AQA

A-LEVEL
COMPUTER SCIENCE
7517/1

Paper 1

Mark scheme
June 2019

Version: 1.0 Final

I96A75I7I/MS

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

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.

A-level Computer Science

Paper 1 (7517/1) – applicable to all programming languages A, B, C, D and E

June 2019

The following annotation is used in the mark scheme:

; - means a single mark
// - means an alternative response
/ - means an alternative word or sub-phrase
A - means an 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.

Page 3 contains 'Level of Response marking instructions'.

Pages 6 to 19 contain the generic mark scheme.

Pages 20 to 52 contain the 'Program Source Code' specific to the programming languages for questions 05.1, 10.1, 11.1, 12.1 and 13.1

pages 20 to 24 – VB.NET
pages 25 to 28 – PYTHON 3
pages 29 to 32 – PYTHON 2
pages 33 to 37 – PASCAL
pages 38 to 44 – C#
pages 45 to 52 – JAVA

Examiners are required to assign each of the candidate's responses to the most appropriate level according to **its overall quality**, and 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 5.1, the marks available for the AO3 elements are as follows:

AO3 (design) – 4 marks
AO3 (programming) – 8 marks

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

Question			Marks

01	1	**All marks AO2 (analyse)**
		
		Have a flag variable that is set to True if a swap is made and reset to False at the start of each pass / the outer loop // Have a flag variable that is set to True at the start of each pass to indicate that the list is in order and set to False if a swap is made;
		change the outer loop so that it would stop repeating if no swaps have been made;
		
		After the inner loop; subtract 1 from N; // alter inner loop (for) upper limit;
		by subtracting `Count1` from N;
			4

01	2	**All marks AO1 (understanding)** Sorting a list is (always) a tractable problem // sorting a list is always polynomial time (or better); A problem does not change from being tractable to intractable / polynomial to exponential as the problem size grows (an intractable problem is one that is not solvable in a reasonable amount of time as the size of the problem grows);	**2**
01	3	**All marks AO1 (understanding)** Use of heuristic; An algorithm that makes a guess/estimate based on experience; **N.E.** algorithm that uses previous knowledge/experience That provide a close-to-optimal solution/approximation // that only works in some cases; **A.** non-optimal Relax some of the constraints on the solution; **A.** solve simpler version of problem **A.** Reduce the size of the search space **Max 2 marks**	**2**

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>02</td><td>1</td><td>
All marks AO2 (apply)
<table>
<tr><th colspan="9">Tape</th><th>Current state</th></tr>
<tr><td>...</td><td>0</td><td>#</td><td>1</td><td>0*</td><td>1</td><td>0</td><td>0</td><td>...</td><td>S3</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>1*</td><td>0</td><td>1</td><td>0</td><td>0</td><td>...</td><td>S4</td></tr>
<tr><td>...</td><td>0</td><td>#*</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>...</td><td>S4</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>1*</td><td>0</td><td>1</td><td>0</td><td>0</td><td>...</td><td>S0</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#</td><td>0*</td><td>1</td><td>0</td><td>0</td><td>...</td><td>S1</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#</td><td>0</td><td>1*</td><td>0</td><td>0</td><td>...</td><td>S2</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#</td><td>0</td><td>1</td><td>0*</td><td>0</td><td>...</td><td>S2</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#</td><td>0</td><td>1*</td><td>1</td><td>0</td><td>...</td><td>S3</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#</td><td>0*</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S3</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#*</td><td>0</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S4</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#</td><td>0*</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S0</td></tr>
<tr><td>...</td><td>0</td><td>#</td><td>#*</td><td>0</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S5</td></tr>
<tr><td>...</td><td>0</td><td>#*</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S5</td></tr>
<tr><td>...</td><td>0*</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S5</td></tr>
<tr><td>...</td><td>0</td><td>1*</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>...</td><td>S6</td></tr>
</table>
Mark as follows:

1 mark: first row of tape is correct
1 mark: current state and read/write head position correct for first row of tape
1 mark: second and third rows of tape and current state are correct
1 mark: last row of tape is correct
1 mark: all other rows of current state are correct and read/write head in correct position for row two onwards

A. alternative, unambiguous, ways of representing read/write head position
I. inclusion of shaded rows/columns
</td><td>5</td></tr>
<tr><td>02</td><td>2</td><td>Mark is for AO2 (analyse)

Make a copy of a string of 1s;

A. double the number of 1s on the tape</td><td>1</td></tr>
<tr><td>02</td><td>3</td><td>Mark is for AO2 (analyse)

Moves the read/write head to the start of the (original) string of 1s // moves the read/write head back to where it started from;</td><td>1</td></tr>
<tr><td>02</td><td>4</td><td>All marks AO1 (knowledge)

A Turing machine that can execute/simulate the behaviour of any other Turing machine // can compute any computable sequence;

Faithfully executes operations on the data precisely as the simulated TM does;
(Note: must have idea of same process)</td><td>2</td></tr>
</table>

		Description of/Instructions for TM (and the TM's input) are stored on the (Universal Turing machine's) tape // The UTM acts as an interpreter; **A.** take any other TM and data as input *Alternative definition:* A UTM, U, is an interpreter that reads the description <M> of any arbitrary Turing machine M; and faithfully executes operations on data D precisely as M does.; The description <M> is written at the beginning of the tape, followed by D.; **Max 2 marks**	

