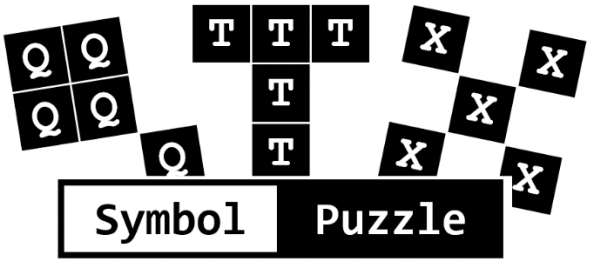

Skeleton Code Breakdown

Static Methods

Identifier / Data		Description
Main		
Parameters	**default**	This is the main entrance point for the application. It is used to determine whether the application is going to use a standard, randomly created puzzle or load an external puzzle text file. It initialises a string variable called **Again** assigning the value 'y' and an integer variable of **Score**. The method then enters into the main application loop. It is held in that loop by the value of the **Again** variable. The loop operates using the following steps: • Prompt the user if they would like to create a standard puzzle or load an external file. • Instantiate a new **Puzzle** object called **MyPuzzle**. • If the user has entered a filename, call the default constructor in the **Puzzle** class passing in the filename as a parameter. This will create a new puzzle from an external text file. • If the user does not enter a filename (which is calculated by the length of the filename being zero), call the overloaded constructor in the **Puzzle** class passing in the values 8 and 38. The first parameter is the **GridSize** for a standard puzzle, creating a grid which is 8 columns wide by 8 rows. The second parameter is calculated by squaring the **GridSize**, multiplying the result by 0.6 and then rounding down to an integer value. • The method then calls the **AttemptPuzzle()** method on **MyPuzzle** which starts the puzzle. The resultant **Score** when a puzzle is complete is returned and assigned to the **Score** variable, which is displayed to the user. • The method then prompts the user if they would like to do another puzzle, setting the **Again** variable appropriately to either stop the loop and, therefore, end the program, or allow it to repeat again.
Return values	n/a	

Class: Puzzle

<table>
<tr><th colspan="2">Identifier / Data</th><th>Description</th></tr>
<tr><td colspan="3"><<constructor>></td></tr>
<tr><td>Parameters</td><td>Filename : String

OR

Size : Int
StartSymbols : Int</td><td rowspan="3">Uses overloading to accept two different versions of the constructor.

This version of the constructor is called when an external puzzle file is loaded.

If one string argument is passed (in the case of the user loading an external puzzle file), the method:

Initialises the following private attributes:
• Score to 0
• SymbolsLeft from parameter StartSymbols
• GridSize from parameter Size
• Grid as List
• AllowedPatterns as List
• AllowedSymbols as List

These attributes are subsequently populated by the LoadPuzzle() method.

The method calls the LoadPuzzle() method passing the Filename parameter.

This version of the constructor is called when a new standard puzzle is generated.

If two integer arguments are passed (in the case of the user creating a standard puzzle), the method:

Initialises the following private attributes:
• Score to 0
• SymbolsLeft from parameter StartSymbols
• GridSize from parameter Size
• Grid as List

The method performs a count-controlled loop upperbound to the square of the GridSize. Inside the loop the method creates and adds all the cells for a standard puzzle into the Grid. Using a random number between 1 and 100, each cell has a 90% chance of being a normal cell and a 10% chance of being a blocked cell.

The method then initialises AllowedPatterns as a list of patterns and AllowedSymbols as a list of strings. It then generates the default pattern objects for the Q, X and T patterns, adding them to the AllowedPatterns list together with adding the associated symbol ‘Q’, ‘X’ or ‘T’ to the AllowedSymbols list.</td></tr>
<tr><td>Return values</td><td>n/a</td></tr>
<tr><td colspan="2"></td></tr>
</table>

AttemptPuzzle (public)		
Parameters	n/a	This method is the main loop for attempting puzzles one at a time. It is held in the loop using the local Boolean variable **Finished**. The method firstly displays the current puzzle state by calling the **DisplayPuzzle()** method. It then displays the current user **Score**. The method then asks the user to separately enter the **Row** and **Column** of where they would like to place a symbol into the puzzle. The method uses try…catch structures to check if the user has entered integer values, but does not check if they are valid within the bounds of the grid. The catch does not give any error messages to the user for erroneous input. The method then calls the **GetSymbolFromUser()** method to get a symbol to place into the grid from the user, and decrements the number of symbols available left to be used in the puzzle. The method then creates a local copy of the cell at the location given by the user and tests to see if the symbol given by the user can be used in that cell by calling the **CheckSymbolAllowed()** method passing the symbol entered by the user as a parameter. If the symbol can be used in that cell, the symbol in the cell is changed using the **ChangeSymbolInCell()** method. The method then checks if this update generates a pattern match by calling the **CheckforMatchWithPattern()** method, passing **Row** and **Column** entered by the user earlier as parameters. The result of this check is assigned to an integer variable **AmountToAddToScore**. If this is greater than 0, it is added to the user **Score**. The method then checks if all the symbols available have been used, and if so, exits the main program loop. If the main program has exited, the final puzzle state is displayed by calling the **DisplayPuzzle()** method and the user **Score** is returned.
Return values	**Score** : Int	

