AQA

AS COMPUTER SCIENCE (7516/1A/1B/1C/1D/1E)

Paper 1

Mark scheme

Mark schemes are prepared by the Lead Assessment Writer and considered, together with the relevant questions, by a panel of subject teachers. This mark scheme includes any amendments made at the standardisation events which all associates participate in and is the scheme which was used by them in this examination. The standardisation process ensures that the mark scheme covers the students' responses to questions and that every associate understands and applies it in the same correct way. As preparation for standardisation each associate analyses a number of students' scripts: alternative answers not already covered by the mark scheme are discussed and legislated for. If, after the standardisation process, associates encounter unusual answers which have not been raised they are required to refer these to the Lead Assessment Writer.

It must be stressed that a mark scheme is a working document, in many cases further developed and expanded on the basis of students' reactions to a particular paper. Assumptions about future mark schemes on the basis of one year's document should be avoided; whilst the guiding principles of assessment remain constant, details will change, depending on the content of a particular examination paper.

Further copies of this Mark Scheme are available from http://www.aqa.org.uk/

COMPONENT NUMBER: **Paper 1**

COMPONENT NAME:

STATUS:

DATE: **7 Aug 2014**

The following annotation is used in the mark scheme.

; - means a single mark
// - means alternative response
/ - means an alternative word or sub-phrase
A - means acceptable creditworthy answer
R - means reject answer as not creditworthy
NE - means not enough
I - means ignore
DPT - in some questions a specific error made by a candidate, if repeated, could result in the loss of more than one mark. The **DPT** label indicates that this mistake should only result in a candidate losing one mark, on the first occasion that the error is made. Provided that the answer remains understandable, subsequent marks should be awarded as if the error was not being repeated.

01	**1**	**All marks AO3 (programming)** **Python 2.6:** `print "How far to count?"` `HowFar = input()` `while HowFar < 1:` `  print "Not a valid number, please try again."` `  HowFar = input()` `for MyLoop in range(1,HowFar+1):` `  if MyLoop%3 == 0 and MyLoop%5 == 0:` `  print "FizzBuzz"` `  elif MyLoop%3 == 0:` `  print "Fizz"` `  elif MyLoop%5 == 0:` `  print "Buzz"` `  else:` `  print MyLoop` **1 mark:** Correct prompt "`How far to count?`" followed by `HowFar` assigned value entered by user; **1 mark:** `WHILE` loop has syntax allowed by the programming language and correct condition for the termination of the loop; **1 mark:** Correct prompt "`Not a valid number, please try again.`" followed by `HowFar` being assigned value entered by the user (must be inside iteration structure); **1 mark:** Correct syntax for the `FOR` loop using correct range appropriate to language; **1 mark:** Correct syntax for an `IF` statement, including a `THEN` and `ELSE/ELIF` part; **1 mark:** Correct syntax for `MyLoop MOD 5 = 0` and `MyLoop MOD 3 = 0` used in the `IF` statement(s); **1 mark:** Correct output for cases in the selection structure where `MyLoop MOD 3 = 0` or `MyLoop MOD 5 = 0` or both - outputs "`FizzBuzz`", "`Fizz`" or "`Buzz`" correctly; **1 mark:** Correct output (in the `ELSE` part of selection structure), when `MyLoop MOD 3 <> 0` and `MyLoop MOD 5 <> 0` - outputs value of `MyLoop`;	8

01	**2**	**All marks AO3 (evaluate)** **Info for examiners:** must match code from 01.1, including prompts on screen capture matching those in code. Code for 01.1 must be sensible.	1

		First Test `How far to count?` `18` `1` `2` `Fizz` `4` `Buzz` `Fizz` `7` `8` `Fizz` `Buzz` `11` `Fizz` `13` `14` `FizzBuzz` `16` `17` `Fizz` **Second Test** `How far to count?` `-1` `Not a valid number, please try again.` Screenshot with user input of `18` and correct output and user input of -1 and correct output; **A.** different formatting of output eg line breaks	

01	**3**	**Mark is for AO2 (analysis)** A FOR loop is used as it is to be repeated a known number of times;	1

01	**4**	**All marks AO2 (analysis)** Example of input: [nothing input] [a string] for example: `12A` Method to prevent: can protect against by using a try,except structure // exception handling; test the input to see if digits only;	3

		convert string to integer and capture any exception; use a repeat/while structure // alter repeat/while to ask again until valid data input; **1 mark:** Example of input **Max 2 marks:** Description of how this can be protected against	

01	**5**	**All marks AO1 (understanding)** Use of indentation to separate out statement blocks; Use of comments to annotate the program code; Use of procedures / functions / sub-routines; Use of constants; **Max 3, any from 4 above**	3

01	**6**	**All marks AO2 (apply)** | **Input string**	**Accepted by FSM?**
---	---
`aaab`	YES
`abbab`	NO
`bbbbba`	YES	 **1 mark:** Two rows of table completed correctly; **OR** **2 marks:** All three rows of table completed correctly; **A.** Alternative indicators for YES and NO	2

01	**7**	**All marks AO2 (apply)** **1 mark:** a string containing zero or more (**A.** 'any number of') b characters; **1 mark:** and an odd amount of a characters; **N.E.** all strings containing an odd number of characters	2

02	**1**	**Mark is for AO1 (understanding)** `Cavern // TrapPositions;`	1

02	**2**	**Mark is for AO1 (understanding)** `SetPositionOfItem // MakeMonsterMove;`	1

02	**3**	**Mark is for AO1 (understanding)** `Count1 // Count2 // Choice;` `(Python Only) NO_OF_TRAPS // N_S_DISTANCE // W_E_DISTANCE;`	1

02	**4**	**Mark is for AO1 (understanding)** `GetMainMenuChoice // GetNewRandomPosition // SetPositionOfItem // SetUpGame // SetUpTrainingGame // GetMove // CheckValidMove // CheckIfSameCell // MoveFlask`	1

02	**5**	**All marks AO3 (analysis)** a nested loop is used as we need to repeat something inside a section that is also repeating; so that for each row we can loop through each column; to work our way through the 2 dimensions of the cavern; `Count1` controls the rows of the display; `Count2` controls the columns of the display; **Max 3:** Any 3 from above	3