Question			**Marks**
03	**1**	**Mark is for AO2 (analyse)** I E H C A B G D F	**1**
03	**2**	**Mark is for AO1 (knowledge)** Removing (unnecessary) details;	**1**
03	**3**	**Mark is for AO1 (knowledge)** Grouping by common characteristics // a hierarchical / 'kind-of' relationship;	**1**
03	**4**	**Mark is for AO2 (analyse)** (If there is a relationship between two cells is still represented but) if the relationship is because two cells are in the same row/column/two-by-two block is no longer represented // the nature of the link between the two cells is not represented; **A.** the location of a cell is not represented	**1**
03	**5**	**All marks for AO1 (understanding)** Adjacency matrix appropriate when there are many edges between vertices // when graph/matrix is not sparse; when edges frequently changed; when presence/absence of specific edges needs to be tested frequently; **Max 2 marks** **A** Alternative words which describe edge, eg connection, line	**2**
03	**6**	**Mark is for AO1 (understanding)** Directed (graph) // digraph;	**1**

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>04</td><td>1</td><td>Mark for AO1 (knowledge)

Zero or more (of the preceding element/character/value);
A. any number of the preceding element/character/value</td><td>1</td></tr>
<tr><td>04</td><td>2</td><td>Mark for AO1 (knowledge)

Zero or one (of the preceding element/character/value) // (the preceding element/character/value is) optional;</td><td>1</td></tr>
<tr><td>04</td><td>3</td><td>All marks AO2 (apply)

<table>
<tr><th>String</th><th>Belongs to language (Y/N)?</th></tr>
<tr><td>1</td><td>Y</td></tr>
<tr><td>11</td><td>N</td></tr>
<tr><td>01</td><td>Y</td></tr>
<tr><td>0111</td><td>Y</td></tr>
<tr><td>0101</td><td>N</td></tr>
<tr><td>111</td><td>N</td></tr>
<tr><td>0011</td><td>N</td></tr>
</table>
Mark as follows:

1 mark: four rows correct
2 marks: five rows correct
3 marks: all seven rows correct</td><td>3</td></tr>
</table>

<table>
<tr><th colspan="2">Question</th><th colspan="2">Marks</th></tr>
<tr><td>05</td><td>1</td><td>

4 marks for AO3 (design) and 8 marks for AO3 (programming)

Mark Scheme

<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 meets most of the requirements. All of the appropriate design decisions have been taken. To award 12 marks, all of the requirements must be met.</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 program. The program displays relevant prompts, inputs the two words and includes one iterative structure and two selection structures. An attempt has been made to check that all the characters in the first word are in the second word, although this may not work correctly under all circumstances. The solution demonstrates good design work as most of the correct design decisions have been made.</td><td>7–9</td></tr>
<tr><td>2</td><td>A program has been written 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 program 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 program has been written and a few 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. It is unlikely that any of the key design elements of the task have been recognised.</td><td>1–3</td></tr>
</table>

Guidance

Evidence of AO3 design – 4 points:

Evidence of design to look for in responses:

1. Identifying that a selection structure is needed after all letter counts have been compared to output a message saying it can be made from the letters in the 2_{nd} word or that it can't
2. Identifying that a loop is needed that repeats a number of times based on the

</td><td>12</td></tr>
</table>

		length of the first word // identifying that a loop is needed that repeats 26 times // identifying that a loop is needed that repeats a number of times determined by the number of unique characters in the first word 3. Identifying that the number of times a letter occurs in the first string needs to be less than or equal to the number of times it occurs in the second string 4. Boolean (or equivalent) variable used to indicate if the first word can be formed from the letters in the second word // array of suitable size to store the count of each letter 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 for AO3 programming – 8 points:** Evidence of programming to look for in response: 5. (Suitable prompts asking user to enter the two words followed by) user inputs being assigned to appropriate variables (**R.** if inside or after iterative structure), two variables with appropriate data types created to store the two words entered by the user 6. Iterative structure to look at each letter in first word has correct syntax and start/end conditions // iterative structure to look at each letter in the alphabet has correct syntax and start/end conditions 7. Correctly counts the number of times that a letter occurs in one of the words 8. Selection structure that compares the count of a letter in the first word with the count of that letter in the second word **A.** incorrect counts **A.** incorrect comparison operator 9. Correctly counts the number of times each letter in one of the two words occurs 10. Program works correctly if the two words entered are the same 11. Program works correctly when first word contains more instances of a letter than there are in the second word (i.e. says that it cannot be formed from the second word) 12. Program works correctly for all word pairs consisting of just upper case letters **Alternative mark scheme** (based on removing an instance of a letter from the 2_{nd} word each time it appears in the 1_{st} word) 1. Identifying that a selection structure is needed after all the letters that appear in both words have been removed from the first word to output a message saying it can be made from the letters in the second word or that it can't 3. Identifying that a letter can be removed from the second word if it appears in the first word 7. Selection structure that checks if letter in first word appears in the second word 8. Removes a letter from the second word if it appears in the first word. 9. Sets indicator to false if a letter does not appear in the second word	
05	**2**	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from question 05.1, including prompts on screen capture matching*	**1**

		those in code. *Code for question 05.1 must be sensible.* Screen captures showing: • the string NINE being entered followed by the string ELEPHANTINE and then a message displayed saying that the first word can be formed from the second. • the string NINE being entered followed by the word ELEPHANT and then a message displayed saying that the first work cannot be formed from the second.	

