.exe" ...
********** Welcome to the IP Subnet Calculator **********
This program returns a good range of values using your Internet Protocol version 4(Ipv4)

Enter your IPv4 address:
* please, include dots between different sections
32.20.130.0
Enter your Subnet Mask:
* please, include dots between different sections
255.255.128.0
You have entered the following value:
[ IPv4 Address ]: 32.20.130.0
[ Subnet Mask ]: 255.255.128.0

+------------------------------+--------------------------------------+
| Information                  |                Result                |
+------------------------------+--------------------------------------+
| IPv4 Class                   | Class A                              |
+------------------------------+--------------------------------------+
| Binary IPv4 Address          | 00100000.00010100.10000010.00000000 |
+------------------------------+--------------------------------------+
| Binary Subnet Mask           | 11111111.11111111.10000000.00000000 |
+------------------------------+--------------------------------------+
| CIDR Notation                | 32.20.128.0/17                       |
+------------------------------+--------------------------------------+
| Short Notation               | 32.20.130.0/17                       |
+------------------------------+--------------------------------------+
| Number of bits borrowed      | 9                                    |
+------------------------------+--------------------------------------+
| Subnet Mask Index            | 41                                   |
+------------------------------+--------------------------------------+
| Wildcard Mask                | 0.0.127.255                          |
+------------------------------+--------------------------------------+
| Network ID                   | 32.20.128.0                          |
+------------------------------+--------------------------------------+
| Broadcast ID                 | 32.20.255.255                        |
+------------------------------+--------------------------------------+
| Host IPv4 Address Range      | 32.20.128.0 - 32.20.255.255          |
+------------------------------+--------------------------------------+
| Total Number of Hosts        | 32768                                |
+------------------------------+--------------------------------------+
| Host IPv4 Addresses Available | 32.20.128.1 to 32.20.255.254        |
+------------------------------+--------------------------------------+
| Total Number of usable Hosts | 32766                                |
+------------------------------+--------------------------------------+

**CIDR (Classless Inter-Domain Routing)

Process finished with exit code 0
```

Restrictions, limitations, and constraints

No restrictions, limitations and constraints.

Testing Issues

No tests were created for this software

6.2 Expected software response

The software should respond to 4 cases:

1 - When the user types a wrong IP address, it should prompt for the entry of a valid one.

2 – When the user types a correct IP address, it should prompt for the entry of a subnet mask

3 – When the user types an incorrect subnet mask, it should prompt the user for a valid one

4 – When the user types the correct subnet mask, it should calculate and display all the information in a table format

6.4 Identification of critical components

The most critical component is the calculator. It implements all the expected behavior and if invalid data is passed to it, it will not perform as expected.

Deliverable

The deliverable is a zip folder containing the 3 compiled .class files.

7.1 Release notes

Version 1.0.0 of SubnetCalculator

7.2 How To's

To run the program the user needs to unzip the 3 files into a folder, then enter this folder via the terminal and type the command:

java SubnetCalculatorMain

7.3 Deliverable and Deployment

There is no deployment necessary.

References

To conclude the development of this software, the following references were used.

[1] A. Machado, Lucidchart. [Online]. [Accessed 04 December 2018].

[2] C. Hope, "IP," Computer Hope - free help since 1998, 13 November 2018. [Online]. Available: https://www.computerhope.com/jargon/i/ip.htm. [Accessed 04 December 2018].

[3] Rick Graziani, Allan Johnson, "Chapter 7 - IP Addressing," in *Introduction to Network v6*, Indianapolis , Cisco Press, 2017, pp. 413 - 448.

[4] S. Adams, "What is CIDR notation?," spiceworks , 30 August 2016. [Online]. Available: https://community.spiceworks.com/networking/articles/2495-what-is-cidr-notation. [Accessed 04 December 2018].

[5] S. Antoniou, "Simplify Routing with Subnetting: How to Organize your Network int Smaller Subnets," PluralSight, 08 November 2007. [Online]. Available: https://www.pluralsight.com/blog/it-ops/simplify-routing-how-to-organize-your-network-into-smaller-subnets. [Accessed 04 December 2018].

[6] A. unkown, "Java String join() with examples," GeeksforGeeks , [Online]. Available: https://www.geeksforgeeks.org/java-string-join-examples/. [Accessed 04 December 2018].

[7] Stephan, "How can I create table using ASCCII in a console?," Stackoverflow, 12 March 2016. [Online]. Available: https://s•tackoverflow.com/questions/15215326/how-can-i-create-table-using-ascii-in-a-console. [Accessed 04 December 2018].

[8] mkyong, "How to validate IP address with Regular expression," Mkyong.com, 07 November 2009. [Online]. Available: http://www.mkyong.com/regular-expressions/how-to-validate-ip-address-with-regular-expression/. [Accessed 04 December 2018].