02	**6**	**All marks AO1 (understanding)** **1 mark:** Only need to change its value once//at the top of the program // from one place only (for the new value to apply wherever it is used); **1 mark:** Makes program code easier to understand;	2

02	**7**	**All marks AO2 (analyse)** **1 mark:** (Command inside loop) randomly chooses coordinates to place item at; **1 mark:** The condition checks that no other item has already been placed at the selected coordinates // the location is empty; **1 mark:** The while loop is required to repeat the coordinate selection until an empty location is found // to keep choosing coordinates if the location found is not empty;	3

02	**8**	**All marks AO2 (analyse)** **1 mark:** If the monster moves into the flask cell and the flask is not moved elsewhere it will not be there when the monster moves away from this cell; **1 mark:** You can't have two items in any cell; **1 mark:** By swapping the monster and the flask cells we can ensure that the flask stays in the cavern;	3

02	**9**	**All marks AO2 (analyse)** **1 mark:** Even though the monster usually makes 2 moves the player might be eaten on the first of the two moves; **1 mark:** A while loop allows us to complete 2 moves when necessary but exit on the first move if the player is eaten;	2

02	**10**	**All marks AO2 (understanding)** Easier reuse of routines in other programs; Routine can be included in a library; Helps to make the program code more understandable; Ensures that the routine is self-contained // routine is independent of the rest of the program; (global variables use memory while a program is running) but local variables use memory for only part of the time a program is running; reduces possibility of undesirable side effects; using global variables makes a program harder to debug; **Max 2:** Any 2 from above	2

03	**1**	**1 mark for AO3 (design) and 3 marks for AO3 (programming)** **AO3 (design) – 1 mark:** Note that AO3 (design) mark is for selecting appropriate techniques to use to solve the problem, so should be credited whether the syntax of programming language statements is correct or not and regardless of whether the solution works. **1 mark:** Identification of correct logical conditions required to determine if the player attempts to move North from the northern end of the cabin; **AO3 (programming) – 3 marks:** Note that AO3 (programming) marks are for programming and so should only be awarded for syntactically correct code that performs its required function. **1 mark:** Selection statement with two correct conditions; **1 mark:** Value of `False` returned correctly by the function if illegal north move is made; **R.** if a value of `False` will always be returned by the function **R.** if all north moves will return `False` **R.** if all moves when `PlayerPosition.NoOfCellsSouth` is in row 1 will return `False` **1 mark:** Value of `True` returned correctly by the function if legal north move is made; **A.** Answers which combine all the checks for a valid move into one selection statement **Python 2.6:** `def CheckValidMove(PlayerPosition,Direction):` `  ValidMove = True` `  if not (Direction in ['N','S','W','E','M']):` `  ValidMove = False` **`  if PlayerPosition.NoOfCellsSouth == 0 and Direction == 'N':`** **`  ValidMove = False`** `  return ValidMove`	4

03	**2**	**Mark is for AO3 (programming)**	1

		Python 2.6: `...` `MoveDirection = ''` `DisplayCavern(Cavern, MonsterAwake)` `while not (Eaten or FlaskFound or (MoveDirection == 'M')):` `    ValidMove = False` `    while not ValidMove:` `        DisplayMoveOptions()` `        MoveDirection = GetMove()` `        ValidMove = CheckValidMove(PlayerPosition, MoveDirection)` `        `**`if not ValidMove:`** `            `**`print "That is not a valid move, please try again."`** `    if MoveDirection != 'M':` `...` **1 mark:** Selection structure with correct condition that displays the correct message under the correct circumstances;	

03	**3**	**Mark is for AO3 (evaluate)** **Info for examiner:** Must match code from 03.1 and 03.2, including prompts on screen capture matching those in code. Code for 03.1 and 03.2 must be sensible. `Please enter your choice:` `1` ` -------------` `|*|					|` ` -------------` `|						|` ` -------------` `|						|` ` -------------` `|						|` ` -------------` `|						|` ` -------------`  `Enter N to move NORTH` `Enter E to move EAST`	1

		Enter S to move SOUTH Enter W to move WEST Enter M to return to the Main Menu N That is not a valid move, please try again. Screen capture(s) showing correct cavern state with a player at the northern end of the cavern (top line). 'N' being entered at prompt, followed by correct error message being displayed ;	

<table>
<tr>
<td>04</td>
<td>1</td>
<td>
1 mark for AO3 (design) and 7 marks for AO3 (programming)

<u>Mark Scheme</u>
<table>
<tr><th>Level</th><th>Description</th><th>Mark Range</th></tr>
<tr><td>4</td><td>A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution. The score is updated correctly as a result of all four described triggers. At the end of the game the required message is displayed in at least one of the two circumstances. To award eight marks, the code must perform exactly as required in the question. It is evident from the program code that the code has been designed appropriately to ensure that the task is achieved.</td><td>7-8</td></tr>
<tr><td>3</td><td>There is evidence that a line of reasoning has been followed to produce a logically structured subroutine that works correctly in most cases but with some omissions (e.g. the score may not be updated correctly in one of the four cases or the message that is displayed may not match the question). It is evident from the program code that it has been designed appropriately to update the score correctly in most circumstances.</td><td>5-6</td></tr>
<tr><td>2</td><td>There is evidence that a line of reasoning has been partially followed as the score is updated correctly as a result of at least two of the listed triggers. The correct message is not displayed. There is not enough evidence that a line of reasoning has been followed to award a mark for the design of the solution.</td><td>3-4</td></tr>
<tr><td>1</td><td>A variable has been used to store the score and there is an attempt to modify this as a result of at least one of the four listed triggers. This modification may not be in exactly the right place and the value to change the score by may be incorrect, but it should be possible to see that it was intended to be linked to a particular trigger. To award two marks instead of one, some of the code must be syntactically correct. There is insufficient evidence to suggest that a line of reasoning has been followed or that the solution has been designed.</td><td>1-2</td></tr>
</table>
</td>
<td>8</td>
</tr>
</table>

Guidance

Evidence of AO3 (design) - 1 point:

Evidence of design to look for in responses:

- Identifying the correct locations in the program code to change the score at. To be credited for this point, the correct location for at least three of the four changes must be identified, but the amount that `Score` is changed by could be incorrect, as could the syntax.

