**Daily Coding Problem**

Good morning! Here's your coding interview problem for today.

This question was asked by Zillow.

You are given a 2-d `matrix` where each cell represents number of coins in that cell. Assuming we start at `matrix[0][0]`, and can only move right or down, find the maximum number of coins you can collect by the bottom right corner.

For example, in this matrix

```
0 3 1 1
2 0 0 4
1 5 3 1
```

The most we can collect is 0 + 2 + 1 + 5 + 3 + 1 = 12 coins.

This is definitely not any more difficult than challenge by Slack.
It is a case of reducing code in some areas, but also extending it in other areas.
The alternation in both directions is still valid DOWN => RIGHT => DOWN and also RIGHT => DOWN = RIGHT, since instead of obstacles the grid, it has coins of different values.

There are other changes, unfortunately it was realised that I had excess code in the Slack challenge (see below).

I will reflect changes below in both the following methods:
public void movesRight
public void movesDown

```
if (!outBounds)
 {

for (int i=1; i<=RIGHTvalue; i++)
 {
    try
    {

    //hit a wall
    if ((matrix[currentPosY][currentPosX+i]==1))
    {
        System.out.println("hit a wall at" + "["+currentPosY +"]" + "[" +(currentPosX+RIGHTvalue) +  "]" + " - discard sequence" );
        successfulFinish=false;
        decision(movementPattern);
        i=RIGHTvalue;

        //this can be enable for better efficiency, however all conditions
        //are set in the code to not process any conditions for remaining moves
        //due to obstructions
        //t=numbers.length;
        //break;

    }
 }
 catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("OUT OF BOUNDS: =>  " + "["+(currentPosY)+"]"+"["+(currentPosX+t)+"]" );
        successfulFinish=false;
        outBounds=true;

    }

 }
```

It was also later realised that the **try and catch** was not required since it would only perform this execution if
currentPosX + RIGHTvalue < matrix[0].length  (!outofBounds).
It is very likely I introduced **try and catch** before I introduced the  if(!outBounds), hence it failed in testing.  At end I had double resilience and it is not required.
I will remove this surplus in coin challenge by Zillow.

All the code highlighted in orange would be removed. Instead of searching for a wall within the RIGHTvalue, it is interested to ensure that there is actually a path and collect the coins.

Following code would be added here:
//mantains running total
totalCoins = totalCoins + matrix[currentPosY]
[currentPosX+1];

At this point, it has discovered that whilst moving across, it has

This is now at the point where it has to store the results.
I will try to follow the same technique as Snakes and Ladders in which once it completed the entire board (in this case the matrix),  it would present the following information. Again it is useful to have it ongoing, in event of memory issues..

*******ANALYSIS OF ALL THE MOVES UNDERTAKEN  [Start Position => End Position]******

Minimum Coins so far: 8

Maximum Coins so far: 135

In this section of the code, it would adhere to similar recording of results as Snakes and Ladders. In Snakes and Ladders we were concerned with minimum moves. If it was a new minimum, it would overwrite the existing valiue.  If it was equal to minimum, it would keep existing records and add a new entry at the end...
For this challenge of total coins, it has to store both the valueSet and also totalCoins

```
if (currentPosY==(matrix.length-1) && currentPosX==(matrix[0].length-1))
{
    successfulFinish=true;
    System.out.println("*****Successful: " + valuesSet);


    completedPaths[count] = (valuesSet +  "    Subset: " + count + "  " + movementPattern);

    count++;

}
```

In terms of displaying the results mid execution,  since it still has to be based on following methods due to variation in coin values,  I am left with an identical flaw as my previous Slack code.

I will remain adamant to fix this, notably  since in an open grid (with no constraints), there will be more cascaded moves...
This is a reminder of the issue that had occurred.

```
//this now processes each String in the ValuesSet (which will be Strings not outputted y
for (int entry=0; entry<valuesSet.length; entry++)
{
    if (valuesSet[entry]!="ALREADY PROCESSED")    //as per above, it needs bypass these
    {
```

```
//if it completes a successful move...
if (startToFinish(valuesSet[entry]))
{
    subsetEntry++;    //static variable, it will keep track of number successful com

    System.out.println(valuesSet[entry] + "    Subset: " + subsetEntry  + "  at cycl
}
```

This would be the point in the Staircase class in which it attempts to check for boolean =true

This would invoke the startToFinish method  and it would create a new instance of the Direction class

```
//this method instantiates Direction class.
//this does processing for movement in matrix based on values in subsets
//totalling k..
//if it successfully finishes at last position (bottom right) of matrix, true is returnec
public static boolean startToFinish(String valuesSet)
{
    Direction d = new Direction(valuesSet, matrix);

    return d.successfulFinish;
}
```

In the constructor Direction it would call both scenarios...

```
public Direction(String valuesSet, int[][] matrix)
{
    this.valuesSet=valuesSet;
    this.matrix=matrix;

    movesAlternateDownRight();
    movesAlternateRightDown();
}
```

Both set the boolean for successfulFinish and  it will be overwritten. So, the value in method will always be from the last method...

