CTEC1703 Computer Programming Coursework

Further explanation sheet

The coursework is about writing a simple game. This explanation sheet will help you to better understand the required functionality. Some features are more advanced, and it is expected you will attempt to complete as much of the game as possible. The mark you receive will be based upon how many unit tests you pass - the tests are designed to assess that the features work correctly.

You have been given a class `Game`, which consists of a `board` of size 10x10. The board is numbered as follows:

		x									
		0	1	2	3	4	5	6	7	8	9
Y	0	p	.	w	.	.	.	.	.	.	.
	1	.	.	w	.	.	.	.	.	.	.
	2	.	.	w	.	.	.	.	.	.	.
	3	.	c	.	.	.	.	.	.	.	.
	4	.	.	.	.	.	.	.	.	.	.
	5	.	.	.	.	.	.	.	.	.	.
	6	.	.	.	.	.	.	.	.	.	.
	7	.	.	.	.	.	.	.	.	.	.
	8	.	.	.	.	.	.	.	.	.	.
	9	.	.	.	.	.	.	.	.	.	.

The grid above should help you to visualise this scenario, and we have used p to show the current player position and w to show a wall. Coordinates are given as pairs in the form (x,y). Although different conventions can be used, here we define the origin as being in the upper left corner (which is common in computer graphics). Moving in the x direction would go right (or left) and moving in the y direction would go down (or up).

Therefore, in the snapshot shown above, the player is positioned in (0,0) at the origin. In the Game class you have been provided, the current position of the player is stored in `positionX` and `positionY`.

The use of “left”, “right”, “up”, and “down” allows the player to be moved around the board. You cannot move diagonally.

You will note the variable called `board` which is of type `Array[Array[Int]]` - this represents the grid shown above, where each cell is either empty (-1), a wall (0), or a coin (positive number). In the above example, there are walls in (2,0), (2,1), and (2,2). The player can move within the board. If the player tries moving into a wall or off the board, nothing happens. So, in the example above, the player could move right or down, but not up or left. If the player has moved right once, it cannot move right again (because of the wall). In these examples we use the symbols . for empty cell, w for wall, and c for coin because they are easier to read. In the board, they are represented by the aforementioned numbers.

There are also coins to collect, represented by positive numbers. If the player moves onto a coin, the `points` is increased by the amount the coin is worth and that coin is removed (the cell becomes -1). This is done by the `checkCoin` method. For example, if the points are 0 and the player moves onto position (1,3) – this contains a coin in the board above, suppose of value 100. This means the points would increase to 100. Any further moves onto (1,3) would not change the points, as the coin would have been removed.

You can generally compare cells of the grid to determine if there is a wall or a coin. So something like `if(board(5)(6)==0)` would check if there is a wall at this position. `if(board(5)(6)>0)` would check for a coin (any value greater than 0 is a coin). Of course, you may wish to look at several cells using loops etc.

The player can also save the current position to `saveX` and `saveY` by calling `save()`. If the player continues moving, the covered positions are evaluated by the `checkCoins` method. Covered positions are those inside a rectangle where the current position and the saved positions form two opposite corners of the rectangle. As soon as 9 or more positions are covered, the saved position is reset to -1, -1, indicating no saved position (these are the initial values of `saveX` and `saveY` as well) and all coins in the covered positions are collected. For example, the player is in position (0,0), save is applied, and the player then moves right (parts of the board have been left out to save space, the same is true in other examples further down):

		x									
		0	1	2	3	4	5	6	7	8	9
Y	0	s	p	w	.	.	.	.	.	.	.
	1	.	.	w	.	.	.	.	.	.	.
	2	.	c	w	.	.	.	.	.	.	.
	3	c	.	.	.	.	.	.	.	.	.

Here, the player (p) is in (1,0), the saved position (s) is in (0,0). There are two positions covered. The player moves down one position. There are now 4 positions covered. With a further move down, 6 positions are covered and the coin at (1,2) will be collected. A further move down would result in 8 positions being covered. Nothing happens here. However, with a further move down, the situation is this (note the player is now at (1,4)):

		x									
		0	1	2	3	4	5	6	7	8	9
Y	0	s	.	w	.	.	.	.	.	.	.
	1	.	.	w	.	.	.	.	.	.	.
	2	.	.	w	.	.	.	.	.	.	.
	3	c	.	.	.	.	.	.	.	.	.
	4	.	p	.	.	.	.	.	.	.	.

	5	.	.	.	.	.	.	.	.	.	.

There are now 10 positions covered. Since 9 or more are covered, we now collect all remaining coins in here, which would include (0,1), (0,2), (0,3), and (0,4), even though these positions were not touched by the player – note (0,3) contains a coin, which will therefore be collected. It would also reset the saved position to -1,-1, which indicates no saved position. Walls count as covered positions as well, if they are in the rectangle, however, you cannot move through them of course.

The method `move` takes a string where up, left, down, and right are encoded as w, a, s, and d respectively. The moves are executed according to the string, so “aaddw” would move to the left twice, to the right twice and up once. If there is a wall, an individual move (but not the complete string) is not executed. After each individual move a coin is collected if moved onto and the saved position is evaluated to see if 9 or more positions have been covered. This is the same as happens after moving left, right, up, or down.

The method `suggestMove` returns a string in the same format as noted for the `move` method, which moves the player from the current position to position (x,y). No specific requirements for the efficiency of the solution exist. The move cannot jump walls. The method is restricted to finding a path which follows two sides of a rectangle defined by the current position and the target. This means there is a given number of moves in one direction first, followed by a given number of moves in another direction. If the first move was horizontal, the second is vertical, and vice versa. If this is not possible due to walls, the method returns an empty string. If two paths are possible, any of them can be returned. No actual move is done. For example, below the two potential paths to get from (0,0) to (3,3) are indicated:

		x									
		0	1	2	3	4	5	6	7	8	9
Y	0	v> p	>	>w	v	.	.	.	.	.	.
	1	v	.	w	v	.	.	.	.	.	.
	2	v	.	w	v	.	.	.	.	.	.
	3	>	>	>	.	.	.	.	.	.	.
	4	.	.	.	.	.	.	.	.	.	.

Since the move “dddsss” is blocked by a wall, it is not possible, so the method would return “sssddd”. If this pathway would be blocked as well, an empty string would be returned.

The file Game.scala also contains an object `GameBuilder` which allows the initialisation of predefined game scenarios, such as those shown throughout this explanation. This object could go in its own file, but has been placed within the same file as the Game class as we want you to be able to submit a single file. There are three ways in which the game can be initialised, and the first has been completed for you. Use the comments above the other initialisation methods to help complete them (*we suggest you do this at an early stage as the unit tests refer to these*). Lists of Tuples are used to provide coordinates. Remember that a tuple like (3,3) represents a single value of type tuple - in this case a 2-tuple (pair), whereas (4,5,7) is a 3-tuple (triplet). The elements of a tuple can be accessed individually using _1, _2, etc. (This may be helpful: https://docs.scala-lang.org/tour/tuples.html)