Note that AO3 (design) point is for selecting appropriate techniques to use to solve the problem, so should be credited whether the syntax of programming language statements is correct or not and regardless of whether the solution works.

Evidence of AO3 (programming) – 7 points:

Evidence of programming to look for in responses:

- `Score` is assigned the value 0 – before the first repetition structure in `PlayGame`
- `Score` is incremented by 10 after a valid player move
- `Score` is incremented by 50 when the flask is found
- `Score` is decreased by 10 when a trap is activated
- `Score` is decreased by 50 when eaten by the monster
- Correct message displayed with `Score` if player wins
- Correct message displayed with `Score` if player loses

Note that AO3 (programming) points are for programming and so should only be awarded for syntactically correct code.

Example Solution - Python 2.6

```
def PlayGame(Cavern, TrapPositions,
MonsterPosition, PlayerPosition, FlaskPosition,
MonsterAwake):
  Score = 0
  Eaten = False
  FlaskFound = False
  MoveDirection = ''
  DisplayCavern(Cavern, MonsterAwake)
  while not (Eaten or FlaskFound or
```

```
(MoveDirection == 'M')):
    ValidMove = False
    while not ValidMove:
      DisplayMoveOptions()
      MoveDirection = GetMove()
      ValidMove =
CheckValidMove(PlayerPosition, MoveDirection)
      if not ValidMove:

          print "That is not a valid move,
please try again."

    if MoveDirection != 'M':
      Score = Score + 10
      MakeMove(Cavern, MoveDirection,
PlayerPosition)
      DisplayCavern(Cavern, MonsterAwake)
      FlaskFound =
CheckIfSameCell(PlayerPosition, FlaskPosition)
      if FlaskFound:
        DisplayWonGameMessage()
        Score = Score + 50
        print "Your score was: ",Score
      Eaten = CheckIfSameCell(MonsterPosition,
PlayerPosition)
      if not MonsterAwake.Is and not FlaskFound
and not Eaten:
        MonsterAwake.Is =
CheckIfSameCell(PlayerPosition,
TrapPositions[0])
        if not MonsterAwake.Is:
          MonsterAwake.Is =
CheckIfSameCell(PlayerPosition,
TrapPositions[1])
        if MonsterAwake.Is:
          DisplayTrapMessage()
          Score = Score - 10
          DisplayCavern(Cavern, MonsterAwake)
      if MonsterAwake.Is and not Eaten and not
FlaskFound:
        Count = 0
        while Count < 2 and not Eaten:
          MakeMonsterMove(Cavern,
MonsterPosition, FlaskPosition, PlayerPosition)
          Eaten =
CheckIfSameCell(MonsterPosition,
PlayerPosition)
```

		print '' raw_input("Press Enter key to continue") DisplayCavern(Cavern, MonsterAwake) Count += 1 if Eaten: DisplayLostGameMessage() **Score = Score - 50** **print "Your score was: ",Score**	

04	**2**	**One mark for AO3 (evaluate)** **Info for examiner:** Must match code from 04.1, including prompts on screen capture matching those in code. Code for 04.1 must be sensible. E `  -------------` `|						|` ` -------------` `|						|` ` -------------` `|						|` ` -------------` `|						|` ` -------------` `|		|M| |*| |` ` -------------` Well Done! You have found the flask containing the Styxian potion. You have won the game of MONSTER! Your score was: 70 **1 mark:** Screen capture(s) showing correct cavern state followed by message 'Your score was: 70';	1

<table>
<tr>
<td>05</td>
<td>1</td>
<td>
3 marks for AO3 (design) and 9 marks for AO3 (programming)

<u>Mark Scheme</u>
<table>
<tr><th>Level</th><th>Description</th><th>Mark Range</th></tr>
<tr><td>4</td><td>A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution that is efficient and makes use of nested loops to iterate through the required cells in the array to test for the presence of both traps. A formal interface is used to pass at least some of the required data into and out of the subroutine. All of the appropriate design decisions have been taken.</td><td>10-12</td></tr>
<tr><td>3</td><td>There is evidence that a line of reasoning has been followed to produce a logically structured subroutine that either works correctly in most cases (e.g. some cells may be missed from the checks or only one trap may be checked for) or works correctly in all cases but is not efficient (e.g. multiple IF statements used instead of nested loops). A formal subroutine interface may or may not have been used. The solution demonstrates good design work as most of the correct design decisions have been taken.</td><td>7-9</td></tr>
<tr><td>2</td><td>A subroutine has been created and some appropriate, syntactically correct programming language statements have been written. There is evidence that a line of reasoning has been partially followed as although the subroutine may not have the required functionality, it can be seen that the response contains some of the statements that would be needed in a working solution. There is evidence of some appropriate design work as the response recognises at least one appropriate technique that could be used by a working solution, regardless of whether this has been implemented correctly.</td><td>4-6</td></tr>
<tr><td>1</td><td>A subroutine has been created and some appropriate programming language statements have been written but there is no evidence that a line of reasoning has been followed to arrive at a working solution. The statements written may or may not be syntactically correct and the subroutine will have very little or none of the</td><td>1-3</td></tr>
</table>
</td>
<td>12</td>
</tr>
</table>

	required functionality. It is unlikely that any of the key design elements of the task have been recognised.	

Guidance

Evidence of AO3 (design) - 3 points:

Evidence of design to look for in responses:

- Identifying that the use of nested loops is the most efficient way to solve this problem
- Identifying that the appropriate technique to use to solve the problem is to set the value of a flag to an initial value and then change this if a trap is found
- Identifying that there are two traps, both of which must be checked for

Note that AO3 (design) points are for selecting appropriate techniques to use to solve the problem, so should be credited whether the syntax of programming language statements is correct or not and regardless of whether the solution works.

