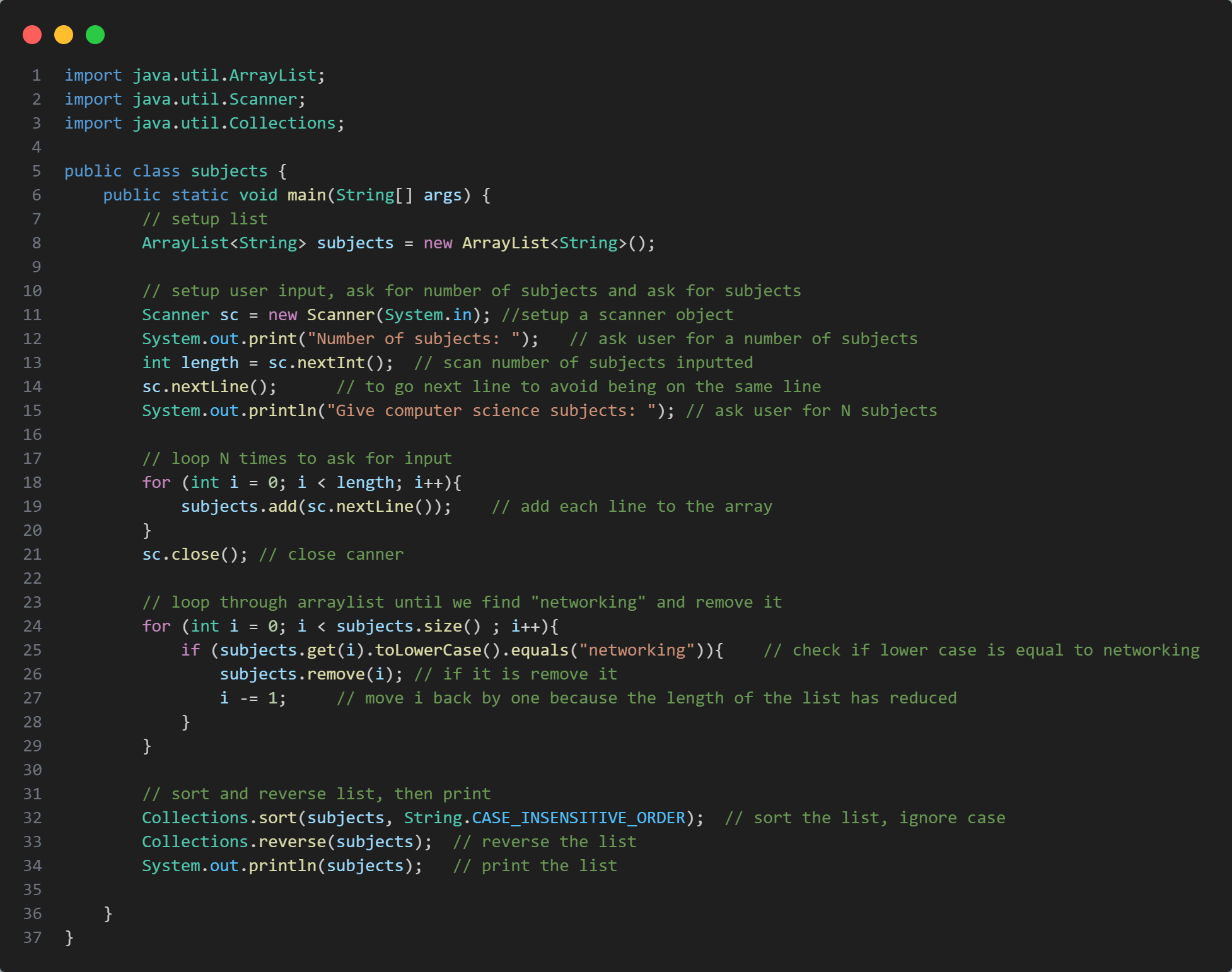
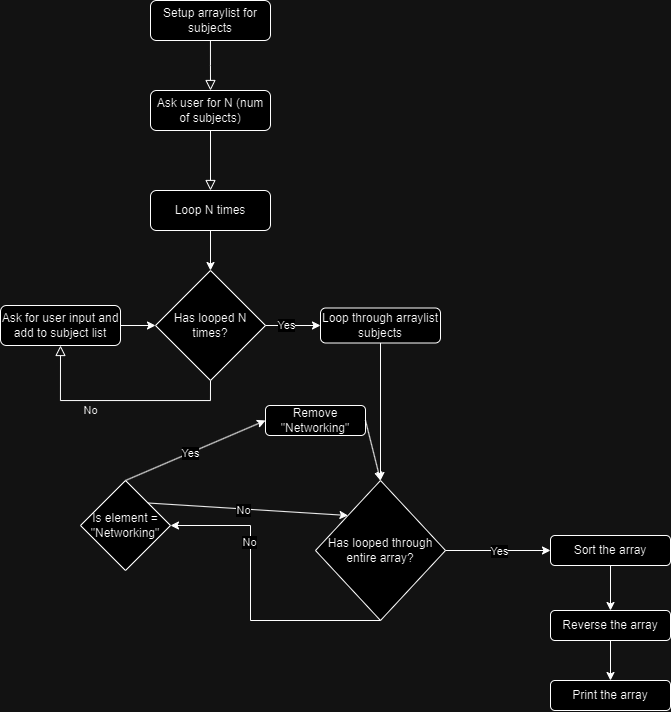
**Problem 1**

