AQA

AS
Computer Science

Paper 1 (7516/1 – applicable for **all** programming languages A, B, C, D and E)
Mark Scheme

7516
June 2016

Version: 1.0 Final

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 aqa.org.uk

AS Computer Science

Paper 1 (7516/1 – applicable to **all** programming languages A, B C, D and E)

June 2016

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 - means "Don't penalise twice". 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.

Pages 2 to 19 contain the generic mark scheme.

Pages 20 to 37 contain the 'Program Source Codes' specific to the programming languages for questions 5.1, 9.1, 10.1, 10.2 and 11.1;

pages 20 to 22 – VB.NET
pages 23 to 25 – PASCAL/Delphi
pages 26 to 29 – C#
pages 30 to 32 – JAVA
pages 33 to 35 – PYTHON 2
pages 36 to 37 – PYTHON 3

Level of response marking instructions

Level of response mark schemes are broken down into levels, each of which has a descriptor. The descriptor for the level shows the average performance for the level. There are marks in each level.

Before you apply the mark scheme to a student's answer read through the answer and annotate it (as instructed) to show the qualities that are being looked for. You can then apply the mark scheme.

Step 1 Determine a level

Start at the lowest level of the mark scheme and use it as a ladder to see whether the answer meets the descriptor for that level. The descriptor for the level indicates the different qualities that might be seen in the student's answer for that level. If it meets the lowest level then go to the next one and decide if it meets this level, and so on, until you have a match between the level descriptor and the answer. With practice and familiarity you will find that for better answers you will be able to quickly skip through the lower levels of the mark scheme.

When assigning a level you should look at the overall quality of the answer and not look to pick holes in small and specific parts of the answer where the student has not performed quite as well as the rest. If the answer covers different aspects of different levels of the mark scheme you should use a best fit approach for defining the level and then use the variability of the response to help decide the mark within the level, ie if the response is predominantly level 3 with a small amount of level 4 material it would be placed in level 3 but be awarded a mark near the top of the level because of the level 4 content.

Step 2 Determine a mark

Once you have assigned a level you need to decide on the mark. The descriptors on how to allocate marks can help with this. The exemplar materials used during standardisation will help. There will be an answer in the standardising materials which will correspond with each level of the mark scheme. This answer will have been awarded a mark by the Lead Examiner. You can compare the student's answer with the example to determine if it is the same standard, better or worse than the example. You can then use this to allocate a mark for the answer based on the Lead Examiner's mark on the example.

You may well need to read back through the answer as you apply the mark scheme to clarify points and assure yourself that the level and the mark are appropriate.

Indicative content in the mark scheme is provided as a guide for examiners. It is not intended to be exhaustive and you must credit other valid points. Students do not have to cover all of the points mentioned in the Indicative content to reach the highest level of the mark scheme.

An answer which contains nothing of relevance to the question must be awarded no marks.

Examiners are required to assign each of the candidates' responses to the most appropriate level according to **its overall quality**, then allocate a single mark within the level. When deciding upon a mark in a level examiners should bear in mind the relative weightings of the assessment objectives

eg

In question **10.1**, the marks available for the AO3 elements are as follows:

AO3 (design) – 2 marks
AO3 (programming) – 6 marks

In question **11.1** the marks available for the AO3 elements are as follows:

AO3 (design) – 2 marks
AO3 (knowledge) – 10 marks.

Where a candidate's answer only reflects one element of the AO, the maximum mark they can receive will be restricted accordingly.

01	**1**	**Mark is for AO2 (apply)** B;	1
01	**2**	**Mark is for AO2 (apply)** C;	1

02	**1**	**All marks AO2 (apply)** **Mark as follows:** **1 mark:** one row correct **2 marks:** all rows correct	2
02	**2**	**All marks AO2 (apply)** Strings that start with zero or more 1s; **A.** starts with any number of 1s as BOD which may or may not be followed by a 0; **A.** there can be at most one 0 in the string and end with an x; **A.** 'end' being by implication **NOTE:** 'ending with either x or 0x**' is worth two marks** **NOTE: MAX 2** if answer is not fully correct	3

Input string	Accepted by FSM?
111011x	NO
1110x	YES
111001x	NO

03		**All marks AO3 (evaluate)** **Mark as follows:** **1 mark:** one row completed correctly in the table **2 marks:** three rows completed correctly in the table	2

Value for x	Type of test data
25	Valid / normal / typical **A.** expected / acceptable
1	Boundary / extreme
-8	Invalid / erroneous

04 1 **All marks AO2 (apply)** 5

ItemsCount	NewItemsCount	LoopA	Done	LoopB	NewItems			
					[0]	[1]	[2]	[3]
4	1				12	0	0	0
	2	1	False	0		25		
		2	False	0				
			True	1				
		3	False	0				
	3			1			53	

1. `LoopA` running over the values `1`, `2`, `3` and stops at 3;
2. `LoopB` is set to 0 then changes to 1 then changes to 0 then changes to 1 and no further changes;
3. `NewItems` becoming `12`, `25`, `0`, `0` at the end of `LoopA` value `1`;
4. `NewItems` not changing during `LoopA` value `2`;
R. if `NewItems` has not changed previously
5. `NewItems` having value `12`, `25`, `53`, `0` at end of table;

A. `NewItems` without trailing zeroes
A. `NewItems` without repeated values for `12` and `25`

Note: 12 might not be seen in `NewItems[0]` as does not need to be copied across into EAD in which case column should be blank

I. columns `NewItemsCount` and `Done` (first and third columns in EAD)

04 2 **Mark is for AO2 (apply)** 1

`NewItems` contains an array/list of the unique items from the `Items` array/list;
Remove duplicate items from an array/list;