Evidence of AO3 (programming) – 9 points:

Evidence of programming to look for in responses:

- `TrapDetector` subroutine created – with begin and end of subroutine
- `TrapPositions` and `PlayerPosition` passed as parameters to the `TrapDetector` subroutine
- `True/False` returned by subroutine
- Initial value of flag set to keep track of whether trap detected
- Use of one loop to iterate through some of the cells
- Use of nested loops to iterate through all of the cells
- Selection statement to check for `Trap1` in a specific cell
- Selection statement to check for `Trap2` in a specific cell
- Value of flag changes if a trap detected

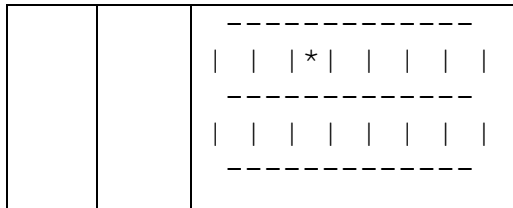
Less efficient solutions may use multiple IF statements instead of loops to check the required cells.

Note that AO3 (programming) points are for programming and so should only be awarded for syntactically correct code.

		Example Solution - Python 2.6 `def TrapDetector(TrapPositions, PlayerPosition):` `  TrapFound = False` `  for Count1 in range(PlayerPosition.NoOfCellsSouth-1,PlayerPosition.NoOfCellsSouth+2):` `  for Count2 in range(PlayerPosition.NoOfCellsEast-1,PlayerPosition.NoOfCellsEast+2):` `  if TrapPositions[0].NoOfCellsEast == Count2 and TrapPositions[0].NoOfCellsSouth == Count1:` `  TrapFound = True` `  if TrapPositions[1].NoOfCellsEast == Count2 and TrapPositions[1].NoOfCellsSouth == Count1:` `  TrapFound = True` `  return TrapFound`	

05	**2**	**Mark if for AO3 (programming)** **Python 2.6:** `  if MoveDirection != 'M':` `  MakeMove(Cavern, MoveDirection, PlayerPosition)` `  DisplayCavern(Cavern, MonsterAwake)` **`  if TrapDetector(TrapPositions, PlayerPosition):`** **`  print "Trap detected"`** **`  else:`** **`  print "No trap detected"`** `  FlaskFound = CheckIfSameCell(PlayerPosition, FlaskPosition)` `  if FlaskFound:` `  DisplayWonGameMessage()` `  Eaten = CheckIfSameCell(MonsterPosition, PlayerPosition)` **Marking:** **1 mark:** Call to `TrapDetector` subroutine in correct place and message displayed in correct circumstances;	1

05	**3**	**Mark is for AO3 (evaluate)** **Info for examiner:** Must match code from 05.1 and 05.2, including prompts on screen capture matching those in code. Code for 05.1 and 05.2 must be sensible. W ------------- | | | | | | | ------------- | | | | | | | ------------- | | | |*| | | ------------- | | | | | | | ------------- | | | | | | | ------------- Trap detected S ------------- | | | | | | | ------------- | | | | | | | ------------- | | | | | | | ------------- | | | |*| | | ------------- | | | | | | | ------------- Trap detected W ------------- | | | | | | | ------------- | | | | | | | ------------- | | | | | | |	1

		 **1 mark:** Screen capture(s) for all three tests cases, showing correct cavern states followed by correct messages;	

05	**4**	**All marks AO2 (analyse)** Cavern: **1 mark:** Cavern variable will need a symbol to represent rock // use 'R' to represent rock in the cavern variable; ResetCavern: **Max 2 marks:** any 2 from: When looking at the outer cells; randomly select if cell is to be rock; mark this cell as rock using a set symbol; CheckValidMove: **Max 2 marks:** any 2 from: Alter function to pass in cavern parameter; Check if move will take player into a cell that is rock; if so return False;	5

05	**5**	**All marks AO3 (evaluate)** The whole cavern might end up being rock; There might be insufficient spaces to place all the items in; The player might be trapped (surrounded by rock); The monster might be trapped (surrounded by rock); The flask might be inaccessible; **Max 2, any 2 from 4 above**	2

C#

01	**1**	`int HowFar;` `int MyLoop;` `Console.WriteLine("How far to count?");` `HowFar = int.Parse(Console.ReadLine());` `while (HowFar < 1)` `{` `  Console.WriteLine("Not a valid number, please try again.");` `  HowFar = int.Parse(Console.ReadLine());` `}` `for (MyLoop = 1; MyLoop < HowFar+1; MyLoop++)` `{` `  if (MyLoop % 3 == 0 && MyLoop % 5 == 0)` `  {` `  Console.WriteLine("FizzBuzz");` `  }` `  else` `  {` `  if (MyLoop % 3 == 0)` `  {` `  Console.WriteLine("Fizz");` `  }` `  else` `  {` `  if (MyLoop % 5 == 0)` `  {` `  Console.WriteLine("Buzz");` `  }` `  else` `  Console.WriteLine(MyLoop);` `}`	8

03	**1**	`public static Boolean CheckValidMove(CellReference PlayerPosition, char Direction)`	8

		{ ValidMove = true; if (!(Direction == 'N' || Direction == 'S' || Direction == 'W' || Direction == 'E' || Direction == 'M')) { ValidMove = false; } **if (PlayerPosition.NoOfCellsSouth == 0 && Direction == 'N')** **{** **ValidMove = false;** **}** return ValidMove; }	

03	**2**	..... MoveDirection = ''; DisplayCavern(Cavern, MonsterAwake); while (!(Eaten || FlaskFound || MoveDirection == 'M')) { ValidMove = false; while (!ValidMove) { DisplayMoveOptions(); MoveDirection = GetMove(); ValidMove = CheckValidMove(PlayerPosition, MoveDirection); **if (!ValidMove)** **{** **Console.WriteLine("That is not a valid move, please try again.");** **}** } if (MoveDirection != 'M') {	8