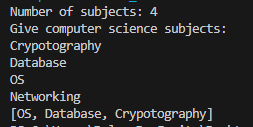
Code



Logic

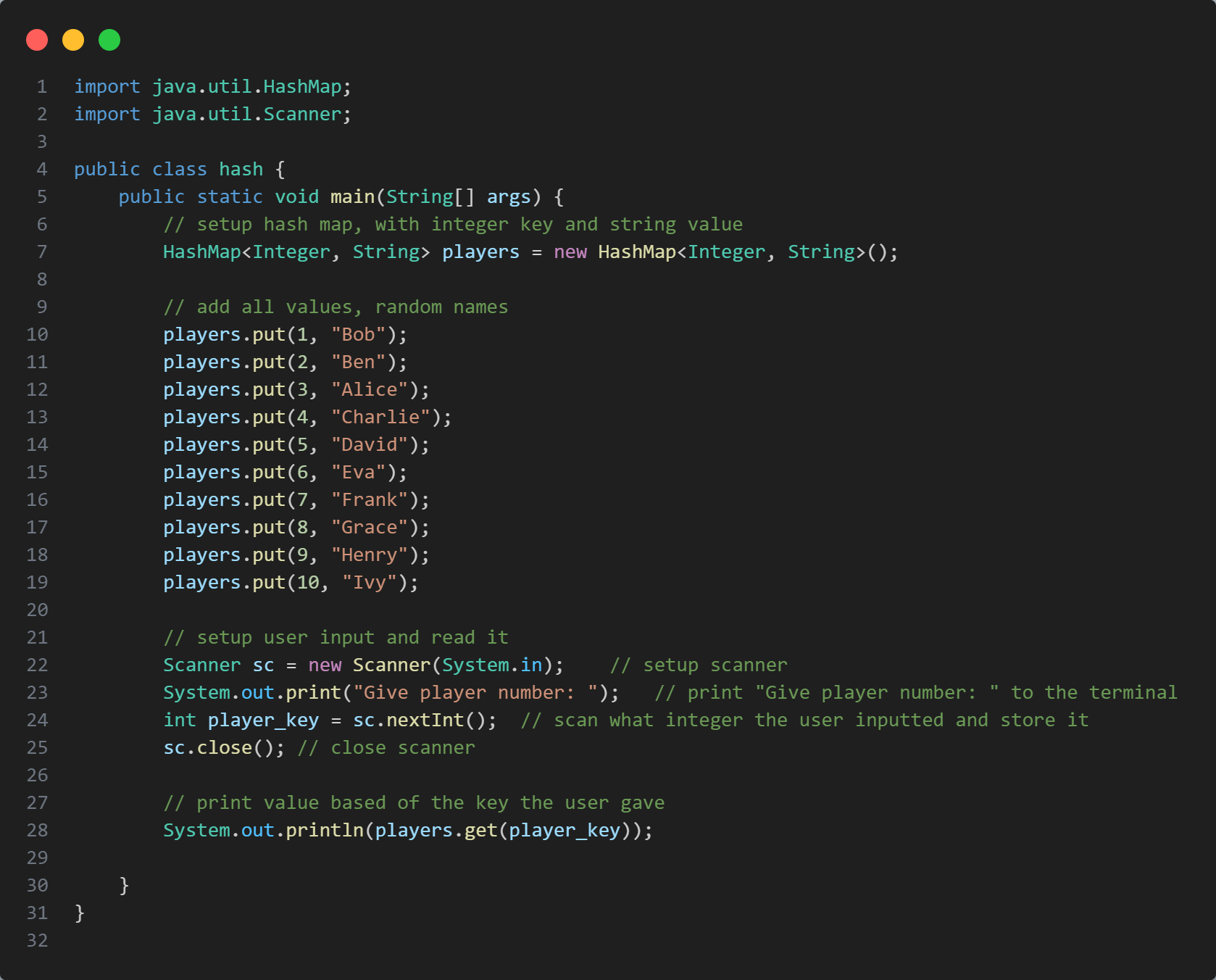
****

Results

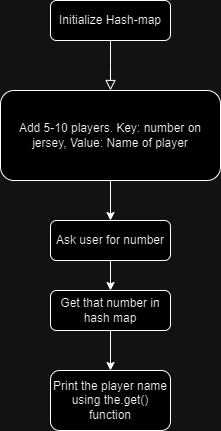


**Problem 2**

Code

****

Logic

****

Results

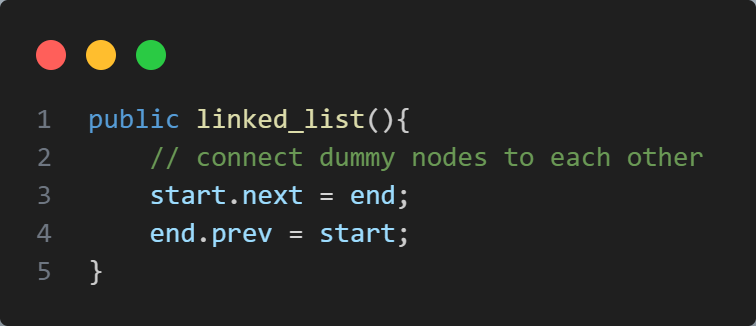