Question			Marks
06		**Marks are for AO1 (understanding)** The binary file cannot be easily read by a person (so the game data is hidden more from the user); No need for string / data type conversion routines; File size likely to be smaller (as not all the stored data is text); **A.** Might make the program code easier to understand (as less need for string conversion routines); **N.E.** binary file cannot be read **Max 2 marks**	**2**

Question			Marks
07	1	**Mark is for AO2 (analyse)** The item is not in `Items`/the list/the array // The item does not exist;	**1**
07	2	**Mark is for AO2 (analyse)** It searches by name if the ID parameter has a value of -1 // it searches by name if `ItemIDToGet` is -1; It searches by ID if the ID parameter is not -1 // it searches by ID if `ItemIDToGet` is not -1 (**A.** >= 0 **R.** >0); **Max 1 mark**	**1**
07	3	**Mark is for AO2 (analyse)** Linear // n // O(n);	**1**
07	4	**Marks are for AO3 (evaluation)** When looking for an item by name and there are two/multiple items in `Items` with the same name; **A.** more than one item by implication and the item being sought is not the first item in `Items` with that name;	**2**

		Alternative answer When looking for an item by ID and there ar two/multiple iterms in Items with the same ID; and the item being sought is not the first item in `Items` with that ID; **Alternative answer** When looking for an item by name and there is an item with ID of -1 in `Items`; and the item with ID -1 is before the item being searched for in the list;	
07	**5**	**One mark for AO2 (analyse) and three marks for AO1 (understanding)** **Mark for AO2** Apply a hash function to the specified ID // apply a hash function to the value in `ItemIDToGet`; **Marks for AO1** This will give the position in the array where that item has been stored; If another item is in that position then use a method to check if a collision (occurred when adding items to hash table) **A.** description of specific method for checking if a collision had occurred when adding items to the table // if another item is in that position then use a method to check related locations; If the location is empty (and any positions used to deal with collisions are empty or do not contain the item) then the item does not exist;	**4**
07	**6**	**Mark is for AO2 (analyse)** The hash function can only be applied to the ID (not the name) // hash functions can only be applied to one piece of data; Would need two hash tables (one based on IDs and one based on names); Because searches need to be done based on two different properties of an item; **A.** there are not many items in the game (so the benefit of using hashing is minimal); **Max 1 mark**	**1**

Question			Marks
08	1	**All marks for AO2 (analyse)** The intersection of **B** with the union of **D** and **E** **//** The union of **E** with the intersection of **B** and **D** **Alternative answer** **B** ∩ (**D** ∪ **E**) // **E** ∪ (**D** ∩ **B**) **Mark as follows** 1 mark for using the sets **B**, **D** and **E** **R.** if answer also uses set **C** 1 mark for the union of set **E** with another set 1 mark for using the intersection operation with set **B** and another set **Max 2 marks** if any errors **A.** answers using alternative set notations **I.** intersection with set **A**	**3**
08	2	**Mark is for AO2 (analyse)** **A** and **B**;	**1**
08	3	**Mark is for AO2 (analyse)** Because there could also be items in a container object (that is in the current location); **A.** explanation that uses an example eg if player is in the cellar the black die is getable even though it is in a container (the shelf) not the cellar.	**1**
08	4	**Mark is for AO1 (understanding)** A set is a subset of itself but not a proper subset of itself // There will be at least one value in a set that is not in a proper subset of that set (that does not have to be case for a subset);	**1**

Question			Marks
09	1	**Mark is for AO2 (analyse)** `Main`; **I.** case **I.** spacing **R.** if any additional code **R.** if spelt incorrectly	**1**
09	2	**Mark is for AO2 (analyse)** `PlayGame`; **I.** case **I.** spacing **R.** if any additional code **R.** if spelt incorrectly	**1**
09	3	**Mark is for AO2 (analyse)** `LoadGame`; `IsNumeric` (Java only); **Max 1** **I.** case **I.** spacing **R.** if any additional code **R.** if spelt incorrectly	**1**
09	4	**Mark is for AO1 (understanding)** Local variables have more limited scope; Global variables exist throughout the entire program; Local variables only exist in a part/block/subroutine of the program; Local variables can only be accessed in a part/block/subroutine of the program; Global variables can be accessed from any part of the program; **Max 1 mark**	**1**
09	5	**Mark is for AO1 (knowledge)** Modularisation of a program; Allows reuse of subroutines; Less chance of side-effects; **A.** advantages resulting from modularisation eg easier to test each subroutine independently **Max 1 mark**	**1**

Question			Marks
10	1	**All marks for AO3 (programming)** 1. Creates a random number; 2. Selection structure with random number used in condition; 3. Selection structure with one message in then part and one message in else part – one of the messages must be the original message "`Sorry, you don't know how to ***.`" and one must be the new message "`Sorry, I don't know what *** means.`"; **R.** other messages **R.** if spacing incorrect **I.** case **I.** punctuation **A.** answers that use two selection structures as long as they are equivalent to using an if…then...else structure 4. Each message has probability of being displayed 50% of the time; **A.** any suitable message **A.** answers with value between 0 and just less than 1is generated where 0.5 is rounded incorrectly **Max 3** if code contains errors **Max 2** if both error messages could be displayed sometimes	**4**
10	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from question 10.1, including prompts on screen capture matching those in code.* *Code for question 10.1 must be sensible.* Screen captures showing the command `eat` being entered (**I.** any text after the `eat` command) followed by one of the two messages – this should be done at least twice and there must be evidence that both messages can be displayed;	**1**