04	**1**	public static void PlayGame(char[,] Cavern, CellRefence[] TrapPositions, CellReference MonsterPosition, CellReference PlayerPosition, CellReference FlaskPosition, Boolean MonsterAwake)	8

```
{
  int Score = 0;
  Boolean Eaten = false;
  Boolean FlaskFound = false;
  char MoveDirection = '';
  DisplayCavern(Cavern, MonsterAwake);
  while (!(Eaten || FlaskFound || (MoveDirection == 
'M'))
  {
    ValidMove = false;
    while (!ValidMove)
    {
      DisplayMoveOptions();
      MoveDirection = GetMove();
      ValidMove = CheckValidMove(PlayerPosition, 
MoveDirection);
      if (!ValidMove)
      {
          Console.WriteLine("That is not a valid 
move, please try again.");
      }
    }
    if (MoveDirection != 'M')
    {
      Score = Score + 10;
      MakeMove(Cavern, MoveDirection, 
PlayerPosition);
      DisplayCavern(Cavern, MonsterAwake);
      FlaskFound = CheckIfSameCell(PlayerPosition, 
FlaskPosition);
      if (FlaskFound)
      {
        DisplayWonGameMessage();
        Score = Score + 50;
        Console.WriteLine("Your score was: " + 
Score);
      }
      Eaten = CheckIfSameCell(MonsterPosition, 
PlayerPosition);
      if (!MonsterAwakes && !FlaskFound && !Eaten)
      {
        MonsterAwake = 
CheckIfSameCell(PlayerPosition, TrapPositions[0]);
        if (!MonsterAwake)
        {
          MonsterAwake = 
CheckIfSameCell(PlayerPosition, TrapPositions[1]);
        }
        if (MonsterAwake)
```

```
          {
            DisplayTrapMessage();
            Score = Score - 10;
            DisplayCavern(Cavern, MonsterAwake);
          }
        }
        if (MonsterAwake && !Eaten && !FlaskFound)
        {
          Count = 0;
          while (Count < 2 && !Eaten)
          {
            MakeMonsterMove(Cavern, MonsterPosition,
FlaskPosition, PlayerPosition);
            Eaten = CheckIfSameCell(MonsterPosition,
PlayerPosition);
            Console.WriteLine();
            Console.ReadLine("Press Enter key to
continue");
            DisplayCavern(Cavern, MonsterAwake);
            Count = Count + 1;
          }
        }
        if (Eaten)
        {
          DisplayLostGameMessage();
          Score = Score - 50;
          Console.WriteLine("Your score was: " +
Score);
        }
      }
    }
}
```

05	**1**	`public static Boolean TrapDetector(CellReference[] TrapPositions, CellReference PlayerPosition)` `{` `  Boolean TrapFound = false;` `  int Count1;` `  int Count2;` `  for (Count1 = PlayerPosition.NoOfCellsSouth - 1; Count1 < PlayerPosition.NoOfCellsSouth + 2; Count1++)` `  {` `  for (Count2 = PlayerPosition.NoOfCellsEast - 1; Count2 < PlayerPosition.NoOfCellsEast + 2; Count2++)` `  {` `  if (TrapPositions[0].NoOfCellsEast ==`	**8**

		Count2 && TrapPositions[0].NoOfCellsSouth == Count1) { TrapFound = true; } if (TrapPositions[1].NoOfCellsEast == Count2 && TrapPositions[1].NoOfCellsSouth == Count1) { TrapFound = true; } } } return TrapFound; }	

05	**2**	if (MoveDirection != 'M') { MakeMove(Cavern, MoveDirection, PlayerPosition); DisplayCavern(Cavern, MonsterAwake); **if (TrapDetector(TrapPositions, PlayerPosition))** **{** **Console.WriteLine("Trap detected");** **}** **else** **{** **Console.WriteLine("No trap detected");** **}** FlaskFound = CheckIfSameCell(PlayerPosition, FlaskPosition); if (FlaskFound) { DisplayWonGameMessage(); } Eaten = CheckIfSameCell(MonsterPosition, PlayerPosition); }	8

Java

01	**1**	``` int howFar; int myLoop; Scanner in = new Scanner(System.in); System.out.println("How far to count?"); System.out.println("Enter i Value: "); howFar = in.nextInt(); while (howFar < 1) { System.out.println("Not a valid number, please try again."); howFar = in.nextInt(); } for (myLoop = 1; myLoop < howFar+1; myLoop++) { if (myLoop % 3 == 0 && myLoop % 5 == 0) { System.out.println("FizzBuzz"); } else { if (myLoop % 3 == 0) { System.out.println("Fizz"); } else { if (myLoop % 5 == 0) { System.out.println("Buzz"); } else System.out.println(myLoop); } } } ``` If students use the AQAConsole module, a possible solution : ``` int howFar; int myLoop; console.println("How far to count?"); howFar = console.readInteger(); while (howFar < 1) { console.println("Not a valid number, please try again."); howFar = console.readInteger()); ```	8

```
}
for (myLoop = 1; myLoop < howFar+1; myLoop++)
{
    if (myLoop % 3 == 0 && myLoop % 5 == 0)
    {
        console.println("FizzBuzz");
    }
    else
    {
        if (myLoop % 3 == 0)
        {
            console.println("Fizz");
        }
        else
        {
            if (myLoop % 5 == 0)
            {
                console.println("Buzz");
            }
            else
                console.println(myLoop);
       }
    }
}
```

03 **1**

```
public boolean checkValidMove(CellReference
playerPosition, char direction)
{
    validMove = true;
    if (!(direction == 'N' || direction == 'S' ||
            direction == 'W' || direction == 'E' ||
                                          direction
== 'M'))
    {
        validMove = false;
    }
    if (playerPosition.noOfCellsSouth == 0 &&
direction == 'N')
    {
        validMove = false;
    }
    return validMove;
}
```

8

03	**2**	`.....` `moveDirection = '';` `displayCavern(cavern, monsterAwake);` `while (!(eaten || flaskFound || moveDirection == 'M'))` `{` `  validMove = false;` `  while (!validMove)` `  {` `  displayMoveOptions();` `  moveDirection = getMove();` `  validMove = checkValidMove(playerPosition, moveDirection);` **`  if (!validMove)`** **`  {`** **`  console.println("That is not a valid move, please try again.");`** **`  }`** `  }` `  if (moveDirection != 'M')` `  {` `.....`	8