**Problem 3**

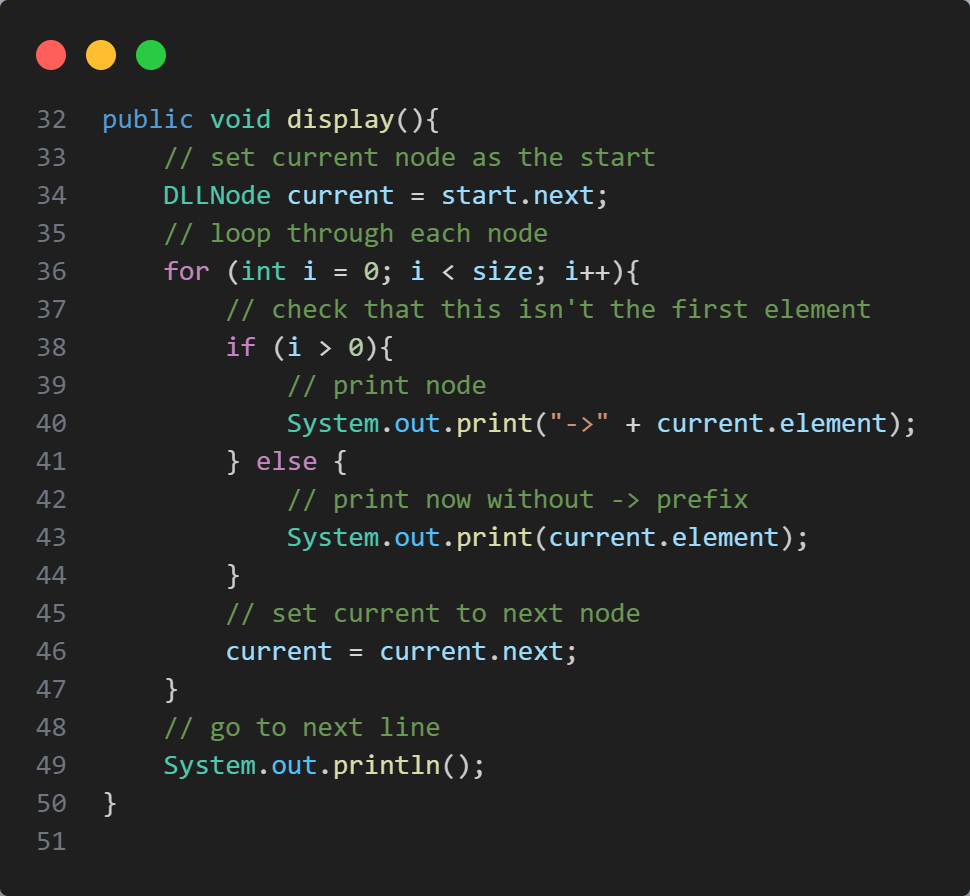
Code + Explanation



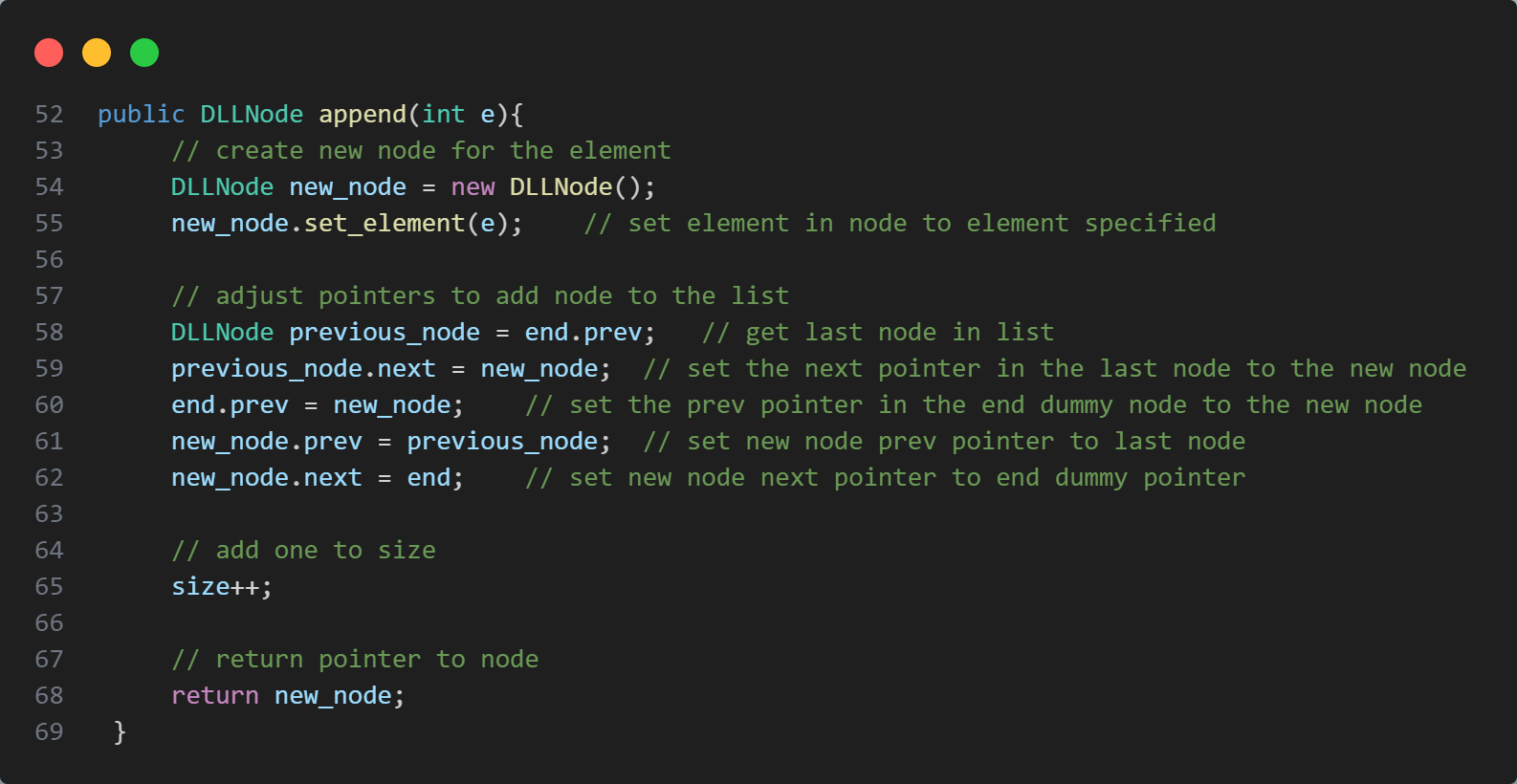
**­­­­**First, I initialise the start, end and size instance variables, to keep a pointer to the start and the end of the list.