Max 1

05	**1**	**All marks AO3 (programming)** 1. Correct variable declarations for `Value`, `Operation` and `Count`; **Note for examiners** If a language allows variables to be used without explicit declaration (eg Python) then this mark should be awarded if the three correct variables exist in the program code and the first value they are assigned is of the correct data type. 2. Correct prompt "`Enter integer (0-99): `"; 3. `Value` assigned value entered by user; 4. `WHILE` loop, with syntax allowed by the programming language and correct condition for the termination of the loop; 5. Correct syntax for the `IF` statement including condition and `ELSE` part; **A.** use of `ELSE IF` but condition must be correct 6. Correct syntax for `(Value DIV 10) + (Value MOD 10)` and/or `(Value DIV 10) * (Value MOD 10)`; 7. `Count` is given initial value `0` before iteration structure and incremented by `1` inside iteration structure; 8. Correct prompt "`The persistence is: `" and immediately followed by value of `Count` and after iteration structure; **Max 7** If code would not function correctly **Max 7** If brackets are missing for the multiplicative calculation at mark point 6. **I.** Case of variable names, strings and output messages **I.** Spacing in prompts **A.** Minor typos in variable names and output messages **A.** Initialisation of variables at declaration stage	8

05	**2**	**Mark is for AO3 (evaluate)**
		Info for examiners: must match code from 05.1, including prompts on screen capture matching those in code. Code for 05.1 must be sensible.
		
		First Test
		
		`Enter integer (0-99):`
		`47`
		`Calculate additive or multiplicative persistence (a or m)?`
		`m`
		`The persistence is:`
		`3`
		
		Second Test
		
		`Enter integer (0-99):`
		`77`
		`Calculate additive or multiplicative persistence (a or m)?`
		`a`
		`The persistence is:`
		`2`
		
		Mark as follows:
		Both tests showing provided test data being entered, correct choice being made (m/a) and correct persistence value being displayed ;
			1

05	**3**	**Mark is for AO2 (analysis)** The number of times the iteration/while loop will be performed is unknown (at the start of the loop) / not predetermined / indefinite; **N.E.** loop until a condition is met	1

06	**1**	**Mark is for AO1 (understanding)** `Orientation`; **R.** if any additional code **R.** if spelt incorrectly **I.** case	1
06	**2**	**Mark is for AO1 (understanding)** `CheckWin` // `PrintBoard`; C#, VB, Java, Pascal: `SetUpBoard` // `SetUpShips`; **R.** if any additional code (including routine interface) **R.** if spelt incorrectly **I.** case	1
06	**3**	**Mark is for AO1 (understanding)** `ValidateBoatPosition` // `CheckWin`; **R.** if any additional code (including routine interface) **R.** if spelt incorrectly **I.** case	1

07	**1**	**Mark is for AO1 (understanding)** Improves readability of code; Easier to update the programming code if the value changes (**A**. by implication); Reduce the likelihood of inconsistency causing errors; Unlike a variable the value can't be changed / is not mutable; **Max 1**	1
07	**2**	**All marks AO2 (analyse)** **1 mark:** `FOR` loop is used to run across the length of the new ship // Number of iterations/length of ship is known; **1 mark:** program checks if the board already has a ship positioned in this cell // this cell is not empty; **1 mark:** (Use of loop structure) avoids the need for many `IF` statements; **Max 2**	2

07	**3**	**All marks AO2 (analyse)** Code is checking that the end of a ship does not go outside of the board / outside of bounds of array / greater than 10; for a ship placed vertically; **A.** Boat does not go off the <u>bottom</u> of the board ;;	2
07	**4**	**1 mark for AO1 (knowledge) and 2 marks for AO2 (analyse)** **AO1 1 mark:** Exception handling is responding to the occurrence of fatal errors / errors that would cause a crash / runtime errors; **AO2 1 mark:** Could be used to trap any errors when converting user input into an integer // to trap an error if a non-integer entry; **AO2 1 mark:** Code could then ask user to re-enter the value // default value used // automatically count as a miss // display an error message;	3
08	**1**	**Mark is for AO2 (analysis)** `MakePlayerMove`; **R.** if spelt incorrectly **I.** case	1
08	**2**	**Mark is for AO2 (analysis)** `CheckWin`; **R.** if spelt incorrectly **I.** case	1
08	**3**	**Mark is for AO2 (apply)** `GetRowColumn`; **R.** if spelt incorrectly **I.** case	1
08	**4**	**All marks AO1 (understanding)** Decomposition (of a problem/program) // use of top-down approach // Structured programming makes use of subroutines / modules; Use of block structures / compound statements; Structured programming makes use of control structures; **A.** by example - sequence / selection / iteration; Avoidance of use of `goto` statements; **Max 2**	2

08	**5**	**All marks AO1 (understanding)** Because the scope of the two variables is different ;; // Because they are (both) local variables; in different subroutines;	2
08	**6**	**All marks AO1 (understanding)** In text files all data is stored as strings / ASCII values / Unicode values / characters // text files use only byte values that display sensibly on a VDU // stores only values that can be open and read in a text editor; **A.** text file is human-readable Binary files store data using different data types; **A.** By example **A.** Binary files can only be correctly interpreted by application that created them	2

09	**1**	**1 mark for AO3 (design) and 4 marks for AO3 (programming)** Note that AO3 (design) marks 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. **AO3 (design) - 1 mark:** **1.** Identifying that an appropriate technique is required to repeatedly input the data; **AO3 (programming) - 4 marks:** **2.** Check against lower/upper bound for `Row`; **3.** 2. combined correctly with check against other bound; **4.** The message "`Invalid value entered`" is displayed only if invalid value has been entered for at least one of the correct bounds; **I.** Case of output message **A.** Minor typos in output message **I.** Spacing in output message **5.** Function only returns values if row is valid and always returns a value if row is valid; **NOTE:** 2. and 3. could be combined into checking against a range of values and be awarded both marks **NOTE:** Ignore any code validating against column **NOTE:** If no attempt is made at validating `Row` but an attempt is made at validating `Column` then mark points can be accepted for points 1,2,3 if done appropriately for `Column` (stop at MAX 3)	5

09	**2**	**Mark is for AO3 (evaluate)**
		
		****SCREEN CAPTURE****
		Must match code from 09.1, including prompts on screen capture matching those in code. Code for 09.1 must be sensible.
		