04	**1**	`public void playGame(char[][] cavern, CellRefence[] trapPositions, CellReference monsterPosition, CellReference playerPosition, CellReference flaskPosition, boolean monsterAwake)` `{` **`  int score = 0;`** `  boolean eaten = false;` `  boolean flaskFound = false;` `  char moveDirection = '';` `  displayCavern(cavern, monsterAwake);` `  while (!(eaten || flaskFound || (moveDirection == 'M'))` `  {` `  validMove = false;` `  while (!validMove)` `  {` `  displayMoveOptions();` `  moveDirection = getMove();` `  validMove = checkValidMove(playerPosition, moveDirection);` `  if (!validMove)` `  {` `  console.println("That is not a valid move, please try again.");` ` }	8

```
    }
    if (moveDirection != 'M')
    {
      score = score + 10;
      makeMove(cavern, moveDirection,
playerPosition);
      displayCavern(cavern, monsterAwake);
      flaskFound = checkIfSameCell(playerPosition,
flaskPosition);
      if (flaskFound)
      {
        displayWonGameMessage();
        score = score + 50;
        console.println("Your score was: " + Score);
      }
      eaten = checkIfSameCell(monsterPosition,
playerPosition);
      if (!monsterAwakes && !flaskFound && !eaten)
      {
        monsterAwake =
checkIfSameCell(playerPosition, trapPositions[0]);
        if (!monsterAwake)
        {
          monsterAwake =
checkIfSameCell(playerPosition, trapPositions[1]);
        }
        if (monsterAwake)
        {
          displayTrapMessage();
          score = score - 10;
          displayCavern(cavern, monsterAwake);
        }
      }
      if (monsterAwake && !eaten && !flaskFound)
      {
        count = 0;
        while (count < 2 && !eaten)
        {
          makeMonsterMove(cavern, monsterPosition,
flaskPosition, playerPosition);
          eaten = checkIfSameCell(monsterPosition,
playerPosition);
          console.println();
          console.readLine("Press Enter key to
continue");
          displayCavern(cavern, monsterAwake);
          count = count + 1;
        }
      }
```

		`if (eaten)` `{` `displayLostGameMessage();` **`score = score - 50;`** **`console.println("Your score was: " + Score);`** `}` `}` `}` `}`	

05	**1**	`public boolean trapDetector(CellReference[] trapPositions, CellReference playerPosition)` `{` `boolean trapFound = false;` `int count1;` `int count2;` `for (count1 = playerPosition.noOfCellsSouth - 1; count1 < playerPosition.noOfCellsSouth + 2; count1++)` `{` `for (count2 = playerPosition.noOfCellsEast - 1; count2 < playerPosition.noOfCellsEast + 2; count2++)` `{` `if (trapPositions[0].noOfCellsEast == count2 && trapPositions[0].noOfCellsSouth == count1)` `{` `trapFound = true;` `}` `if (trapPositions[1].noOfCellsEast == count2 && trapPositions[1].noOfCellsSouth == count1)` `{` `trapFound = true;` `}` `}` `}` `return trapFound;` `}`	8

05	**2**	`if (moveDirection != 'M')` `{` `makeMove(cavern, moveDirection, playerPosition);` `displayCavern(cavern, monsterAwake);`	8

```
      if (trapDetector(trapPositions,
playerPosition))
      {
          console.println("Trap detected");
      }
      else
      {
          console.println("No trap detected");
      }
      flaskFound = checkIfSameCell(playerPosition,
flaskPosition);
      if (flaskFound)
      {
        displayWonGameMessage();
      }
      eaten = checkIfSameCell(monsterPosition,
playerPosition);
  }
```

Pascal

01	**1**	`program FizzBuzz(input,output);` `var` `  HowFar,MyLoop : Integer;` `begin` `  writeln('How far to count?');` `  readln(HowFar);` `  while HowFar < 1 Do` `  readln(HowFar);` `  for MyLoop := 1 to HowFar do` `  begin` `  if (MyLoop Mod 3 = 0) And (MyLoop Mod 5 = 0) then` `  writeln('FizzBuzz')` `  else if MyLoop Mod 3 = 0 then` `  writeln('Fizz')` `  else if MyLoop Mod 5 = 0 then` `  writeln('Buzz')` `  else writeln(MyLoop);` `  end;` `end.`	8

03	**1**	`Function CheckValidMove(PlayerPosition : TCellReference; Direction : Char) : Boolean;` `  Var` `  ValidMove : Boolean;` `  Begin` `  ValidMove := True;` `  If Not (Direction In ['N','S','W','E','M'])` `  Then ValidMove := False;` **`  If (PlayerPosition.NoOfCellsSouth = 1) And (Direction = 'N')`** **`  Then ValidMove := False;`** `  CheckValidMove := ValidMove;` `  End;`	8

03	**2**	`DisplayCavern(Cavern, MonsterAwake);` `  Repeat` `  Repeat` `  DisplayMoveOptions;` `  MoveDirection := GetMove;` `  ValidMove := CheckValidMove(PlayerPosition, MoveDirection);`	8

		If not(ValidMove) **Then writeln('That is not a valid move, please try again.');** Until ValidMove; If MoveDirection <> 'M' Then Begin	

04	**1**	Procedure PlayGame(Var Cavern : TCavern; TrapPositions : TTrapPositions; Var MonsterPosition, PlayerPosition, FlaskPosition : TCellReference; Var MonsterAwake : Boolean); Var Count : Integer; Eaten : Boolean; FlaskFound : Boolean; MoveDirection : Char; ValidMove : Boolean; **Score : Integer;** Begin **Score := 0;** Eaten:= False; FlaskFound := False; DisplayCavern(Cavern, MonsterAwake); Repeat Repeat DisplayMoveOptions; MoveDirection := GetMove; ValidMove := CheckValidMove(PlayerPosition, MoveDirection); If not(ValidMove) Then writeln('That is not a valid move, please try again.'); Until ValidMove; If MoveDirection <> 'M' Then Begin **Score := Score + 10;** MakeMove(Cavern, MoveDirection, PlayerPosition); DisplayCavern(Cavern, MonsterAwake); FlaskFound := CheckIfSameCell(PlayerPosition, FlaskPosition); If FlaskFound Then Begin DisplayWonGameMessage;	8

```
                    Score := Score + 50;
                    writeln('Your score was: ',Score);
                  End;
              Eaten :=
CheckIfSameCell(MonsterPosition, PlayerPosition);
              If Not MonsterAwake And Not FlaskFound
And Not Eaten
                Then
                  Begin
                    MonsterAwake :=
CheckIfSameCell(PlayerPosition, TrapPositions[1]);
                    If Not MonsterAwake
                      Then MonsterAwake :=
CheckIfSameCell(PlayerPosition, TrapPositions[2]);
                    If MonsterAwake
                      Then
                        Begin
                          DisplayTrapMessage;
                          Score := Score - 10;
                          DisplayCavern(Cavern,
MonsterAwake);
                        End;
                  End;
              If MonsterAwake And Not Eaten And Not
FlaskFound
                Then
                  Begin
                    Count := 0;
                    Repeat
                      MakeMonsterMove(Cavern,
MonsterPosition, FlaskPosition, PlayerPosition);
                      Eaten :=
CheckIfSameCell(MonsterPosition, PlayerPosition);
                      Writeln;
                      Writeln('Press Enter key to
continue');
                      Readln;
                      DisplayCavern(Cavern,
MonsterAwake);
                      Count := Count + 1;
                    Until (Count = 2) Or Eaten;
                  End;
                If Eaten
                  Then
                    Begin
                      DisplayLostGameMessage;
                      Score := Score - 50;
                      writeln('Your score was:
',Score);
```