****

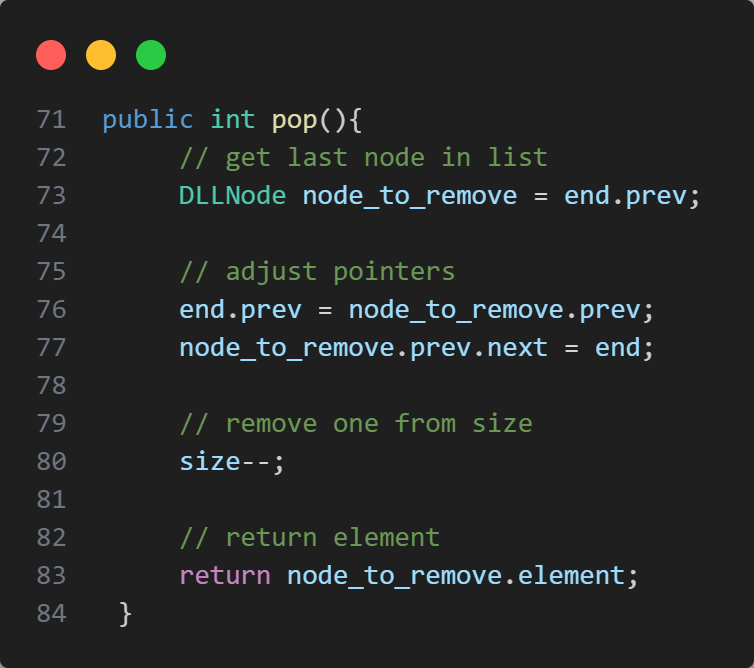
In the constructor, I connect the start and end nodes together, using their internal pointers



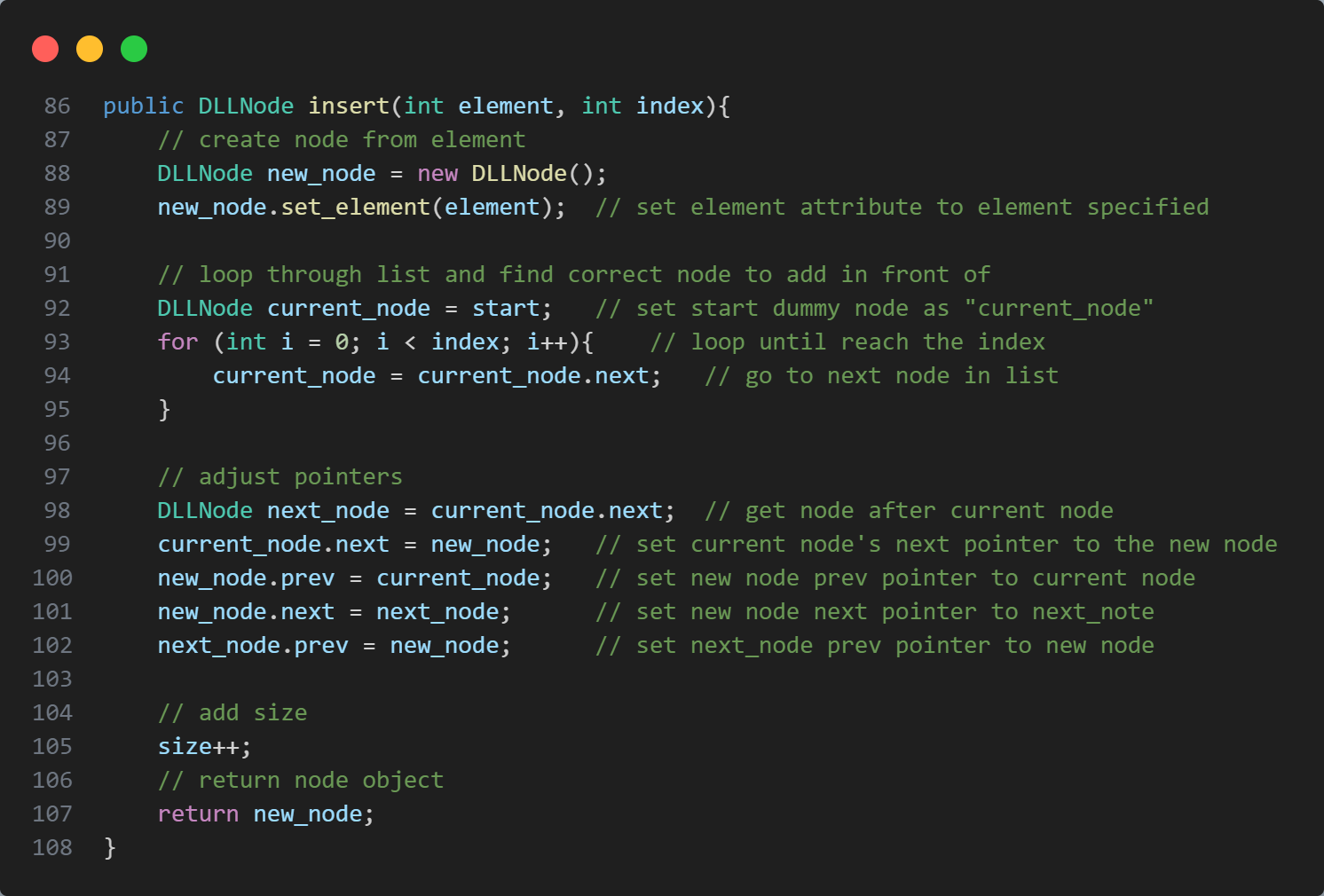
The display function simply loops through all of the connected nodes, starting at the one in front of the start node, printing its contents, then updating the current node to the next node, and so on. This function is O(n) time complexity, where n is the length of the list.



This “append” function adds an item to the end of the list. It first creates the node, with the specified element. It then gets the last node already in the list, and changes its pointers add slot in the newly created node. It then adds one to the size counter, and returns the reference to the newly created node. This is done in O(1) time complexity