		`Please enter column: 6`
		`Please enter row: 10`
		
		`Invalid value entered`
		
		`Please enter column: 6`
		`Please enter row: 9`
		
		`Hit at (6,9)`
		
		Mark as follows
		Screen capture(s) showing the requested test being performed;
		
		NOTE: if they have coded a check to validate column the screenshot might not show the second request for entry of column as 6 would be seen as valid
			1

10	**1**	**2 marks for AO3 (design) and 6 marks for AO3 (programming)**	8

Level	Description	Mark Range
4	A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution. The `Ships` data structure is modified correctly for each hit. The sunk message is displayed at the appropriate time. 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.	7-8
3	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 `Ships` data structure is modified incorrectly 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 ensure that the task is mainly achieved.	5-6
2	There is evidence that a line of reasoning has been partially followed as the `Ships` data structure is modified (though possibly incorrectly). The correct message might not be displayed. There is little or no evidence that a line of reasoning has been followed to award a mark for the design of the solution.	3-4
1	An attempt to modify the `Ships` data structure. This modification may not be in exactly the right place and the value to change the structure by may be incorrect, but it should be possible to see that it was intended to be linked to a particular ship. 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.	1-2

Guidance

Evidence of AO3 (design) - 2 points:

1. designing a suitable interface for the `CheckSunk` subroutine with suitable parameters and return values;

   Suitable parameters could be `board,ships,row,column` OR `ships` and the character from `board[row,column]`;

   JAVA: `rowColumn` could be passed rather than `row` and `column`

2. identifying suitable structures so that every ship can be checked (e.g. loop or 5 `IF` statements);

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) - 6 points:

3. `CheckSunk` subroutine created with begin and end of subroutine;

4. Use the letter stored in the square that has been hit;
   ie use of `board[row][column]` and set to a variable or used with a comparison to a ship name first character

5. Compare the letter from the square (4.) with the first letter of the type of ship ;
   ie either via direct characters such as '`A`','`P`' or using something similar to `ships[][][0]`
   R. if would only work for 2 or fewer ships

6. Take one away from the size of the ship identified as being hit;
   R. if would only work for 2 or fewer ships

7. Selection statements to compare size of the hit ship against 0;
   R. if would only work for 2 or fewer ships

8. Correct message (type of ship followed by `'is sunk!'`) is displayed under the correct circumstances for **all** ships;
   NOTE: For the 'sunk' message look to make sure it is only being outputted at the time of sinking

   I. Case of output message
   A. Minor typos in output message
   I. Spacing in output message

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

NOTE: Code could be written as a function or procedure - either would be acceptable.

10	**2**	**Mark is for AO3 (programming)** **1 mark:** Call to `CheckSunk` is in the correct position in the `MakePlayerMove` subroutine (must be before setting the board cell to `'h'`); **A.** if they have declared a variable to temporarily store the current `board[row][column]` ship character and passed this before or after setting board cell to 'h'	1

10	**3**	**Mark is for AO3 (evaluate)** *****SCREEN CAPTURE***** *Must match code from 10.1 and 10.2, including prompts on screen capture matching those in code. Code for 10.1 and 10.2 must be sensible.* `Please enter column: 1` `Please enter row: 5` `Hit at (1,5)` `Patrol Boat is sunk!` `The board looks like this:` `  0   1   2   3   4   5   6   7   8   9` `0								|` `1								|` `2								|` `3	m							|` `4	h							|` `5	h							|` `6								|` `7								|` `8								|` `9								| **Mark as follows** Screen capture(s) showing the final shot being performed (1, 5), final board layout and the message `'Patrol Boat is sunk!'`; **I**. Message concerning firing a torpedo (candidate may have attempted question 11 before 10)	1

11 **1** **2 marks for AO3 (design) and 10 marks for AO3 (programming)** 12

Note that AO3 (design) marks 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.

Level	Description	Mark Range
4	A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution that is efficient. Code is written to ensure only one torpedo can be fired during a game and offers the user a chance to fire a torpedo if appropriate. When a torpedo is fired it moves correctly. Appropriate messages are displayed. A formal interface is used to pass at least some of the required data into and out of the `MakePlayerTorpedoMove` subroutine. All of the appropriate design decisions have been taken.	10-12
3	There is evidence that a line of reasoning has been followed. The `PlayGame` subroutine has been altered to allow the user to fire a torpedo. The `MakePlayerTorpedoMove` subroutine uses a logically structured subroutine that works correctly in most cases. 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.	7-9
2	The `PlayGame` subroutine has been adapted but it might not ensure that the player can only fire one torpedo. A `MakePlayerTorpedoMove` subroutine structure has been created and there may be some appropriate, syntactically correct programming language statements written. There is evidence that a line of reasoning has been partially followed for `PlayGame` and/or `MakePlayerTorpedoMove` as although there may not be all of 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.	4-6
1	An attempt has been made to alter the `PlayGame` subroutine and/or 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 required functionality. It is unlikely that any of the key design elements of the task have been recognised.	1-3

AO3 (design) - 2 marks:

1. Identification of the need for a variable to hold whether a torpedo has been fired or not;

2. Use of an iteration structure to control the moving of the torpedo;

AO3 (programming) - 10 marks:

PlayGame

3. Logic to determine whether a torpedo can be fired is correct and is in an appropriate place in the `PlayGame` subroutine;

4. Correct prompt '`Fire a torpedo? (Y/N)`' is displayed;

   I. Case of output message
   A. Minor typos in output message
   I. Spacing in output message

5. Selection statement coded to handle a torpedo shot or a normal shot calling the relevant subroutine and code is in appropriate place in the `PlayGame` subroutine;

6. Variable is correctly updated to reflect the fact that a torpedo has been fired;

MakePlayerTorpedoMove

7. A suitable interface for the `MakePlayerTorpedoMove` subroutine with suitable parameters (`board` and possibly `ships`) and return value

8. Code correctly exits iteration when torpedo moves off board and torpedo hits a ship

9. Code will move torpedo up board

10. Code correctly determines when torpedo has hit a ship updating board to '`h`' (when necessary)

11. If current board position is empty cell '`-`' is updated to contain an '`m`' and code behaves correctly for cell already containing an '`m`'.