Question			Marks
11	1	**All marks for AO3 (programming)** 1. Iterative structure to loop through each item in `Items`; 2. Selection structure inside iterative structure with valid syntax and correct condition for selection structure that compares player's ID (1001) / `Inventory` with location of an item; 3. One added to appropriately-named variable used to count number of objects in inventory; **R.** if not inside selection structure inside iterative structure 4. Selection structure, after attempt at iterative structure, that compares count of items in inventory (**A.** incorrect count) with the number 5 (**A.** alternative logic e.g. > 4); **R.** if incorrect logic 5. Message inside attempt at selection structure from mark point 4 saying that player can't carry any more; **A.** selection structure in wrong place in code 6. If the number of items in the inventory is fewer than five then code added does not prevent item from being added to inventory; **Note for examiners:** this mark can only be awarded if mark points 1 and 4 have been awarded 7. If the number of items in the inventory is five (or more) then the item is not added to the inventory, the item stays in its current location and the result of getting the item is not executed; **A.** other values to five for number of items in inventory based on incorrect answer for mark point 4 **Max 6 marks** if code contains errors	**7**
11	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from question 11.1, including prompts on screen capture matching those in code.* *Code for question 11.1 must be sensible.* Screen capture(s) showing that the red die and torch are picked up by the player but not the book;	**1**

Question			Marks
12	1	**All marks for AO3 (programming)** 1. Creating a new subroutine called `DropItem`; **R**. other names for subroutine **I.** case 2. Adding new option to the selection structure in `PlayGame` for the `drop` command; 3. Call to `DropItem` inside the option added for mark point 2; **R.** if name does not match name of created subroutine **R.** if parameter list for subroutine call does not match parameter list for new subroutine 4. Parameter list for the new subroutine and contains `Items`, the item to drop and the current location of the player; **I.** additional parameters that are not needed **A.** alternatives to these parameters as long as evidence of attempt to get them to be usable is in code eg passing `Characters` instead of just the location of the player as long as some code to extract the location is included in the new subroutine The following all relate to the `DropItem` subroutine: 5. Gets the index of the item to drop; 6. Selection structure that checks if the item to drop does exist and results in appropriate error message being displayed if it doesn't; 7. Selection structure that checks if the item to drop is in the player's inventory and results in appropriate error message being displayed if it isn't; 8. Selection structure that checks if item to drop is fragile; 9. If item is in player's inventory and is fragile an appropriate message is displayed; **A.** incorrect conditions for mark points 7 and/or 8 10. If item is in player's inventory and is fragile then item is removed from `Items` // if item is in player's inventory and is fragile then the location of the item is changed to a location that does not exist; **A.** incorrect conditions for mark points 7 and/or 8 11. Location of item to drop is changed to the current location if it is in the player's inventory and is not fragile and an appropriate message is displayed; **A.** incorrect conditions for mark points 7 and/or 8 **A.** no attempts for mark points 7 and/or 8 12. Logic for mark points 6 –11 is correct, program won't display any incorrect messages and does not try to access position -1 in `Items` if item does not exist; **Max 11** if code contains errors	**12**
12	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from question 12.1, including prompts on screen capture matching those in code.* *Code for question 12.1 must be sensible.* Screen capture(s) showing that the player's inventory contains just the flask and that the contents of the room are the apple, torch and red die;	**1**

Question			Marks
13	1	**All marks for AO3 (programming)** 1. Code modified to roll the player's die three times; 2. Appropriate data structure(s) / variables to store the results of the player's dice rolls; 3. Code identifies the highest/smallest of the three numbers rolled by the player; 4. Code multiplies one of the results of the player's dice rolls by 100, another by 10 and adds the results of these two multiplications to the result of the other die roll; 5. Correct calculation of the player's score; 6. Code modified to roll the other character's die three times; 7. Correct calculation of the other character's score; 8. All expected messages, including messages showing the result of each die roll, displayed under the expected circumstances **Max 7** if code contains errors or if other parts of the subroutine no longer work correctly	**8**
13	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from question 13.1, including prompts on screen capture matching those in code.* *Code for question 13.1 must be sensible.* Screen capture(s) showing two tests with correct scores calculated for both player and other character and correct result displayed; **A.** missing results of individual die rolls not displayed	**1**

VB.Net

05	1	Dim Word1, Word2 As String Dim CanBeMadeFromSecondWord As Boolean = True Dim Pos As Integer Console.Write("Enter the first word: ") Word1 = Console.ReadLine Console.Write("Enter the second word: ") Word2 = Console.ReadLine For Pos = 0 To Word1.Length - 1 If Word1.Split(Word1(Pos)).Length - 1 > Word2.Split(Word1(Pos)).Length - 1 Then CanBeMadeFromSecondWord = False End If Next If CanBeMadeFromSecondWord Then Console.WriteLine("Yes") Else Console.WriteLine("No") End If Console.ReadLine() **Alternative answer** Dim Word1, Word2 As String Dim CanBeMadeFromSecondWord As Boolean = True Dim Pos As Integer Dim Loc As Integer Console.Write("Enter the first word: ") Word1 = Console.ReadLine Console.Write("Enter the second word: ") Word2 = Console.ReadLine For Pos = 0 To Word1.Length - 1 If Word2.Contains(Word1(Pos)) Then Loc = Word2.IndexOf(Word1(Pos)) Word2 = Word2.Remove(Loc, 1) Else CanBeMadeFromSecondWord = False End If Next If CanBeMadeFromSecondWord Then Console.WriteLine("Yes") Else Console.WriteLine("No") End If Console.ReadLine() **Alternative answer** Dim Word1, Word2 As String Dim CanBeMadeFromSecondWord As Boolean = True Dim Pos As Integer Dim Counts(25, 1) As Integer Console.Write("Enter the first word: ") Word1 = Console.ReadLine Console.Write("Enter the second word: ") Word2 = Console.ReadLine For Each ch In Word1	12

```
  Counts(Asc(ch) - 65, 0) += 1
Next
For Each ch In Word2
  Counts(Asc(ch) - 65, 1) += 1
Next
Pos = 0
While Pos <= 25
  If Counts(Pos, 0) > Counts(Pos, 1) Then
    CanBeMadeFromSecondWord = False
  End If
  Pos += 1
End While
If CanBeMadeFromSecondWord Then
  Console.WriteLine("Yes")
Else
  Console.WriteLine("No")
End If
Console.ReadLine()
```

