Overview:

This coursework asked me to create a program that creates and validates individual blocks of sudoku puzzles. Particular emphasis was placed upon the use of loops and two-dimensional arrays and with following best practice regarding the use of types and names with object attributes. The use of inheritance also played an important role in the coursework.
My solution successfully incorporated all parts of the specification.

Code Design:

For my code design, I mainly followed the suggested approach given by the specification. Thus I ended up with five classes, these being Assignment2, Block, TextualBlock, Cell, and TextualCell.

The Assignment2 class contained a main method and two helper methods. The main method simply prompts the user to enter a block type and then uses a switch case to handle the user input. The only valid inputs that the user may make are “t” and “n”, any other inputs trigger the default case in the switch case which simply outputs an error message to the user. If the user does enter a correct input, one of the two helper methods are called. Both helper methods are essentially the same. Each of them creates a block and then runs this block’s initialise() method. The only difference is that the textualPuzzle() helper method creates a TextualBlock object instead of a standard Block object which the numericPuzzle() method creates.

Most of the code in this project is contained within the Block class. This class contains several attributes, the most important of which is the cellArray which is a two-dimensional array of Cell objects whose dimensions are dictated by the blockSize attribute (initially set to three). It was decided against making the blockSize attribute static or final because it would need to be altered for TextualBlock objects. The class also contains two booleans (formatValid and structureValid) and two Strings (formatErrorDescription and structureErrorDescription). These attributes are used to notify users about potential mistakes or errors in their input to the program.

The most code intensive parts of this class come in the forms of the intialise() and the checkStructure() methods. In the initialise() method, the user is first prompted to “Enter cell values:”. The user must then enter one row of cells at a time with each cell’s value separated by commas until the cellArray object is full. A for loop is used to handle inputs and to ensure that the user doesn’t enter too many lines. As the user inputs their desired cell values, the method employs several if statements to check if the formatting of the block is valid. The first of these uses the hasNextLine() method from a Scanner object to determine if the next line exists or not. If hasNextLine() returns false, validFormat is set to false, formatErrorDescription becomes an error message which describes the problem with the input and the for loop is exited. A similar process occurs each time an error is caught. If hasNextLine() returns true, the String.split() method is used to split up the user input into a String array. The length of this array is then validated to see if the user entered in a row with the correct number of cells in it. If this fails the error process described above is run. If this test is passed, the cellArray is finally populated with the user input values. To do this, the class uses the makeCell() method which creates a Cell

object given a String input. The String input is taken from the array created by String.split() earlier in the program and the Cell objects are put into the cellArray. This is accomplished using another for loop. Finally, the method uses each Cell object's validValue() method to test if each cell has valid value. If not, the error process described previously is once again used. This process is then repeated until cellArray is full. After this, if the user's block format is completely correct, the method will finally output "Valid format" and then call the checkStructure() method. If not, the method will output the structureErrorDescription. For loops were used in this method in order to minimize code repetition, while many if statements were needed due to the many errors which needed to be caught by this method.

The checkStructure() method's job is to check that no values repeat in the user inputed block. The method first uses a series of nested for loops to check if any values are repeated in any rows or columns. If they are, a similar error handling process as the one inside the initialise() method is initiated. The only difference between these two processes being that the one in checkStructure() sets validStructre to false and changes structureErrorDescription to a new error message. At the same time, the method populates a one-dimensional Cell array with all the Cell objects from cellArray. For loops are then used with this newly created one-dimensional array to determine if any Cell values are repeated across the whole block. Once this process is finished, the method either prints "valid structure" or the structureErrorDescription depending on whether validStructure is true or false. For loops and if statements were once again heavily used in this method for the same reasons as with the initialise() method.

The TextualBlock class is a subclass of the Block class. There are two principal differences between the two. Firstly the blockSize and thus the cellArray's dimensions in the TextualBlock class is set to five instead of three. Secondly, the makeCell() method is overridden to create a TextualCell object instead of a Cell object. These changes were made to fit in line with the specification and to fulfill the goal of practicing the use of inheritance. Furthermore, by using a subclass, the initialise() and checkStructure() methods could be reused for a different puzzle format, reducing the amount of code that needed to be written.

The last two classes of the program are the Cell and TextualCell classes. Each Cell object has a single attribute, its value. This value can be accessed through the getValue() method in the Cell class. Both the Cell and TextualCell classes have a unique validValue() method, which simply checks if their value aligns with the valid values described by the specification. In the Cell version of the method, a for loop is used to simply check if the value of the Cell object falls between 1-9 inclusive, while in the TextualCell version of the method, ascii numbers are used to check that the value falls between capital A and capital Y inclusive. These methods are used in the Block class to validate the values of each cell. By creating a Cell class instead of just using a String array in the Block class, different types of sudoku puzzles with different rules could be more easily accommodated without the need to alter much code in other classes and with the added benefit of being able to reuse already written code.

Testing:

Initially, I checked that my code passed all 23 of the stackschecks tests. A more detailed version of the stacscheck can be found in the output.html file included in the submission.
Additionally, I tested for several other possible inputs not included in the stacscheck tests. A summary of these tests is included in the table below.

Test	Input	Expected Output	Actual Output
Invalid block type 1	h	Invalid block type.	Invalid block type.
Invalid block type 2	N	Invalid block type.	Invalid block type.
Out of range (lower bound) 1	n 0,1,2	Invalid format: value: 0.	Invalid format: value: 0
Out of range (lower bound) 2	n 1,3,4 2,7,-9	Invalid format: value: -9.	Invalid format: value: -9
Numbers as words	n 1,two,3	Invalid format: value: two	Invalid format: value: two
Ludicrous whitespace 1	n 1, 3, 4 2,9, 8 7,6, 5	Valid format, valid structure.	Valid format, valid structure.
Ludicrous whitespace 2	t Q , W,E,R,T Y , U,I, O,P A, S, D,F,G H, J, K,L,X C,V,B,N ,M	Valid format, valid structure.	Valid format, valid structure.
Lowercase letters	t A,B,C,D,e	Invalid format: value: e.	Invalid format: value: e.
Numbers in textual	t 1,2,3,4,5	Invalid format: value: 1.	Invalid format: value: 1.
Textual invalid format row too short	t Q,W,E,R	Invalid format: number of cells in row.	Invalid format: number of cells in row.
Textual invalid row too long	t Q,W,E,R,T,Y	Invalid format: number of cells in row.	Invalid format: number of cells in row.
Textual invalid missing row	t Q,W,E,R,T Y,U,I,O,P A,S,D,F,G	Invalid format: number of cells in row.	Invalid format: number of cells in row.

	H,J,K,L,X		
Textual invalid not enough rows	t Q,W,E,R,T Y,U,I,O,P A,S,D,F,G H,J,K,L,X	Invalid format: number of rows.	
Textual invalid format and invalid value	t Q,W,E,R,T Y,U,I,O p,A,S,D,F,G	Invalid format: number of cells in row.	Invalid format: number of cells in row.
Textual invalid value and invalid format	t q,W,E,R,T Y,U,I,O P,A,S,D,F,G,H	Invalid format: value: g	Invalid format: value: g
Numerical valid	n 9,8,7 6,5,4 3,2,1	Valid format, valid structure.	Valid format, valid structure.
Textual valid	t Q,W,E,R,T Y,U,I,O,P A,S,D,F,G H,J,K,L,X C,V,B,N,M	Valid format, valid structure.	Valid format, valid structure.

Figure 1: Table of Additional Tests Performed

The only test which did not behave as expected was the “Textual invalid not enough rows” test. This occurred because the code simply waited for another line to be input instead of producing an output. The specification mentioned that this would happen, so this does not necessarily call into question the validity or correctness of the code.

The automatic stacscheck tests along with the additional testing is enough to ensure that the code behaves as it is intended to. The automatic stacscheck tests check for valid blocks in both the textual and numerical format, correct structure and format for numerical blocks and correct structure for textual blocks. The additional tests check for invalid block type inputs and correct format for textual blocks. Both the stacscheck tests and my additional testing also test for a range of valid and invalid values for cells. These include, words and numbers that are too large or too small in numerical blocks and lowercase letters, numbers and out of bounds letters in textual blocks. In addition, both test for ignorance of whitespace. With all these tests completed, we can be reasonably sure that the program behaves as intended.

Evaluation:
My solution to coursework was able to fulfill all parts of the specification. It successfully created and validated blocks of sudoku puzzles in both numerical and textual formats as described in the specification. After testing was completed, the program seemed to reliably behave as expected for essentially all inputs.

Conclusion:
Overall, by following the suggested approach, this coursework did not feel particularly difficult although it did seem more complex than the previous coursework. In my opinion, the most difficult part of the coursework was writing the initialise() method in the Block class. While it ended up functioning as intended, due to the number of things that needed to be validated by this method, my solution's code felt a bit bloated. With additional time, I would've liked to find a more efficient way to run the initialise() and checkStructure() methods and maybe even break them up into smaller, more manageable methods. Furthermore, I think it would be an interesting challenge to expand this program so that it can validate entire sudoku puzzles instead of just one block of a puzzle.
Additionally, I observed an interesting behavior in which, when more than the maximum number of rows for cellArray were input at the same time (this could be done through copying and pasting), the program would not read the extra rows. This meant that invalid blocks could be input and the program would output that both the structure and the format would be valid. Examples of this can be found in the table below.

Input	Output
n 1,2,3 4,5,6 7,8,9 h,-89,100,:)	Valid format, valid structure.
t Q,W,E,R,T Y,U,I,O,P A,S,D,F,G H,J,K,L,X C,V,B,N,M 1,5,h,Z,z,0,4	Valid format, valid structure.

Figure 2: An interesting behavior

With additional time, I would've also liked to create a solution to this. This could be done by making the program count the total number of rows input and throwing an error message if it exceeded the number outlined by the blockSize attribute in the Block class.