12. Messages produced appropriately and displayed under correct conditions when the torpedo misses (moves off the board) and hits a ship.

NOTE: `MakePlayerTorpedoMove` could be written as a procedure or function - either would be acceptable

NOTE: A call to `CheckSunk` is not necessary for full marks

11	**2**	**All marks for AO3 (evaluate)** *****SCREEN CAPTURE***** *Must match code from 11.1, including prompts on screen capture matching those in code. Code for 11.1 must be sensible.*	2

```
MAIN MENU

1. Start new game
2. Load training game
9. Quit

Please enter your choice: 2

The board looks like this:

  0   1   2   3   4   5   6   7   8   9
0   |   |   |   |   |   |   |   |   |
1   |   |   |   |   |   |   |   |   |
2   |   |   |   |   |   |   |   |   |
3   |   |   |   |   |   |   |   |   |
4   |   |   |   |   |   |   |   |   |
5   |   |   |   |   |   |   |   |   |
6   |   |   |   |   |   |   |   |   |
7   |   |   |   |   |   |   |   |   |
8   |   |   |   |   |   |   |   |   |
9   |   |   |   |   |   |   |   |   |

Fire a torpedo(Y/N)? Y

Please enter column: 1
Please enter row: 7

Torpedo hits at (1,5)

The board looks like this:

  0   1   2   3   4   5   6   7   8   9
0   |   |   |   |   |   |   |   |   |
1   |   |   |   |   |   |   |   |   |
2   |   |   |   |   |   |   |   |   |
3   |   |   |   |   |   |   |   |   |
4   |   |   |   |   |   |   |   |   |
5   | h |   |   |   |   |   |   |   |
6   | m |   |   |   |   |   |   |   |
7   | m |   |   |   |   |   |   |   |
8   |   |   |   |   |   |   |   |   |
9   |   |   |   |   |   |   |   |   |
```

```
Please enter column: 2
Please enter row: 1

Sorry, (2,1) is a miss.

The board looks like this:

  0   1   2   3   4   5   6   7   8   9
0   |   |   |   |   |   |   |   |   |
1   |   | m |   |   |   |   |   |   |
2   |   |   |   |   |   |   |   |   |
3   |   |   |   |   |   |   |   |   |
4   |   |   |   |   |   |   |   |   |
5   | h |   |   |   |   |   |   |   |
6   | m |   |   |   |   |   |   |   |
7   | m |   |   |   |   |   |   |   |
8   |   |   |   |   |   |   |   |   |
9   |   |   |   |   |   |   |   |   |
```

Mark as follows
1 mark: Screenshots show player is asked if they want to place a torpedo and then the '`Fire a torpedo?`' prompt is not shown for the second shot;

1 mark: Screenshots show the torpedo moves correctly from (1,7) to (1,5) and causes a hit;

NOTE: Do not penalise extra screenshots showing the torpedo moving up square by square

VB.NET

05 **1** 8

```
Dim Value As Integer
Dim Operation As Char
Dim Count As Integer
Console.Write("Enter integer (0-99): ")
Value = Console.ReadLine()
Console.Write("Calculate additive or multiplicative persistence (a
or m)? ")
Operation = Console.ReadLine()
Count = 0
While Value > 9
  If Operation = "a" Then
    Value = (Value \ 10) + (Value Mod 10)
  Else
    Value = (Value \ 10) * (Value Mod 10)
  End If
  Count = Count + 1
End While
Console.Write("The persistence is: ")
Console.Write(Count)
Console.ReadLine()
```

NOTE: must be \ for integer division

09 **1** 5

```
Sub GetRowColumn(ByRef Row As Integer, ByRef Column As Integer)
  Console.WriteLine()
  Console.Write("Please enter column: ")
  Column = Console.ReadLine()
  Do
    Console.Write("Please enter row: ")
    Row = Console.ReadLine()
    If Row < 0 Or Row > 9 Then
      Console.WriteLine("Invalid value entered")
    End If
  Loop Until Row >= 0 And Row <= 9
  Console.WriteLine()
End Sub
```

Alternative (use of while loop instead of do):

```
Sub GetRowColumn(ByRef Row As Integer, ByRef Column As Integer)
  Console.WriteLine()
  Console.Write("Please enter column: ")
  Column = Console.ReadLine()
  Console.Write("Please enter row: ")
  Row = Console.ReadLine()
  While Row < 0 Or Row > 9
    Console.WriteLine("Invalid value entered")
    Console.Write("Please enter row: ")
    Row = Console.ReadLine()
  End While
  Console.WriteLine()
End Sub
```

10	**1**	`Sub CheckSunk(ByVal Row As Integer, ByVal Column As Integer, ByVal Board(,) As Char, ByRef Ships() As TShip)`
		`  Dim ShipType As Char`
		`  ShipType = Board(Row, Column)`
		`  For i = 0 To Ships.Length - 1`
		`  If Ships(i).Name(0) = ShipType Then`
		`  Ships(i).Size -= 1`
		`  If Ships(i).Size = 0 Then`
		`  Console.WriteLine(Ships(i).Name & " is sunk!")`
		`  End If`
		`  End If`
		`  Next`
		`End Sub`
		