		end; End; Until Eaten Or FlaskFound Or (MoveDirection = 'M'); End;	

05	**1**	Function TrapDetector(TrapPositions : TTrapPositions;PlayerPosition : TCellReference):Boolean; Var TrapFound : Boolean; Count1, Count2 : Integer; Begin TrapFound := False; For Count1 := PlayerPosition.NoOfCellsSouth - 1 To PlayerPosition.NoOfCellsSouth + 1 Do For Count2 := PlayerPosition.NoOfCellsEast - 1 To PlayerPosition.NoOfCellsEast + 1 Do Begin If (TrapPositions[1].NoOfCellsEast = Count2) And (TrapPositions[1].NoOfCellsSouth = Count1) Then TrapFound := True ; If (TrapPositions[2].NoOfCellsEast = Count2) And (TrapPositions[2].NoOfCellsSouth = Count1) Then TrapFound := True; End; TrapDetector := TrapFound; End;	8

05	**2**	Score := Score + 10; MakeMove(Cavern, MoveDirection, PlayerPosition); DisplayCavern(Cavern, MonsterAwake); **If TrapDetector(TrapPositions, PlayerPosition)** **Then writeln('Trap detected')** **Else writeln('No trap detected');** FlaskFound := CheckIfSameCell(PlayerPosition, FlaskPosition); If FlaskFound Then	8

VB.NET

01	**1**	Module Module1 Sub Main() Dim HowFar As Integer Dim MyLoop As Integer Console.WriteLine("How far to count?") HowFar = Console.ReadLine() While HowFar < 1 Console.WriteLine("Not a valid number, please try again.") HowFar = Console.ReadLine() End While For MyLoop = 1 To HowFar If MyLoop Mod 3 = 0 And MyLoop Mod 5 = 0 Then Console.WriteLine("FizzBuzz") ElseIf MyLoop Mod 3 = 0 Then Console.WriteLine("Fizz") ElseIf MyLoop Mod 5 = 0 Then Console.WriteLine("Buzz") Else Console.WriteLine(MyLoop) End If Next Console.ReadLine() End Sub End Module	8

03	**1**	Function CheckValidMove(ByVal PlayerPosition As CellReference, ByVal Direction As Char) As Boolean Dim ValidMove As Boolean ValidMove = True If Not (Direction = "N" Or Direction = "S" Or Direction = "W" Or Direction = "E" Or Direction = "M") Then ValidMove = False End If **If PlayerPosition.NoOfCellsSouth = 1 And Direction = "N" Then** **ValidMove = False** **End If** CheckValidMove = ValidMove End Function	8

03	**2**	Eaten = False FlaskFound = False DisplayCavern(Cavern, MonsterAwake)	8

```
        Do
            Do
                DisplayMoveOptions()
                MoveDirection = GetMove()
                ValidMove =
CheckValidMove(PlayerPosition, MoveDirection)
                If Not (ValidMove) Then
                    Console.WriteLine("That is not a
valid move, please try again.")
                End If
            Loop Until ValidMove
            If MoveDirection <> "M" Then
                MakeMove(Cavern, MoveDirection,
PlayerPosition)
```

04	**1**	Sub PlayGame(ByRef Cavern(,) As Char, ByVal TrapPositions() As CellReference, ByRef MonsterPosition As CellReference, ByRef PlayerPosition As CellReference, ByRef FlaskPosition As CellReference, ByRef MonsterAwake As Boolean) Dim Count As Integer Dim Eaten As Boolean Dim FlaskFound As Boolean Dim MoveDirection As Char Dim ValidMove As Boolean **Dim Score As Integer** **Score = 0** Eaten = False FlaskFound = False DisplayCavern(Cavern, MonsterAwake) Do Do DisplayMoveOptions() MoveDirection = GetMove() ValidMove = CheckValidMove(PlayerPosition, MoveDirection) If Not (ValidMove) Then Console.WriteLine("That is not a valid move, please try again.") End If Loop Until ValidMove If MoveDirection <> "M" Then **Score = Score + 10** MakeMove(Cavern, MoveDirection, PlayerPosition) DisplayCavern(Cavern, MonsterAwake) FlaskFound = CheckIfSameCell(PlayerPosition, FlaskPosition) If FlaskFound Then DisplayWonGameMessage()	8

```
                    Score = Score + 50
                    Console.WriteLine("Your score
was: " & Score)
                End If
                Eaten =
CheckIfSameCell(MonsterPosition, PlayerPosition)
                If Not MonsterAwake And Not
FlaskFound And Not Eaten Then
                    MonsterAwake =
CheckIfSameCell(PlayerPosition, TrapPositions(1))
                    If Not MonsterAwake Then
                        MonsterAwake =
CheckIfSameCell(PlayerPosition, TrapPositions(2))
                    End If
                    If MonsterAwake Then
                        DisplayTrapMessage()
                        Score = Score - 10
                        DisplayCavern(Cavern,
MonsterAwake)
                    End If
                End If
                If MonsterAwake And Not Eaten And
Not FlaskFound Then
                    Count = 0
                    Do
                        MakeMonsterMove(Cavern,
MonsterPosition, FlaskPosition, PlayerPosition)
                        Eaten =
CheckIfSameCell(MonsterPosition, PlayerPosition)
                        Console.WriteLine()
                        Console.WriteLine("Press
Enter key to continue")
                        Console.ReadLine()
                        DisplayCavern(Cavern,
MonsterAwake)
                        Count = Count + 1
                    Loop Until Count = 2 Or Eaten
                End If
                If Eaten Then
                    DisplayLostGameMessage()
                    Score = Score - 50
                    Console.WriteLine("Your score
was: " & Score)
                End If
            End If
        Loop Until Eaten Or FlaskFound Or
MoveDirection = "M"
    End Sub
```