10 | 1 | 4

```
Sub PlayGame(ByVal Characters As ArrayList, ByVal Items As
ArrayList, ByVal Places As ArrayList)
  ...
    Case "quit"
      Say("You decide to give up, try again another time.")
      StopGame = True
    Case Else
      Dim Rno As Integer = Int(Rnd() * 2)
      If Rno = 1 Then
        Console.WriteLine("Sorry, you don't know how to " &
Command & ".")
      Else
        Console.WriteLine("Sorry, I don't know what " & Command
& " means.")
      End If
    End Select
  End While
  Console.ReadLine()
End Sub
```

Alternative answer

```
Case Else
  If GetRandomNumber(0, 1) = 0 Then
    Console.WriteLine("Sorry, you don't know how to " & Command
& ".")
  Else
    Console.WriteLine("Sorry, I don't know what " & Command & "
means.")
  End If
End Select
```

11	1	Sub GetItem(ByVal Items As ArrayList, ByVal ItemToGet As String, ByVal CurrentLocation As Integer, ByRef StopGame As Boolean, ByVal Places As ArrayList) Dim ResultForCommand As String Dim SubCommand As String = "" Dim SubCommandParameter As String = "" Dim IndexOfItem, Position As Integer Dim CanGet As Boolean = False IndexOfItem = GetIndexOfItem(ItemToGet, -1, Items) If IndexOfItem = -1 Then Console.WriteLine("You can't find " & ItemToGet & ".") ElseIf Items(IndexOfItem).Location = Inventory Then Console.WriteLine("You have already got that!") ElseIf Not Items(IndexOfItem).Commands.contains("get") Then Console.WriteLine("You can't get " & ItemToGet & ".") ElseIf Items(IndexOfItem).Location >= MinimumIDForItem AndAlso Items(GetIndexOfItem("", Items(IndexOfItem).Location, Items)).Location <> CurrentLocation Then Console.WriteLine("You can't find " & ItemToGet & ".") ElseIf Items(IndexOfItem).Location < MinimumIDForItem And Items(IndexOfItem).Location <> CurrentLocation Then Console.WriteLine("You can't find " & ItemToGet & ".") Else CanGet = True End If If CanGet Then **Dim NoOfItems As Integer = 0** **For Each Thing In Items** **If Thing.Location = Inventory Then** **NoOfItems += 1** **End If** **Next** **If NoOfItems >= 5 Then** **Console.WriteLine("You can't carry anything else.")** **Else** Position = GetPositionOfCommand(Items(IndexOfItem).Commands, "get") ResultForCommand = GetResultForCommand(Items(IndexOfItem).Results, Position) ExtractResultForCommand(SubCommand, SubCommandParameter, ResultForCommand) If SubCommand = "say" Then Say(SubCommandParameter) ElseIf SubCommand = "win" Then Say("You have won the game") StopGame = True Exit Sub End If If Items(IndexOfItem).Status.contains("gettable") Then ChangeLocationOfItem(Items, IndexOfItem, 1001) Console.WriteLine("You have got that now.") End If **End If** End If End Sub **Alternative answer** Dim IndexOfItem, Position As Integer Dim CanGet As Boolean = False	7