		Alternative (use of if / case / select for each ship type):
		
		`Sub CheckSunk(ByVal Row As Integer, ByVal Column As Integer, ByVal Board(,) As Char, ByRef Ships() As TShip)`
		`  Dim ShipIndex as Integer`
		`  If Board[Row, Column] = "A" Then`
		`  ShipIndex = 0`
		`  End If`
		`  If Board[Row, Column] = "B" Then`
		`  ShipIndex = 1`
		`  End If`
		`  If Board[Row, Column] = "S" Then`
		`  ShipIndex = 2`
		`  End If`
		`  If Board[Row, Column] = "D" Then`
		`  ShipIndex = 3`
		`  End If`
		`  If Board[Row, Column] = "P" Then`
		`  ShipIndex = 4`
		`  End If`
		`  Ships(ShipIndex).Size -= 1`
		`  If Ships(ShipIndex).Size = 0 Then`
		`  Console.WriteLine(Ships(ShipIndex).Name & " is sunk!")`
		`  End If`
		`End Sub`
			8

10	**2**	`  ...`
		`  Else`
		`  Console.WriteLine("Hit at (" & Column & "," & Row & ").")`
		`  **CheckSunk(Row, Column, Board, Ships)**`
		`  Board(Row, Column) = "h"`
		`  End If`
		`End While`
		`...`
			1

11	**1**	`Sub PlayGame(ByVal Board(,) As Char, ByVal Ships() As TShip)`
		`  Dim GameWon As Boolean = False`
		`  Dim TorpedoUsed As Boolean = False`
		`  Dim TorpedoChosen As Char`
		`  Do`
		`  PrintBoard(Board)`
		`  If Not TorpedoUsed Then`
		`  Console.Write("Fire a torpedo? (Y/N)")`
		`  TorpedoChosen = Console.ReadLine`
		`  End If`
		`  If TorpedoChosen = "Y" Then`
		`  MakePlayerTorpedoMove(Board, Ships)`
		`  TorpedoChosen = "N"`
		`  TorpedoUsed = True`
		`  Else`
		`  MakePlayerMove(Board, Ships)`
		`  End If`
		`  GameWon = CheckWin(Board)`
		`  If GameWon Then`
		`  Console.WriteLine("All ships sunk!")`
		`  Console.WriteLine()`
		`  End If`
		`  Loop Until GameWon`
		`  End Sub`
		
		`  Sub MakePlayerTorpedoMove(ByRef Board(,) As Char, ByVal Ships() As TShip)`
		`  Dim Row As Integer`
		`  Dim Column As Integer`
		`  GetRowColumn(Row, Column)`
		`  While Row > 0 And (Board(Row, Column) = "m" Or Board(Row, Column) = "-")`
		`  Board(Row, Column) = "m"`
		`  Row = Row - 1`
		`  End While`
		`  If Board(Row, Column) <> "-" And Board(Row, Column) <> "m" Then`
		`  Console.WriteLine("Torpedo hits at (" & Row & "," & Column & ")")`
		`  Board(Row, Column) = "h"`
		`  Else`
		`  Console.WriteLine("Torpedo failed to hit a ship")`
		`  Board(Row, Column) = "m"`
		`  End If`
		`  End Sub`
			12

PASCAL

05 **1** 8

```
Var
  Value : Integer;
  Operation : Char;
  Count : Integer;
Begin
  Write('Enter integer (0-99): ');
  Readln(Value);
  Write('Calculate additive or multiplicative persistence (a or
m)?');
  Readln(Operation);
  Count := 0;
  While Value > 9 Do
  Begin
    If Operation = 'a' Then
      Value := (Value DIV 10) + (Value MOD 10)
    Else
      Value := (Value DIV 10) * (Value MOD 10);
    Count := Count + 1;
  End;
  Write('The persistence is: ')
  Write(Count);
  Readln();
End.
```

09 **1** 5

```
Procedure GetRowColumn(Var Row : Integer; Var Column :
Integer);
Var
  Valid : Boolean;
Begin
  Valid := False;
  While Not(Valid) Do
  Begin
    Valid := True;
    Writeln;
    Write('Please enter column: ');
    Readln(Column);
    Write('Please enter row: ');
    Readln(Row);
    If (Row < 0) Or (Row > 9) Then
    Begin
      Writeln('Invalid value entered');
      Valid := False;
    End;
  End;
  Writeln;
End;
```

10 **1** 8

```
Procedure CheckSunk(Row : Integer; Column : Integer; Board :
TBoard;Var Ships : TShips);
Var
  ShipType : String;
  i : integer;
Begin
  ShipType := Board[Row][Column][1];
  For i := 1 To length(Ships) Do
  Begin
    If Ships[i].Name[1] = ShipType Then
    Begin
      Ships[i].Size := Ships[i].Size - 1;
      If Ships[i].Size = 0 Then
        Writeln(Ships[i].Name, ' is sunk!');
    End;
  End;
End;
```

Alternative (use of if / case / select for each ship type):

```
Procedure CheckSunk(Row : Integer; Column : Integer; Board :
TBoard;Var Ships : TShips);
Var
  ShipIndex : Integer;
Begin
  If Board[Row][Column] = 'A' Then ShipIndex := 0;
  If Board[Row][Column] = 'B' Then ShipIndex := 1;
  If Board[Row][Column] = 'S' Then ShipIndex := 2;
  If Board[Row][Column] = 'D' Then ShipIndex := 3;
  If Board[Row][Column] = 'P' Then ShipIndex := 4;
  Ships[ShipIndex].Size := Ships[ShipIndex].Size - 1;
  If Ships[ShipIndex].Size = 0 Then
    Writeln(Ships[ShipIndex].Name, ' is sunk!');
End;
```