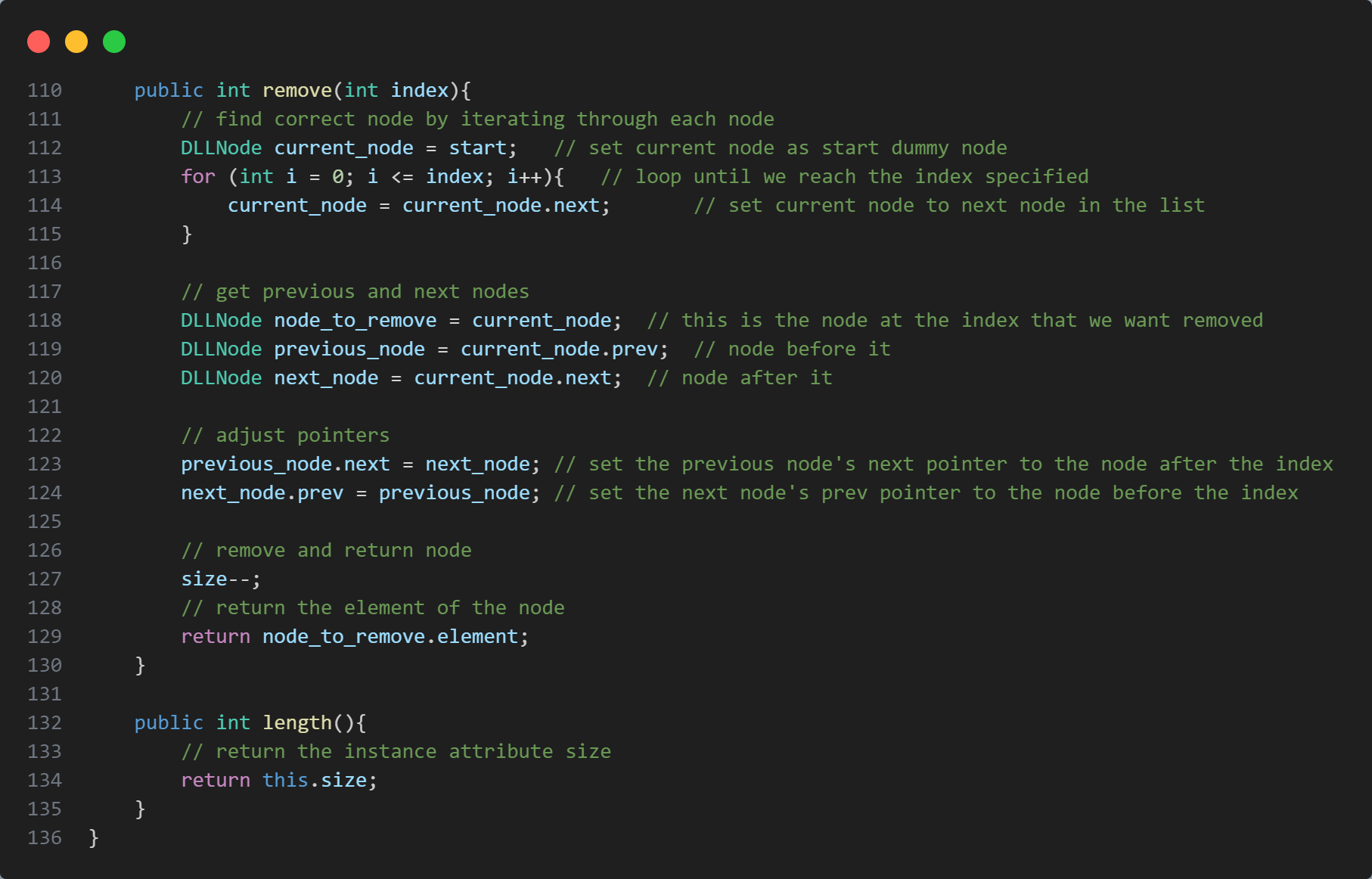


The “pop” function removes the last element in the list. It first gets the last element in the list and stores it. It then updates the pointers on the last dummy node to point to the 2nd last node. We then update the “next” pointer on the 2nd last node to point to the end dummy node. We then remove one from the size variable and return the removed node object. This is done O(1) time complexity.

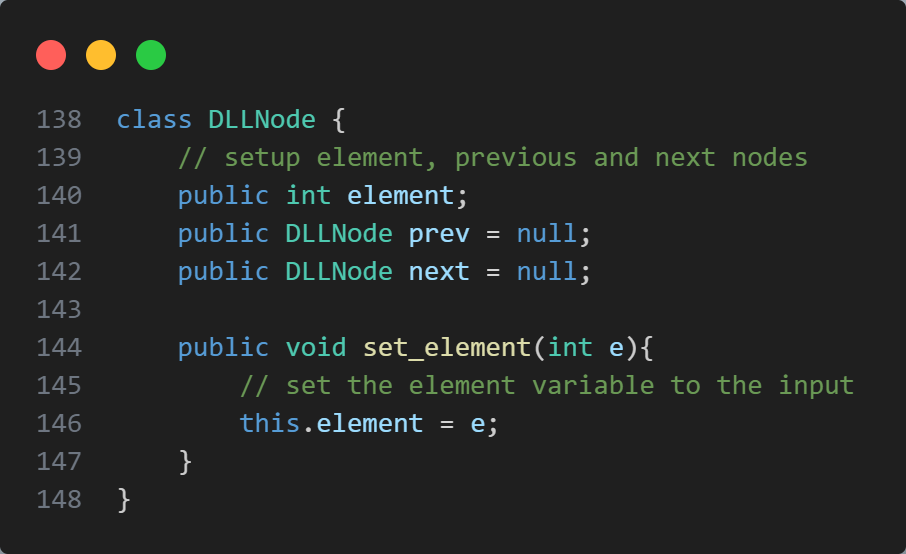


The “insert” function is similar to the “append” function, however it’s able to specify which index to add the node to. The only drawback is that this is much less efficient then “append”, as it must first loop through up to n elements. Therefore, this is O(n) time complexity.

The function first creates the node with the specified element. It then loops through all the connected nodes until it reaches the correct index. It now has the correct reference the index node and can now adjust the pointers around it to slot in the newly created node. It then adds one to the size variable and returns the new element.



The “remove” function takes in an index number and then removes that element from the list and returns it. It first searches through the list for the specified index, which makes this an O(n) time complexity function. Once it’s found it, it adjusts the pointers, by setting the previous node’s “next” pointer to the one in front, and setting the next node’s “prev” pointer to the one in the back. It then removes one from the size variable and returns the element of the removed node.

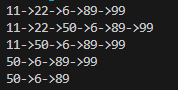


This is simply to show you what the “DLLNode” class looks like. It simply contains 3 instance variables. “Element” which stores the value, “prev” which is a pointer to the previous node, and “next” which is a pointer to the next node. It also contains a “set\_element” function which sets the element.