		Dim ItemsCount As Integer = 0 **For Each Thing In Items** ** If Thing.Location = Inventory Then** ** ItemsCount += 1** ** End If** **Next** IndexOfItem = GetIndexOfItem(ItemToGet, -1, Items) **If ItemsCount >= 5 Then** ** Console.WriteLine("You already have five items, you cannot carry any more")** **Else**If IndexOfItem = -1 Then Console.WriteLine("You can't find " & ItemToGet & ".") CanGet = False	
12	1	Sub PlayGame(ByVal Characters As ArrayList, ByVal Items As ArrayList, ByVal Places As ArrayList) ... Instruction = GetInstruction() Command = ExtractCommand(Instruction) Select Case Command **Case "drop"** **DropItem(Items, Instruction, Characters(0).CurrentLocation)** Case "get" GetItem(Items, Instruction, Characters(0).CurrentLocation, StopGame, Places) ... Case Else Console.WriteLine("Sorry, you don't know how to " & Command & ".") End Select End While Console.ReadLine() End Sub **Sub DropItem(ByVal Items As ArrayList, ByVal ItemToDrop As String, ByVal Location As Integer)** ** Dim IndexOfItem As Integer** ** Dim CanDrop As Boolean = True** ** IndexOfItem = GetIndexOfItem(ItemToDrop, -1, Items)** ** If IndexOfItem = -1 OrElse Items(IndexOfItem).Location <> 1001 Then** ** Console.WriteLine("You don't have that!")** ** CanDrop = False** ** End If** ** If CanDrop Then** ** If Items(IndexOfItem).status.contains("fragile") Then** ** Items.RemoveAt(IndexOfItem)** ** Console.WriteLine("It broke!")** ** Else** ** ChangeLocationOfItem(Items, IndexOfItem, Location)** ** Console.WriteLine("You have dropped it.")** ** End If** ** End If** **End Sub**	12
13	1	Sub PlayDiceGame(ByVal Characters As ArrayList, ByVal Items As ArrayList, ByVal OtherCharacterName As String) ...	8

```
 If Not DiceGamePossible Then
    Console.WriteLine("You can't play a dice game.")
  Else
    Position =
GetPositionOfCommand(Items(IndexOfPlayerDie).Commands, "use")
    ResultForCommand =
GetResultForCommand(Items(IndexOfPlayerDie).Results, Position)
    Dim Results(2) As Integer
    For count = 0 To 2
      Results(count) = RollDie(ResultForCommand(5),
ResultForCommand(7))
      Console.WriteLine("You rolled a " & CStr(Results(count)) &
".")
    Next
    Dim largest As Integer = Results(0)
    Dim smallest As Integer = Results(0)
    Dim middle As Integer = Results(0)
    For count = 1 To 2
      If Results(count) > largest Then
        middle = largest
        largest = Results(count)
      ElseIf Results(count) < smallest Then
        middle = smallest
        smallest = Results(count)
      Else
        middle = Results(count)
      End If
    Next
    PlayerScore = largest * 100 + middle * 10 + smallest
    Position =
GetPositionOfCommand(Items(IndexOfOtherCharacterDie).Commands,
"use")
    ResultForCommand =
GetResultForCommand(Items(IndexOfOtherCharacterDie).Results,
Position)
    For count = 0 To 2
      Results(count) = RollDie(ResultForCommand(5),
ResultForCommand(7))
      Console.WriteLine("They rolled a " & CStr(Results(count))
& ".")
      OtherCharacterScore += Results(count) * 10 ^ count
    Next
    Console.WriteLine("Your score: " & CStr(PlayerScore))
    Console.WriteLine("Their score: " &
CStr(OtherCharacterScore))
    If PlayerScore > OtherCharacterScore Then
      Console.WriteLine("You win!")
      TakeItemFromOtherCharacter(Items,
Characters(IndexOfOtherCharacter).ID)
    ElseIf PlayerScore < OtherCharacterScore Then
      Console.WriteLine("You lose!")
      TakeRandomItemFromPlayer(Items,
Characters(IndexOfOtherCharacter).ID)
    Else
      Console.WriteLine("Draw!")
    End If
  End If
End Sub
```

Python 3

05 1

```
Word1 = ""
Word2 = ""
CanBeMadeFromSecondWord = True
Word1 = input("Enter the first word: ")
Word2 = input("Enter the second word: ")
for Pos in range(0, len(Word1)):
  if Word1.count(Word1[Pos]) > Word2.count(Word1[Pos]):
    CanBeMadeFromSecondWord = False
if CanBeMadeFromSecondWord:
  print("Yes")
else:
  print("No")
```

Alternative answer

```
Word1 = ""
Word2 = ""
CanBeMadeFromSecondWord = True
Word1 = input("Enter the first word: ")
Word2 = input("Enter the second word: ")
for Pos in range(0, len(Word1)):
  if Word1[Pos] in Word2:
    Word2 = Word2.replace(Word1[Pos], "", 1)
  else:
    CanBeMadeFromSecondWord = False
if CanBeMadeFromSecondWord:
  print("Yes")
else:
  print("No")
```

Alternative answer

```
Word1 = ""
Word2 = ""
CanBeMadeFromSecondWord = True
Counts = [[0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0],
[0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0],
[0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0],
[0,0]]
Word1 = input("Enter the first word: ")
Word2 = input("Enter the second word: ")
for ch in Word1:
  Counts[ord(ch) - 65][0] += 1
for ch in Word2:
  Counts[ord(ch) - 65][1] += 1
Pos = 0
while Pos <= 25:
  if Counts[Pos][0] > Counts[Pos][1]:
    CanBeMadeFromSecondWord = False
  Pos += 1
if CanBeMadeFromSecondWord:
  print("Yes")
```

12

		else: print("No")	

10	1	def PlayGame(Characters, Items, Places):
		  ...
		  elif Command == "quit":
		  Say("You decide to give up, try again another time.")
		  StopGame = True
		  else:
		  RNo = random.randint(0, 1)
		  if RNo == 0:
		  print("Sorry, you don't know how to " + Command + ".")
		  else:
		  print("Sorry, I don't know what " + Command + " means.")
		  input()
		
		Alternative answer
		
		  elif Command == "quit":
		  Say("You decide to give up, try again another time.")
		  StopGame = True
		  else:
		  if GetRandomNumber(0, 1) == 0:
		  print("Sorry, you don't know how to " + Command + ".")
		  else:
		  print("Sorry, I don't know what " + Command + " means.")
		  input()
			4