10 **2** 1

```
…
   Writeln('Hit at (', Column, ',', Row, ').');
   CheckSunk(Row, Column, Board, Ships);
   Board[Row][Column] := 'h';
…
```

11 **1** 12

```
Procedure PlayGame(Board : TBoard; Ships : TShips);
Var
  GameWon : Boolean;
  TorpedoUsed : Boolean;
  TorpedoChosen : String;
Begin
  GameWon := False;
  TorpedoUsed := False;
```

```
  While Not(GameWon) Do
  Begin
    PrintBoard(Board);
    If Not(TorpedoUsed) Then
    Begin
      Writeln('Fire a torpedo? (Y/N)');
      Readln(TorpedoChosen);
    End;
    If TorpedoChosen = 'Y' Then
    Begin
      MakePlayerTorpedoMove(Board, Ships);
      TorpedoChosen := 'N';
      TorpedoUsed := True;
    End
    Else
      MakePlayerMove(Board, Ships);
    GameWon := CheckWin(Board);
    If GameWon = True Then
    Begin
      Writeln('All ships sunk!');
      Writeln;
    End;
  End;
End;

Procedure MakePlayerTorpedoMove(Var Board : TBoard; Var
Ships : TShips);
Var
  Row : Integer;
  Column : Integer;
Begin
  GetRowColumn(Row, Column);
  While (Row > 0) And ((Board[Row][Column] = 'm') Or
(Board[Row][Column] = '-')) Do
  Begin
    Board[Row][Column] := 'm';
    Row := Row - 1;
  End;
  If (Board[Row][Column] <> '-') And (Board[Row][Column] <>
'm') Then
  Begin
    Writeln('Torpedo hits at (', Column, ',', Row, ').');
    Board[Row][Column] := 'h';
  End
  Else
  Begin
    Writeln('Torpedo failed to hit a ship.');
    Board[Row][Column] := 'm';
  End;
End;
```

C#

05	**1**	`int Value, Count;`
		`string Operation;`
		`Console.WriteLine("Enter integer (0-99):");`
		`Value = Convert.ToInt32(Console.ReadLine());`
		`Console.WriteLine("Calculate the additive or multiplicative perisistence (a or m)?");`
		`Operation = Console.ReadLine();`
		`Count = 0;`
		`while (Value > 9)`
		`{`
		`  if (Operation == "a")`
		`  {`
		`  Value = (Value / 10) + (Value % 10);`
		`  }`
		`  else`
		`  {`
		`  Value = (Value / 10) * (Value % 10);`
		`  }`
		`  Count = Count + 1;`
		`}`
		`Console.Write("The persistence is: ");`
		`Console.WriteLine(Count);`
			8

09	**1**	`Console.WriteLine();`
		`Console.Write("Please enter column: ");`
		`Column = Convert.ToInt32(Console.ReadLine());`
		`do`
		`{`
		`  Console.Write("Please enter row: ");`
		`  Row = Convert.ToInt32(Console.ReadLine());`
		`  if (Row < 0 || Row > 9)`
		`  {`
		`  Console.WriteLine("Invalid value entered");`
		`  }`
		`} while (Row < 0 || Row > 9);`
		`Console.WriteLine();`
			5

10	**1**	`private static void CheckSunk(int Row, int Column, char[,] Board, ref ShipType[] Ships)`
		`{`
		`  for (int i = 0; i < 5; i++)`
		`  {`
		`  if (Ships[i].ShipName[0] == Board[Row, Column])`
		`  {`
		`  Ships[i].ShipSize = Ships[i].ShipSize - 1;`
		`  if (Ships[i].ShipSize == 0)`
		`  {`
		`  Console.WriteLine(Ships[i].ShipName + " is sunk!");`
		`  }`
		`  }`
		`  }`
		`}`
		
		Alternative (use of if / case / select for each ship type):
		
		`private static void CheckSunk(int Row, int Column, char[,] Board, ref ShipType[] Ships)`
		`{`
		`  int ShipIndex;`
		`  if (Board[Row,Column] == 'A')`
		`  {`
		`  ShipIndex=0;`
		`  }`
		`  else if (Board[Row, Column] == 'B')`
		`  {`
		`  ShipIndex=1;`
		`  }`
		`  else if (Board[Row, Column] == 'S')`
		`  {`
		`  ShipIndex=2;`
		`  }`
		`  else if (Board[Row, Column] == 'D')`
		`  {`
		`  ShipIndex=3;`
		`  }`
		`  else if (Board[Row, Column] == 'P')`
		`  {`
		`  ShipIndex=4;`
		`  }`
		`  Ships[ShipIndex].Size -= 1;`
		`  if (Ships[ShipIndex].Size == 0)`
		`  {`
		`  Console.WriteLine(Ships[ShipIndex].ShipName + " is sunk!");`
		`  }`
		`}`
			8

10	**2**	...
		else
		{
		Console.WriteLine("Hit at (" + Column + "," + Row + ").");
		CheckSunk(Row, Column, Board, ref Ships);
		Board[Row, Column] = 'h';
		...
			1

11	**1**	private static void PlayGame(ref char[,] Board, ref ShipType[] Ships)
		{
		bool GameWon = false;
		bool TorpedoUsed = false;
		string TorpedoChosen;
		while (GameWon == false)
		{
		PrintBoard(Board);
		if (TorpedoUsed == false)
		{
		Console.WriteLine("Fire a torpedo? (Y/N)");
		TorpedoChosen = Console.ReadLine();
		}
		if (TorpedoChosen == "Y")
		{
		MakePlayerTorpedoMove(ref Board, ref Ships);
		TorpedoChosen = "N";
		TorpedoUsed = true;
		}
		else
		{
		MakePlayerMove(ref Board, ref Ships);
		}
		GameWon = CheckWin(Board);
		if (GameWon == true)
		{
		Console.WriteLine("All ships sunk!");
		Console.WriteLine();
			12

```
        }
      }
    }

  private static void MakePlayerTorpedoMove(ref char[,]
Board, ref ShipType[] Ships)
  {
    int Row = 0;
    int Column = 0;
    GetRowColumn(ref Row, ref Column);
    while (Row > 0 && (Board[Row, Column] == 'm' ||
Board[Row, Column] == '-'))
    {
      Board[Row, Column] = 'm';
      Row = Row - 1;
    }
    if (Board[Row, Column] <> '-' && Board[Row, Column] <>
'm')
    {
      Console.WriteLine("Torpedo hits at (" + Column + "," +
Row + ").");
      Board[Row, Column] = 'h';
    }
    else
    {
      Console.WriteLine("Torpedo failed to hit a ship.");
      Board[Row, Column] = 'm';
    }
}
```