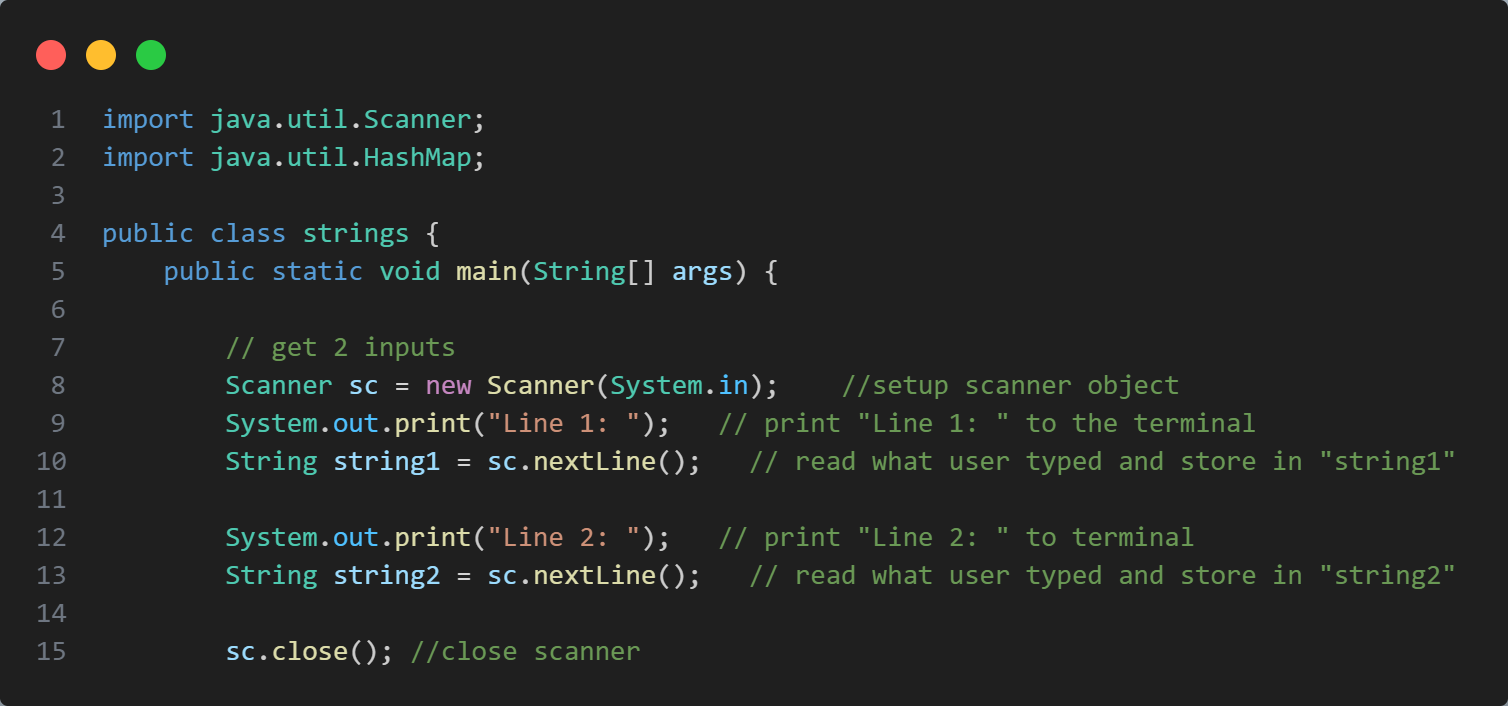
Results



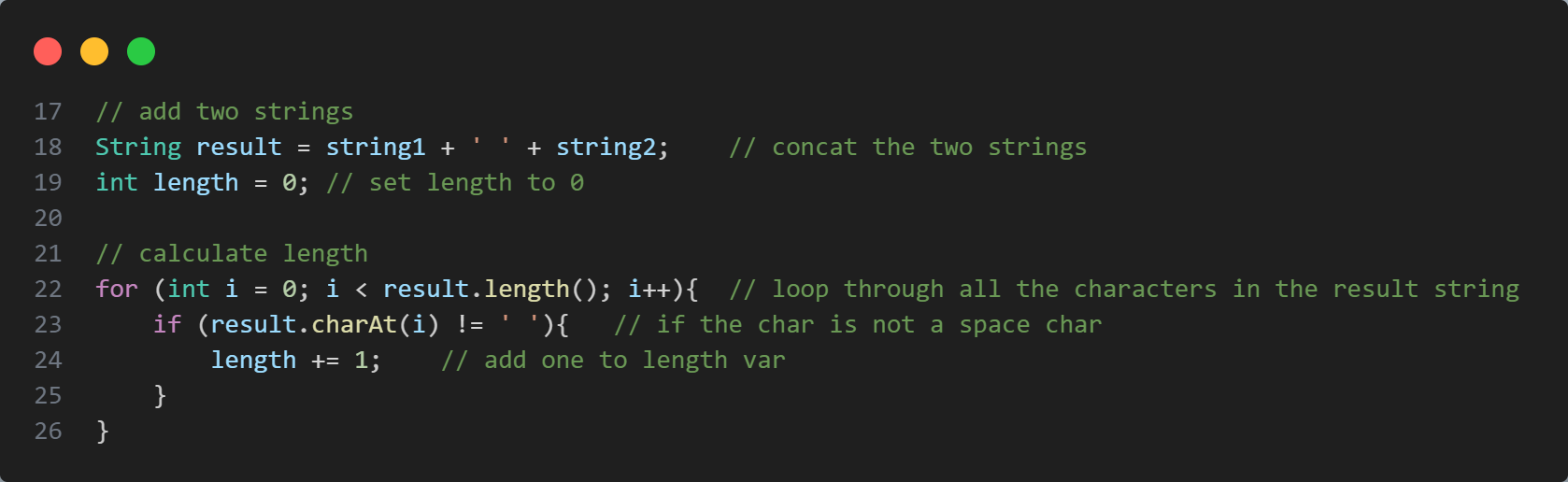
When this “main” function above is ran, this is the output:



**Problem 4**

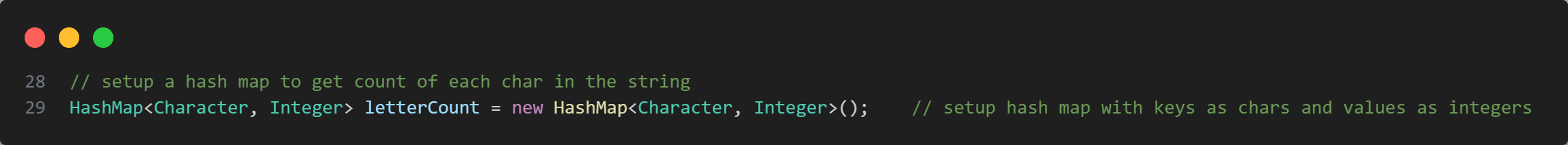
****

At first, I setup the scanner and ask the user for two strings, which are stored as “string1” and “string2”. The scanner is then closed because my IDE was bugging me about it.

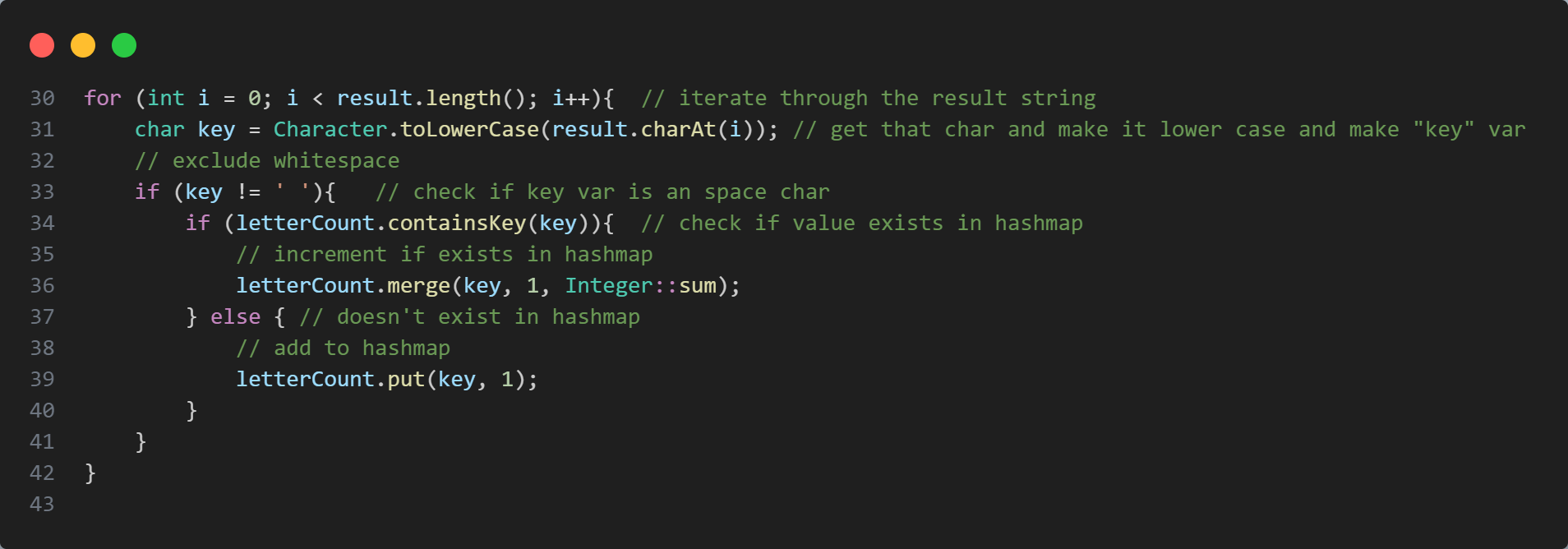


I then add the two strings together through string concatenation, and store the result in the “result” variable. I initialise an integer variable called “length”, which will store the length of the “result” string (excluding white-spaces). I then loop through and check if the current character is a whitespace character. If it’s not, I add one to the length counter.

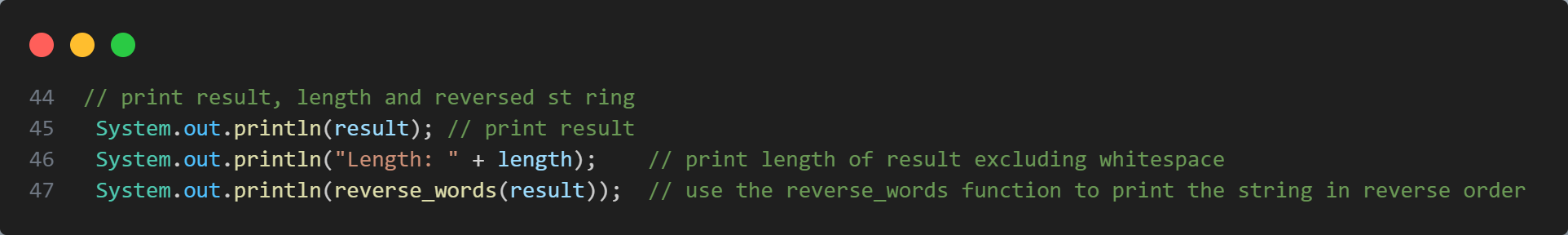
The reason I don’t simply use the “length()” function on the string is because this would count in white-space characters, whereas my method does not.