05	**1**	Function TrapDetector(ByVal TrapPositions() As CellReference, ByVal PlayerPosition As	8

		CellReference) As Boolean Dim TrapFound As Boolean TrapFound = False Dim Count1, Count2 As Integer For Count1 = PlayerPosition.NoOfCellsSouth - 1 To PlayerPosition.NoOfCellsSouth + 1 For Count2 = PlayerPosition.NoOfCellsEast - 1 To PlayerPosition.NoOfCellsEast + 1 If TrapPositions(1).NoOfCellsEast = Count2 And TrapPositions(1).NoOfCellsSouth = Count1 Then TrapFound = True End If If TrapPositions(2).NoOfCellsEast = Count2 And TrapPositions(2).NoOfCellsSouth = Count1 Then TrapFound = True End If Next Next TrapDetector = TrapFound End Function	

05	**2**	If MoveDirection <> "M" Then Score = Score + 10 MakeMove(Cavern, MoveDirection, PlayerPosition) DisplayCavern(Cavern, MonsterAwake) **If TrapDetector(TrapPositions, PlayerPosition) Then** **Console.WriteLine("Trap detected")** **Else** **Console.WriteLine("No trap detected")** **End If** FlaskFound = CheckIfSameCell(PlayerPosition, FlaskPosition) If FlaskFound Then	8

Python 3

01	**1**	`print ("How far to count?")` `HowFar = int(input())` `while HowFar < 1:` `  print ("Not a valid number, please try again.")` `  HowFar = int(input())` `  for MyLoop in range(1,HowFar+1):` `  if MyLoop%3 == 0 and MyLoop%5 == 0:` `  print ("FizzBuzz")` `  elif MyLoop%3 == 0:` `  print ("Fizz")` `  elif MyLoop%5 == 0:` `  print ("Buzz")` `  else:` `  print (MyLoop)`	8

03	**1**	`def CheckValidMove(PlayerPosition,Direction):` `  ValidMove = True` `  if not (Direction in ['N','S','W','E','M']):` `  ValidMove = False` **`  if PlayerPosition.NoOfCellsSouth == 0 and Direction == 'N':`** **`  ValidMove = False`** `  return ValidMove`	8

03	**2**	`while not ValidMove:` `  DisplayMoveOptions()` `  MoveDirection = GetMove()` `  ValidMove = CheckValidMove(PlayerPosition, MoveDirection)` **`  if not (ValidMove):`** **`  print('That is not a valid move, please try again.')`** `  if MoveDirection != 'M':`	8

04	**1**	`def PlayGame(Cavern, TrapPositions, MonsterPosition, PlayerPosition, FlaskPosition, MonsterAwake):` **`  Score = 0`** `  Eaten = False` `  FlaskFound = False` `  MoveDirection = ''` `  DisplayCavern(Cavern, MonsterAwake)` `  while not (Eaten or FlaskFound or (MoveDirection == 'M')):`	8

```
    ValidMove = False

    while not ValidMove:
      DisplayMoveOptions()
      MoveDirection = GetMove()
      ValidMove = CheckValidMove(PlayerPosition,
MoveDirection)
      if not (ValidMove):
          print('That is not a valid move, please
try again.')
    if MoveDirection != 'M':
      Score = Score + 10
      MakeMove(Cavern, MoveDirection,
PlayerPosition)
      DisplayCavern(Cavern, MonsterAwake)
      FlaskFound = CheckIfSameCell(PlayerPosition,
FlaskPosition)
      if FlaskFound:
        DisplayWonGameMessage()
        Score = Score + 50
        print('Your score was:',Score)
      Eaten = CheckIfSameCell(MonsterPosition,
PlayerPosition)
      if not MonsterAwake.Is and not FlaskFound and
not Eaten:
        MonsterAwake.Is =
CheckIfSameCell(PlayerPosition, TrapPositions[1])
        if not MonsterAwake.Is:
          MonsterAwake.Is =
CheckIfSameCell(PlayerPosition, TrapPositions[2])
        if MonsterAwake.Is:
          DisplayTrapMessage()
          Score = Score - 10
          DisplayCavern(Cavern, MonsterAwake)
      if MonsterAwake.Is and not Eaten and not
FlaskFound:
        Count = 0
        while Count < 2 and not Eaten:
          MakeMonsterMove(Cavern, MonsterPosition,
FlaskPosition, PlayerPosition)
          Eaten = CheckIfSameCell(MonsterPosition,
PlayerPosition)
          print('')
          input('Press Enter key to continue')
          DisplayCavern(Cavern, MonsterAwake)
          Count += 1
      if Eaten:
        DisplayLostGameMessage()
        Score = Score - 50
```

		print('Your score was:',Score)	

05	**1**	def TrapDetector(TrapPositions,PlayerPosition): TrapFound = False for Count1 in range(PlayerPosition.NoOfCellsSouth-1,PlayerPosition.NoOfCellsSouth+2): for Count2 in range(PlayerPosition.NoOfCellsEast-1,PlayerPosition.NoOfCellsEast+2): if TrapPositions[0].NoOfCellsEast == Count2 and TrapPositions[0].NoOfCellsSouth == Count1: TrapFound = True if TrapPositions[1].NoOfCellsEast == Count2 and TrapPositions[1].NoOfCellsSouth == Count1: TrapFound = True return TrapFound	8

05	**2**	if MoveDirection != 'M': Score = Score + 10 MakeMove(Cavern, MoveDirection, PlayerPosition) DisplayCavern(Cavern, MonsterAwake) **if TrapDetector(TrapPositions,PlayerPosition):** **print('Trap detected')** **else:** **print('No trap detected')** FlaskFound = CheckIfSameCell(PlayerPosition, FlaskPosition) if FlaskFound:	8