11	1	def GetItem(Items, ItemToGet, CurrentLocation):
		  SubCommand = ""
		  SubCommandParameter = ""
		  CanGet = False
		  IndexOfItem = GetIndexOfItem(ItemToGet, -1, Items)
		  if IndexOfItem == -1:
		  print("You can't find " + ItemToGet + ".")
		  elif Items[IndexOfItem].Location == INVENTORY:
		  print("You have already got that!")
		  elif not "get" in Items[IndexOfItem].Commands:
		  print("You can't get " + ItemToGet + ".")
		  elif Items[IndexOfItem].Location >= MINIMUM_ID_FOR_ITEM and Items[GetIndexOfItem("", Items[IndexOfItem].Location, Items)].Location != CurrentLocation:
		  print("You can't find " + ItemToGet + ".")
		  elif Items[IndexOfItem].Location < MINIMUM_ID_FOR_ITEM and Items[IndexOfItem].Location != CurrentLocation:
		  print("You can't find " + ItemToGet + ".")
		  else:
		  CanGet = True
		  if CanGet:
		  NoOfItems = 0
		  for Thing in Items:
		  if Thing.Location == INVENTORY:
		  NoOfItems += 1
		  if NoOfItems >= 5:
		  print("You can't carry anything else.")
		  else:
		  Position = GetPositionOfCommand(Items[IndexOfItem].Commands, "get")
		  ResultForCommand =
			7

```
GetResultForCommand(Items[IndexOfItem].Results, Position)
      SubCommand, SubCommandParameter =
ExtractResultForCommand(SubCommand, SubCommandParameter,
ResultForCommand)
      if SubCommand == "say":
        Say(SubCommandParameter)
      elif SubCommand == "win":
        Say("You have won the game")
        return True, Items
      if "gettable" in Items[IndexOfItem].Status:
        Items = ChangeLocationOfItem(Items, IndexOfItem,
INVENTORY)
        print("You have got that now.")
  return False, Items
```

Alternative answer

```
  SubCommand = ""
  SubCommandParameter = ""
  CanGet = False
  ItemsCount = 0
  for Thing in Items:
    if Thing.Location == INVENTORY:
      ItemsCount += 1
  IndexOfItem = GetIndexOfItem(ItemToGet, -1, Items)
  if ItemsCount >= 5:
    print("You already have five items, you cannot carry any
more")
  elif IndexOfItem == -1:
    print("You can't find " + ItemToGet + ".")
```

12 1

```
def PlayGame(Characters, Items, Places):
...
    Instruction = GetInstruction()
    Command, Instruction = ExtractCommand(Instruction)
    if Command == "get":
      StopGame, Items = GetItem(Items, Instruction,
Characters[0].CurrentLocation)
    elif Command == "drop":
      Items = DropItem(Items, Instruction,
Characters[0].CurrentLocation)
    elif Command == "use":
      StopGame, Items = UseItem(Items, Instruction,
Characters[0].CurrentLocation, Places)



def DropItem(Items, ItemToDrop, Location):
  CanDrop = True
  IndexOfItem = GetIndexOfItem(ItemToDrop, -1, Items)
  if not(IndexOfItem == -1 or Items[IndexOfItem].Location ==
1001):
    print("You don't have that!")
    CanDrop = False
  if CanDrop:
    if "fragile" in Items[IndexOfItem].Status:
      del(Items[IndexOfItem])
      print("It broke!")
```

12

		else: **ChangeLocationOfItem(Items, IndexOfItem, Location)** **print("You have dropped it.")** **return Items**	

13	1	def PlayDiceGame(Characters, Items, OtherCharacterName):
		PlayerScore = 0
		OtherCharacterScore = 0
		DiceGamePossible, IndexOfPlayerDie, IndexOfOtherCharacter, IndexOfOtherCharacterDie = CheckIfDiceGamePossible(Items, Characters, OtherCharacterName)
		if not DiceGamePossible:
		print("You can't play a dice game.")
		else:
		Position = GetPositionOfCommand(Items[IndexOfPlayerDie].Commands, "use")
		ResultForCommand = GetResultForCommand(Items[IndexOfPlayerDie].Results, Position)
		Results = [0, 0, 0]
		for Count in range(3):
		Results[Count] = RollDie(ResultForCommand[5], ResultForCommand[7])
		print("You rolled a " + str(Results[Count]) + ".")
		Largest = Results[0]
		Smallest = Results[0]
		Middle = Results[0]
		for Count in range(1,3):
		if Results[Count] > Largest:
		Middle = Largest
		Largest = Results[Count]
		elif Results[Count] < Smallest:
		Middle = Smallest
		Smallest = Results[Count]
		else:
		Middle = Results[Count]
		PlayerScore = Largest * 100 + Middle * 10 + Smallest
		Position = GetPositionOfCommand(Items[IndexOfOtherCharacterDie].Commands, "use")
		ResultForCommand = GetResultForCommand(Items[IndexOfOtherCharacterDie].Results, Position)
		for Count in range(3):
		Results[Count] = RollDie(ResultForCommand[5], ResultForCommand[7])
		print("They rolled a " + str(Results[Count]) + ".")
		OtherCharacterScore += Results[Count] * 10 ** Count
		print("Your score:", PlayerScore)
		print("Their score:", OtherCharacterScore)
		if PlayerScore > OtherCharacterScore:
		print("You win!")
		Items = TakeItemFromOtherCharacter(Items, Characters[IndexOfOtherCharacter].ID)
		elif PlayerScore < OtherCharacterScore:
		print("You lose!")
		Items = TakeRandomItemFromPlayer(Items, Characters[IndexOfOtherCharacter].ID)
		else:
		print("Draw!")
		return Items
			8