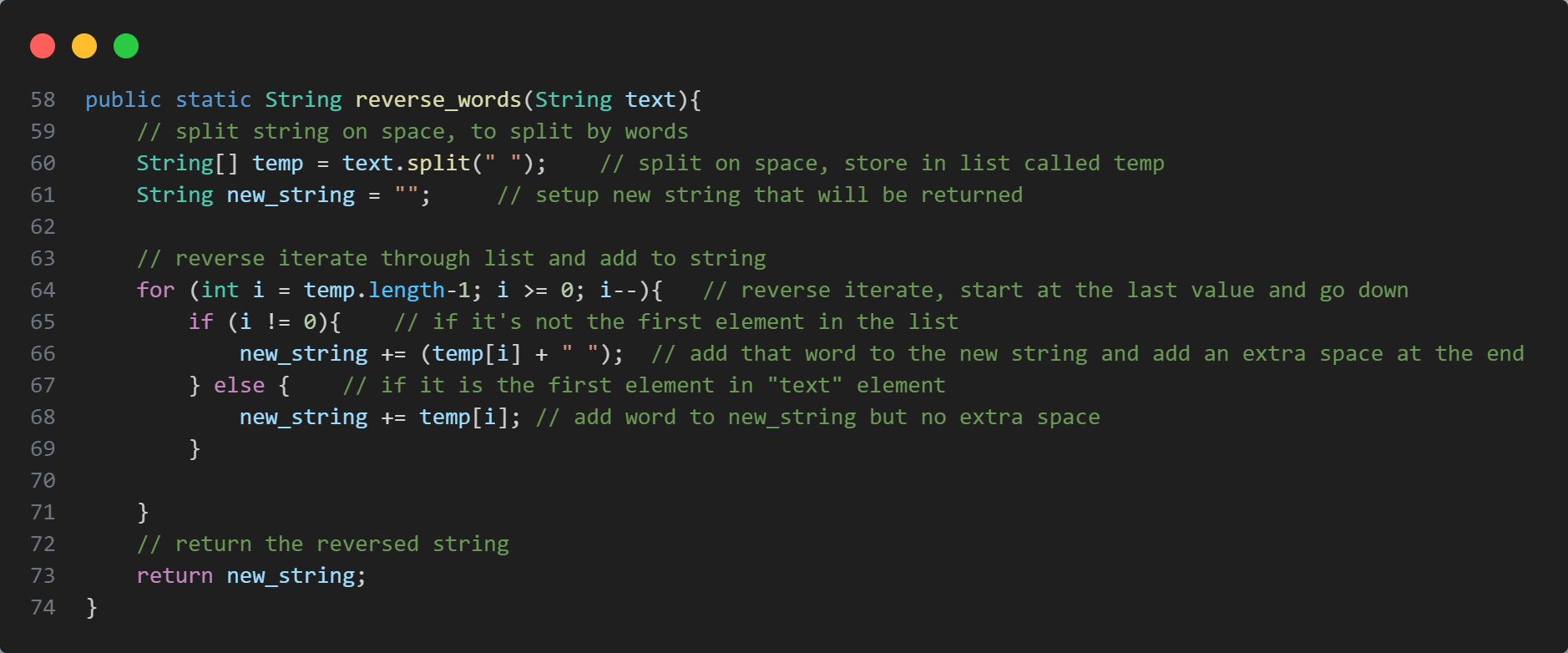
I initialise a HashMap, with a character as the type for the keys and Integers for the type of the values. This HashMap will be to count each character and how many times it appears in the “result” string. We will need to loop through the string.



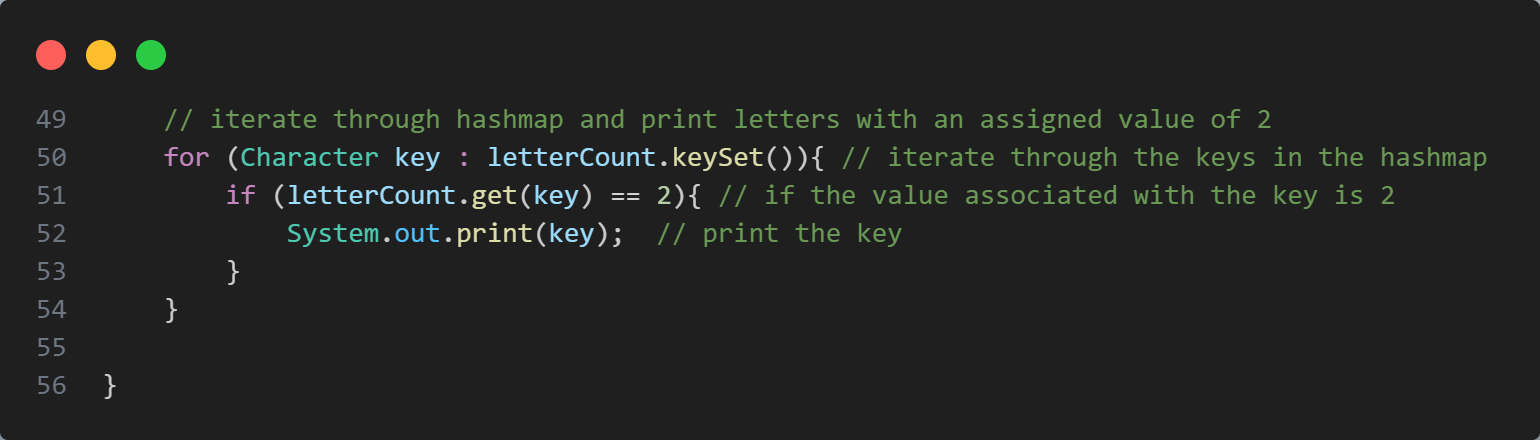
We loop through the “result” string and change each character to lowercase, as we aren’t differentiating between upper and lower case. We exclude whitespace by checking if the character is a space character, and adding it if it isn’t. I also check if the character is already in the HashMap. If it is, I use the “merge” function to increment the existing value.



I then print the “result”, the length of “result” and the reverse of the “result” string, using a function I made, at the end of the program:



This function takes in a String “text” as input and uses regex to split the string on space, giving me a list of each word. I then iterate through that list of words, however in reverse order. I start at the last element of the list and iterate downwards. I add each word to a string. I also check if this word is the last element of the list, if it is, I simply don’t add an extra space at the end. I then return the reversed string.



The final thing I do in this program, is iterate through the keys in the HashMap and check if the value associated with that key is two, and if it is, print it.

Results:

