As well as making a program that did all that was required, I had some extra aims:

- keep the user informed
- avoid un-necessary dialogs
- trap as many types of user-input error as possible

The program starts by calling a method that uses a `javax.swing` dialog to welcome the user and ask him or her to enter his or her name. The user's name is passed to a `do/while` loop that generates a menu that persists so long as the user doesn't choose to end the game. In this loop, a switch statement uses the user's choice to invoke the appropriate action or gently chide the user if invalid numeric input is made. (Sequential `if` statements would have needed something like `if (choice < 0 || choice > 6) { //chide } // end if`) As well as choosing a game-play mode, the user can view instructions (just a formatted textbox) or end the game.

In the main 1-D game method, the computer's integer is generated by populating a 1-D array with random digits: these are generated using the standard `Math` class. Validity is achieved by running the population step within a `do/while` loop that ensures the digits are unique and that the 1st element of the array is not zero.

The user is asked to enter an integer. This integer is then broken down into individual digits which are put into a 1-D array. For example, if the input is `246`, '2' is obtained by dividing the integer by $10^{\text{integerlength} - 1}$. '4' is obtained by subtracting $2 * 10^{\text{integerlength} - 1}$ from the original input, then dividing the remainder by $10^{\text{integerlength} - 2}$ and so on.

Number-of-digits validity is tested by checking whether the integer falls between bounds calculated from the length of integer. Unique-digits validity is tested using the method used when generating the computer's integer. If the input is invalid, the user is informed (if both types of invalidity occur, these are reported in the same dialog) and offered the choice of going back to the menu[1] or trying again. The flow here is controlled by two Booleans (for validity of try and desire not to retry) – if they are both `true`, the program escapes a nested `do/while` loop (which would otherwise allow the user to try inputting again) and moves on to input analysis.

If there is valid input, the number of rounds is incremented. The input is then tested for bullseyes (`user's array[x] == computer's array[x]`) and hits (`user's array[x] == computer's array[anything but x]`). The bullseye test uses a simple `for` loop. The hits test has to ignore bullseyes: hence each element in the computer's array is in turn omitted from the items to be tested against.[2] Hence within the main `for` loop there are nested successive `for` loops to test both 'before' and 'after' the current 'bullseye position'.

The scores are passed to a method that either calls another method to congratulate the user on getting the maximum number of bullseyes and breaks back to the menu or displays the score and asks if the user wants to carry on. The user's wish is returned to the main 1-D game method as a Boolean. Not wishing to carry on escapes the main `do/while` loop: the user is taken back to the menu. Otherwise the program returns to the start of the `do/while` loop.

The 2-D game works in a similar manner but has to allow for there being more than one integer.[3] The integers are stored as rows in 2-D arrays. The array containing the computer's integers is generated by calling the 1-D random array generator once per row/integer. This automatically ensures that the first digit of each integer was not zero.[4]

Within another `do/while` loop, the user is then asked for input. Row by row, the user input is broken into digits (similar code to the 1-D game) which then populate a 2-D array. Number-of-digits validity is tested by recreating the user's integers and passing them to the method used in the 1-D game.[5] The result for each row/integer is stored in a 1-D array: one element for each integer. Unique-digits validity is tested by extracting each row of the 2-D array into a 1-D array, which is then passed to the 1-D unique-digits method. Validity results are stored in another 1-D array. [6]

If any element of the validity 1-D arrays is invalid, the whole input is invalid and the results-arrays are used to generate strings for each row/integer that has failed a validity test. These strings are all then displayed in one dialog which also offers the user another chance to input valid integers. If the user wishes to try again, the program returns to the start of the inner `do/while` loop. If the input was valid, the game moves on to testing for bullseyes and hits.

To test for bulleyes, each row of the 2-D input array is tested using the method from the 1-D game. The results are stored in a 1-D array: one element per row/integer. If the user has achieved the maximum number of bullseyes, the number of rounds taken is displayed, the user is congratulated and the program breaks back to the menu. Otherwise, rows are tested for hits: results are stored in yet another 1-D array. The two results arrays are then used to generate a report string for each row. These strings are displayed together in one dialog which offers the user another round of guessing. As in the 1-D game, choosing not to retry escapes the main game method's `do/while` loop and takes the user back to the menu, while choosing to retry takes the user back to the start of the input `do/while` loop.

1 I'd have liked the user to be able to just exit the program here but I couldn't find a way of breaking up 2 levels.
2 It may have been possible to find the number of matches, then subtract the number of bullseyes to get the number of hits.
3 With hindsight, it may well have been better to write just the 2-D game, but allow the 2-D arrays to have just 1 row for the special case of 1-D play.
4 An alternative would have been to populate the 2-D array with random digits within a `do/while` loop to keep repopulating if any first digit is non-zero. This would have required a separate test (hence extra code) such as summing the digits in the first column: if the result is greater than zero, the array is invalid.
5 It may have been possible simply not to allow the user from entering fewer or more digits than required.
6 It may have been possible to validate each digit for uniqueness as it was entered but that could have given rise to a lot of interruptions during input, especially during 2-D play, potentially leading to a poor user-experience.