JAVA

05 **1**

```
AQAConsole2016 console = new AQAConsole2016();

console.println("Enter integer (0-99): ");
int value = console.readInteger("");
console.println("Calculate additive or multiplicative
persistence (a or m)? ");
char operation =  console.readChar("");

int count = 0;

while (value > 9){
     if(operation == 'a'){
          value = (value/10) + (value%10);
     }
     else{
          value = (value/10) * (value%10);
     }
     count += 1;
}

console.println("The persistence is: " + count);
```

8

A. Putting prompts inside `readInteger` and `readChar` rather than separate `println` statement
A. Putting `count = count + 1` instead of `count += 1`.

A. Answers that don't make use of the AQA classes

NOTE: if a string is used for `operation` then the comparison would be `operation.equals("a")`

09 **1**

```
int[] getRowColumn(){
  int column;
  int row;
  int[] move;
  move = new int[2];
  column = console.readInteger("Please enter column: ");
  boolean valid = false;
  while(!valid){
    row = console.readInteger("Please enter row: ");
    if (row < 0 || row > 9){
      console.println("Invalid input entered");
    }
    else{
      valid = true;
    }
  }
  console.println();
  move[0] = row;
  move[1] = column;
  return move;
}
```

5

10 1

```
void checkSunk(int row, int column, char[][] board, Ship[]
ships){
 char shipType = board[row][column];
 for(int i = 0; i < ships.length; i++){
    if(ships[i].name.charAt(0) == shipType){
      ships[i].length -= 1;
      if(ships[i].length == 0){
       console.println(ships[i].name +  " is sunk!");
      }
    }
  }
}
```

Alternative (use of if / case / select for each ship type):

```
void checkSunk(int row, int column, char[][] board, Ship[]
ships){
  int shipIndex;
  if(board[row][column] == 'A'){
    shipIndex = 0;
  }
  if(board[row][column] == 'B'){
    shipIndex = 1;
  }
  if(board[row][column] == 'S'){
    shipIndex = 2;
  }
  if(board[row][column] == 'D'){
    shipIndex = 3;
  }
  if(board[row][column] == 'P'){
    shipIndex = 4;
  }
  ships[shipIndex].length -= 1;
  if(ships[shipIndex].length == 0){
    console.println(ships[shipIndex].name + " is sunk!");
  }
}
```

8

10 2

```
...
else{
  console.println("Hit at (" + column + "," + row + ").");
  checkSunk(row, column, board, ships);
  board[row][column] = 'h';
...
```

1

11	**1**		12

```
  void playGame(char[][] board, Ship[] ships){
    boolean gameWon = false;
    boolean torpedoUsed = false;
    char torpedoChosen;
    while(!gameWon){
      printBoard(board);
      if (!torpedoUsed){
        console.println("Fire a torpedo? (Y/N)");
        torpedoChosen = console.readChar("");
      }
      if (torpedoChosen == 'Y'){
        MakePlayerTorpedoMove(board, ships);
        torpedoChosen = 'N';
        torpedoUsed = true;
      }
      else{
        makePlayerMove(board, ships);
      }
      gameWon = checkWin(board, ships);
      if(gameWon){
        console.println("All ships sunk!");
        console.println();
      }
    }
  }

  void makePlayerTorpedoMove(char[][] board, Ship[] ships){
    int[] rowColumn = getRowColumn();
    int row = rowColumn[0];
    int column = rowColumn[1];
    while (row > 0 && (board[row][column] == 'm' ||
board[row][column] == '-'){
      board[row][column] = 'm';
      row = row - 1;
    }
    if (board[row][column] != '-' && board[row][column] != 'm'){
      console.println("Torpedo hits at (" + column + "," + row +
").");
      board[row][column] = 'h';
    }
    else{
      console.println("Torpedo failed to hit a ship.");
      board[row][column] = 'm';
    }
}
```

PYTHON 2

05 **1**

```
print "Enter integer (0-99): "
Value = int(raw_input())
print "Calculate additive or multiplicative persistence (a
or m)? "
Operation = raw_input()
Count = 0
while Value > 9:
    if Operation == "a":
        Value = (Value / 10) + (Value % 10)
    else:
        Value = (Value / 10) * (Value % 10)
    Count = Count + 1
print "The persistence is: "
print Count
```

8

A. `Value = input()` rather than `Value = int(raw_input())`
A. Putting prompts inside `raw_input` rather than separate `print` statement

09 **1**

```
def GetRowColumn():
  Valid = False
  while not Valid:
    Valid = True
    print
    Column = int(raw_input("Please enter column: "))
    Row = int(raw_input("Please enter row: "))
    print
    if Row < 0 or Row > 9:
      print "Invalid value entered"
      Valid = False
  return Row, Column
```

5

Alternative:

```
def GetRowColumn():
  print
  Column = int(raw_input("Please enter column: "))
  Valid = False
  while not Valid:
    Valid = True
    Row = int(raw_input("Please enter row: "))
    print
    if Row < 0 or Row > 9:
      print "Invalid value entered"
      Valid = False
  return Row, Column
```

10 **1**

```
def CheckSunk(Row, Column, Board, Ships):
  ShipChar = Board[Row][Column]
  for Ship in range(len(Ships)):
    if Ships[Ship][0][0] == ShipChar:
      Ships[Ship][1] = Ships[Ship][1] - 1
      if Ships[Ship][1] == 0:
        print Ships[Ship][0] + " is sunk!"
```

Alternative (use of if / case / select for each ship type):

```
def CheckSunk(Row, Column, Board, Ships):
  if Board[Row, Column] == "A":
    ShipIndex = 0
  if Board[Row, Column] == "B":
    ShipIndex = 1
  if Board[Row, Column] == "S":
    ShipIndex = 2
  if Board[Row, Column] == "D":
    ShipIndex = 3
  if Board[Row, Column] == "P":
    ShipIndex = 4
  Ships[ShipIndex][1] = Ships[ShipIndex][1] - 1
  if Ships[ShipIndex][1] == 0:
    print Ships[ShipIndex][0] + " is sunk!"
```

8

10 **2**

```
...
else:
  print "Hit at (" + str(Column) + "," + str(Row) + ")."
  CheckSunk(Row, Column, Board, Ships)
  Board[Row][Column] = "h"
...
```

1

11 **1**

```
def PlayGame(Board, Ships):
  GameWon = False
  TorpedoUsed = False
  while not GameWon:
    PrintBoard(Board)
    if not TorpedoUsed:
      TorpedoChosen = raw_input("Fire a torpedo? (Y/N)")
    if TorpedoChosen == "Y":
      MakePlayerTorpedoMove(Board, Ships)
      TorpedoChosen = "N"
      TorpedoUsed = True
    else:
      MakePlayerMove(Board, Ships)
    GameWon = CheckWin(Board)
    if GameWon:
      print "All ships sunk!"
      print
```

12

```
def MakePlayerTorpedoMove(Board, Ships):
  Row, Column = GetRowColumn()
  while Row > 0 and (Board[Row][Column] == "m" or
Board[Row][Column] == "-"):
    Board[Row][Column] = "m"
    Row = Row - 1
  if Board[Row][Column] != "-" and Board[Row][Column] !=
"m":
    print "Torpedo hits at (" + str(Column) + "," + str(Row)
+ ")."
    Board[Row][Column] = "h"
  else:
    print "Torpedo failed to hit a ship."
```

PYTHON 3

05 **1**

```
print("Enter integer (0-99): ")
Value = int(input())
print("Calculate additive or multiplicative persistence (a
or m)? ")
Operation = input()
Count = 0
while Value > 9:
    if Operation == "a":
        Value = (Value // 10) + (Value % 10)
    else:
        Value = (Value // 10) * (Value % 10)
    Count = Count + 1
print("The persistence is: ")
print (Count)
```

A. Putting prompts inside `raw_input` rather than separate `print` statement

NOTE: Python 3 will require // to work for integer division

8

09 **1**

```
def GetRowColumn():
  Valid = False
  while not Valid:
    Valid = True
    print()
    Column = int(input("Please enter column: "))
    Row = int(input("Please enter row: "))
    print()
    if Row < 0 or Row > 9:
      print ("Invalid value entered")
      Valid = False
  return Row, Column
```

5

10 **1**

```
def CheckSunk(Row, Column, Board, Ships):
  ShipType = Board[Row][Column]
  for i in range(0, len(Ships)):
    if Ships[i][0][0] == ShipType:
      Ships[i][1] -= 1
      if Ships[i][1] == 0:
        print(Ships[i][0] + " is sunk!")
```

Alternative (use of if / case / select for each ship type):

```
def CheckSunk(Row, Column, Board, Ships):
  if Board[Row, Column] == "A":
    ShipIndex = 0
  if Board[Row, Column] == "B":
    ShipIndex = 1
  if Board[Row, Column] == "S":
    ShipIndex = 2
```

8

		`  if Board[Row, Column] == "D":`
		`    ShipIndex = 3`
		`  if Board[Row, Column] == "P":`
		`    ShipIndex = 4`
		`  Ships[ShipIndex][1] = Ships[ShipIndex][1] - 1`
		`  if Ships[ShipIndex][1] == 0:`
		`    print(Ships[ShipIndex][0] + " is sunk!")`	

10	**2**	`...`
		`else:`
		`   print("Hit at (" + str(Column) + "," + str(Row) + ").")`
		`   CheckSunk(Row, Column, Board, Ships)`
		`   Board[Row][Column] = "h"`
		`...`
			1

11	**1**	`def PlayGame(Board, Ships):`
		`  GameWon = False`
		`  TorpedoUsed = False`
		`  while not GameWon:`
		`    PrintBoard(Board)`
		`    if not TorpedoUsed:`
		`      TorpedoChosen = input("Fire a torpedo? (Y/N)")`
		`    if TorpedoChosen == "Y":`
		`      MakePlayerTorpedoMove(Board, Ships)`
		`      TorpedoChosen = "N"`
		`      TorpedoUsed = True`
		`    else:`
		`      MakePlayerMove(Board, Ships)`
		`    GameWon = CheckWin(Board)`
		`    if GameWon:`
		`      print("All ships sunk!")`
		`      print()`
		
		`def MakePlayerTorpedoMove(Board, Ships):`
		`  Row, Column = GetRowColumn()`
		`  while Row > 0 and (Board[Row][Column] == "m" or Board[Row][Column] == "-"):`
		`    Board[Row][Column] = "m"`
		`    Row = Row - 1`
		`  if Board[Row][Column] != "-" and Board[Row][Column] != "m":`
		`    print("Torpedo hits at (" + str(Column) + "," + str(Row) + ").")`
		`    Board[Row][Column] = "h"`
		`  else:`
		`    print("Torpedo failed to hit a ship.")`
			12