SOLUTION

One possible solution has arisen. And unfortunately, since it was not a technique not used in my coding before, it has taken a while to realise this. Constructor overloading / changing parameter order in method declaration. This has created a different method signature. And when the constructor has been instantiated, it chooses the correct constructor.
I can then separate out the two methods into separate constructors.

Here is how I managed to perform the operation, it required lots of small changes. **But fortunately it functioned**, which means I can perform this challenge knowing that end user will get some results in event that available memory elapses...

Separate structure for Down => Right => Down.....

```
//this now processes each String in the ValuesSet (which will be Strings not outputted y
for (int entry=0; entry<valuesSet.length; entry++)
{
    if (valuesSet[entry]!="ALREADY PROCESSED")    //as per above, it needs bypass these
    {
```

1 There is now a unique method for alternation Down => Right => Down => Right

```
//if it completes a successful move...
if (startToFinishDownRight(valuesSet[entry]))
{
    subsetEntry++;    //static variable, it will keep track of number successful combination
    System.out.println(valuesSet[entry] + "    Subset: " + subsetEntry  + "  at cycle number
}
```

This would be the point in the Staircase class in which it attempts to check for boolean =true

This would invoke the startToFinishDownRight method and it would create a new instance of the Direction

```
public static boolean startToFinishDownRight(String valuesSet)
{

    //this is the associated constructor
    //public Direction(String valuesSet, int[][] matrix, String alternateDownRight)

    Direction d = new Direction(valuesSet, matrix, "DownRight");

    return d.successfulFinish;
}
```

Also note the order of the arguments differ in order to call correspondng constructor containing moveAlternateDownRight()

```
//observe constructors with same method signature but re-arranged.
public Direction(String valuesSet, int[][] matrix, String alternateDownRight)
{
    this.valuesSet=valuesSet;
    this.matrix=matrix;

    movesAlternateDownRight();

}
```

In the constructor Direction it would call ONLY one method movesAlternateDownRight().
So there is no longer case of value being overwritten boolean successfulFinish

Separate structure for Right => Down => Right.....

```
//this now processes each String in the ValuesSet (which will be Strings not outputted y
for (int entry=0; entry<valuesSet.length; entry++)
{
    if (valuesSet[entry]!="ALREADY PROCESSED")    //as per above, it needs bypass these
    {
```

1 There is now a unique method for alternation Right => Down => Right => Down

```
//if it completes a successful move...
if (startToFinishRightDown(valuesSet[entry]))
{
    subsetEntry++;    //static variable, it will keep track of number successful combination.

    System.out.println(valuesSet[entry] + "    Subset: " + subsetEntry  + "  at cycle number
}
```

This would be the point in the Staircase class in which it attempts to check for boolean =true

This would invoke the startToFinishRightDown method and it would create a new instance of the Direction

```
public static boolean startToFinishRightDown(String valuesSet)
{

    //this is the associated constructor
    //public Direction(String valuesSet, String alternateRightDown, int[][] matrix)

    Direction d = new Direction(valuesSet, "RightDown", matrix);

    return d.successfulFinish;
}
```

Also note the order of the arguments differ in order to call correspondng constructor containing moveAlternateRightDown()

```
//observe constructors with same method signature but re-arranged.
public Direction(String valuesSet, String alternateRightDown, int[][] matrix)
{
    this.valuesSet=valuesSet;
    this.matrix=matrix;

    movesAlternateRightDown();
}
```

In the constructor Direction it would call ONLY one method movesAlternateRightDown().
So there is no longer case of value being overwritten

This will be it, this is all my planning for the coin collecting challenge.
I am officially ready to collection stray coins.

And there is nothing more rewarding than collecting coins in my final matrix movement challenge.

Perhaps if my skills improve further, I can try to perform multi-directional movements...