<table>
<tr><td colspan="3">CheckforMatchWithPattern (public)</td></tr>
<tr><td>Parameters</td><td>Row : Int
Column : Int</td><td rowspan="3">Uses a nested loop to iterate through the grid concatenating together a string variable called PatternString. The string is built from nine cells in a 3 × 3 section of the grid in the order of a helix. A nested loop is used to concatenate all nine possible combinations of pattern that could include the cell at the different locations in a 3 × 3 section of the grid. This is done with a try…catch structure to prevent the code from crashing if a cell location outside of the bounds of the grid is accessed.

Once the PatternString has been generated, the code then uses a foreach loop to check it for a match in the AllowedPatterns list by calling the MatchesPattern() method, passing the PatternString and symbol being checked as parameters.

If a match is found, the method calls the AddToNotAllowedSymbols() on each cell included in the matching 3 × 3 section of the grid, passing in the symbol being checked as a parameter. This prevents the same symbol from being placed into that cell in a future turn.

If a match has been found, the method returns the Score of 10, otherwise it returns the Score of 0.</td></tr>
<tr><td>Return values</td><td>Int</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="3">CreateHorizontalLine (private)</td></tr>
<tr><td>Parameters</td><td>n/a</td><td rowspan="3">Uses iteration to concatenate a horizontal line of ‘-’ characters which is the correct width for the grid being used in the current puzzle.</td></tr>
<tr><td>Return values</td><td>Symbol : String</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="3">DisplayPuzzle (public)</td></tr>
<tr><td>Parameters</td><td>n/a</td><td rowspan="3">Used to print out the grid onto the screen. The method works by using the following steps: Print a blank line. If the GridSize is less than 10, print a wider space onto the screen, then iterate through to the GridSize printing a space followed by the count headings for each column. Print a blank line. Print a horizontal line by calling the CreateHorizontalLine() method. The method then iterates through the Grid list attribute printing out the row number followed by space then a ‘|’ symbol and the symbol in each cell of the grid. The iteration uses the MOD function to calculate the length of a row before printing a final ‘|’ symbol followed by a horizontal line underneath. This process is repeated until the whole grid has been printed to the screen.</td></tr>
<tr><td>Return values</td><td>n/a</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="3">GetCell (private)</td></tr>
<tr><td>Parameters</td><td>Row : Int
Column : Int</td><td rowspan="3">Uses the Row and Column parameters to calculate the correct Cell element in the one-dimensional Grid list which is then returned. If the Row and Column parameters generate an index location which is less than 0, the method raises an IndexError. The other pre-release code versions do not do this.</td></tr>
<tr><td>Return values</td><td>Cell</td></tr>
<tr><td colspan="2"></td></tr>
</table>

<table>
<tr><td colspan="3">GetSymbolFromUser (private)</td></tr>
<tr><td>Parameters</td><td>n/a</td><td rowspan="3">Used for getting a symbol from the user to place into the grid. The method uses a loop to repeatedly ask the user to enter the symbol they want to use until they enter a valid symbol which is part of the AllowedSymbols list.

When they enter a valid symbol, it is returned.</td></tr>
<tr><td>Return values</td><td>Symbol : String</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="3">LoadPuzzle (private)</td></tr>
<tr><td>Parameters</td><td>Filename : String</td><td rowspan="3">The method loads an external text file using the Filename parameter.

See ‘Puzzle File Breakdown’ for details on what each line does in an external puzzle file.

The method sequences through the lines in the external file from the Filename parameter. It performs the following tasks:

Assigns NoOfSymbols from the first line of the file. It then uses this value to iterate through the next lines, assigning the symbols from those lines into the AllowedSymbols list.

Assigns NoOfPatterns from the next single readline. It then uses this value to iterate through the following lines, reading in each pattern. A pattern line contains a comma-separated list – the first element is the symbol for a pattern, and the second element is the pattern itself. These are used to instantiate a new pattern object which is then added to the AllowedPatterns list.

Assigns GridSize from the next single readline. It then uses the square of this value to iterate through the following lines, reading in each cell. A cell line contains a comma-separated list – the first element is the symbol for the cell, and the remaining elements is a sublist which contains all of the symbols for the SymbolsNotAllowed list for that cell. If the first element is an ‘@’ symbol the application instantiates and appends a BlockedCell into the Grid. Otherwise the application instantiates a Cell by using the cell symbol and subsequent symbols not allowed elements.

Once all the cells have been appended to the grid, the method assigns the next line in the file to the Score attribute and the final line to the SymbolsLeft attribute.

The method uses a try…catch structure to handle file errors. If an error occurs, an error message is given to the user, but the method does not give the user the opportunity to try to reload the file.</td></tr>
<tr><td>Return values</td><td>n/a</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Class: Pattern

<table>
<tr><th colspan="2">Identifier / Data</th><th>Description</th></tr>
<tr><td colspan="3"><<constructor>></td></tr>
<tr><td>Parameters</td><td>SymbolToUse : String
PatternString : String</td><td rowspan="2">Initialises the following private attributes:
• Symbol from parameter SymbolToUse.
• PatternSequence from parameter PatternString.</td></tr>
<tr><td>Return values</td><td>n/a</td></tr>
<tr><td colspan="3">GetPatternSequence (public)</td></tr>
<tr><td>Parameters</td><td>n/a</td><td rowspan="2">Returns the value of the private attribute PatternSequence.</td></tr>
<tr><td>Return values</td><td>PatternSequence : String</td></tr>
<tr><td colspan="3">MatchesPattern (public)</td></tr>
<tr><td>Parameters</td><td>PatternString : String
SymbolPlaced : String</td><td rowspan="3">This is used to confirm that a pattern string found by the helix concatenation through a 3 × 3 section of the grid matches the PatternSequence attribute in a pattern object.

If the passed parameter SymbolPlaced does not match the Symbol for the pattern, the method returns false.

If it does match, the method iterates through the private attribute PatternSequence comparing each letter with the letter at the same position in the parameter PatternString. A PatternSequence contains symbol characters (‘Q’, ‘X’ or ‘T’) for the pattern, and the ‘*’ character for letters which don’t need to match in the pattern. The iteration only compares characters between the PatternSequence variable and PatternString parameter which match the symbol. If there are any differences, the pattern match returns false. The comparison in this iteration uses a try…catch to handle any erroneous comparisons of out-of-range errors. If an exception is caught, the method displays to the user that an exception has occurred.

If the iteration completes, all the symbol characters have matched and, therefore, the method returns true.</td></tr>
<tr><td>Return values</td><td>Boolean</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Class: Cell

Identifier / Data		Description
<<constructor>>		
Parameters	n/a	Initialises the protected attribute **Symbol** to an empty string. Initialises the private attribute **SymbolsNotAllowed** to a new empty string list.
Return values	n/a	
AddToNotAllowedSymbols (public)		
Parameters	**SymbolToAdd** : String	Appends the parameter **SymbolToAdd** to the private list attribute **SymbolsNotAllowed**.
Return values	n/a	
ChangeSymbolInCell (public)		
Parameters	**NewSymbol** : String	Assigns the **NewSymbol** parameter to the protected attribute **Symbol**.
Return values	n/a	
CheckSymbolAllowed (public) <<virtual>>		
Parameters	**SymbolToCheck** : String	Iterates through the private list attribute **SymbolsNotAllowed**. If the parameter **SymbolToCheck** is found in the list, the method returns false, otherwise it returns true.
Return values	**Boolean**	
GetSymbol (public)		
Parameters	n/a	The method uses the **IsEmpty()** method to test if the cell is empty. If it is empty, the method returns a '-', otherwise it returns the value of the protected attribute **Symbol**.
Return values	**Symbol** : String	
IsEmpty (public)		
Parameters	n/a	If the private attribute **SymbolsNotAllowed** is empty, the method returns true, otherwise it returns false.
Return values	**Boolean**	
UpdateCell (public)		
Parameters	n/a	This method is not used in the pre-release material. This method has been included into the pre-release code to give the option for a question which creates a new class that inherits the cell class and overrides its base methods.
Return values	n/a	

Class: BlockedCell (inherits from Cell)

Identifier / Data		Description
<<constructor>>		
Parameters	**n/a**	Initialises the parent attribute **Symbol** to '@'.
Return values	n/a	
CheckSymbolAllowed (public) <<override>>		
Parameters	**SymbolToCheck** : String	Overrides the **CheckSymbolAllowed()** method from the base class. While technically this could be used as a helper method to allow a blocked class object to access the private **SymbolsNotAllowed** list attribute in the base class, it is used to simply return false regardless of the value of the parameter **SymbolToCheck**.
Return values	**Boolean**	