Python 2

05	1	Word1 = "" Word2 = "" CanBeMadeFromSecondWord = True Word1 = raw_input("Enter the first word: ") Word2 = raw_input("Enter the second word: ") for Pos in range(0, len(Word1)): if Word1.count(Word1[Pos]) > Word2.count(Word1[Pos]): CanBeMadeFromSecondWord = False if CanBeMadeFromSecondWord: print "Yes" else: print "No" **Alternative answer** Word1 = "" Word2 = "" CanBeMadeFromSecondWord = True Word1 = raw_input("Enter the first word: ") Word2 = raw_input("Enter the second word: ") for Pos in range(0, len(Word1)): if Word1[Pos] in Word2: Word2 = Word2.replace(Word1[Pos], "", 1) else: CanBeMadeFromSecondWord = False if CanBeMadeFromSecondWord: print "Yes" else: print "No" **Alternative answer** Word1 = "" Word2 = "" CanBeMadeFromSecondWord = True Counts = [[0,0], [0,0]] Word1 = raw_input("Enter the first word: ") Word2 = raw_input("Enter the second word: ") for ch in Word1: Counts[ord(ch) - 65][0] += 1 for ch in Word2: Counts[ord(ch) - 65][1] += 1 Pos = 0 while Pos <= 25: if Counts[Pos][0] > Counts[Pos][1]: CanBeMadeFromSecondWord = False Pos += 1 if CanBeMadeFromSecondWord: print "Yes" else: print "No"	12
10	1	def PlayGame(Characters, Items, Places):	4

```
  ...
    elif Command == "quit":
      Say("You decide to give up, try again another time.")
      StopGame = True
    else:
      RNo = random.randint(0, 1)
      if RNo == 0:
        print "Sorry, you don't know how to " + Command + "."
      else:
        print "Sorry, I don't know what " + Command + " means."
  raw_input()
```

Alternative answer

```
    elif Command == "quit":
      Say("You decide to give up, try again another time.")
      StopGame = True
    else:
      if GetRandomNumber(0, 1) == 0:
        print "Sorry, you don't know how to " + Command + "."
      else:
        print "Sorry, I don't know what " + Command + " means."
  raw_input()
```

11 | 1 | 7

```
def GetItem(Items, ItemToGet, CurrentLocation):
  SubCommand = ""
  SubCommandParameter = ""
  CanGet = False
  IndexOfItem = GetIndexOfItem(ItemToGet, -1, Items)
  if IndexOfItem == -1:
    print "You can't find " + ItemToGet + "."
  elif Items[IndexOfItem].Location == INVENTORY:
    print "You have already got that!"
  elif not "get" in Items[IndexOfItem].Commands:
    print "You can't get " + ItemToGet + "."
  elif Items[IndexOfItem].Location >= MINIMUM_ID_FOR_ITEM and
Items[GetIndexOfItem("", Items[IndexOfItem].Location,
Items)].Location != CurrentLocation:
    print "You can't find " + ItemToGet + "."
  elif Items[IndexOfItem].Location < MINIMUM_ID_FOR_ITEM and
Items[IndexOfItem].Location != CurrentLocation:
    print "You can't find " + ItemToGet + "."
  else:
    CanGet = True
  if CanGet:
    NoOfItems = 0
    for Thing in Items:
      if Thing.Location == INVENTORY:
        NoOfItems += 1
    if NoOfItems >= 5:
      print "You can't carry anything else."
    else:
      Position =
GetPositionOfCommand(Items[IndexOfItem].Commands, "get")
      ResultForCommand =
GetResultForCommand(Items[IndexOfItem].Results, Position)
      SubCommand, SubCommandParameter =
```

```
ExtractResultForCommand(SubCommand, SubCommandParameter,
ResultForCommand)
      if SubCommand == "say":
        Say(SubCommandParameter)
      elif SubCommand == "win":
        Say("You have won the game")
        return True, Items
      if "gettable" in Items[IndexOfItem].Status:
        Items = ChangeLocationOfItem(Items, IndexOfItem,
INVENTORY)
        print "You have got that now."
  return False, Items
```

Alternative answer

```
  SubCommand = ""
  SubCommandParameter = ""
  CanGet = False
  ItemsCount = 0
  for Thing in Items:
    if Thing.Location == INVENTORY:
      ItemsCount += 1
  IndexOfItem = GetIndexOfItem(ItemToGet, -1, Items)
  if ItemsCount >= 5:
    print "You already have five items, you cannot carry
anymore"
  elif IndexOfItem == -1:
    print "You can't find " + ItemToGet + "."
```

12 | 1

```
def PlayGame(Characters, Items, Places):
...
    Instruction = GetInstruction()
    Command, Instruction = ExtractCommand(Instruction)
    if Command == "get":
      StopGame, Items = GetItem(Items, Instruction,
Characters[0].CurrentLocation)
    elif Command == "drop":
      Items = DropItem(Items, Instruction,
Characters[0].CurrentLocation)
    elif Command == "use":
      StopGame, Items = UseItem(Items, Instruction,
Characters[0].CurrentLocation, Places)
```

12

```
def DropItem(Items, ItemToDrop, Location):
  CanDrop = True
  IndexOfItem = GetIndexOfItem(ItemToDrop, -1, Items)
  if not(IndexOfItem == -1 or Items[IndexOfItem].Location ==
1001):
    print "You don't have that!"
    CanDrop = False
  if CanDrop:
    if "fragile" in Items[IndexOfItem].Status:
      del(Items[IndexOfItem])
      print "It broke!"
    else:
      ChangeLocationOfItem(Items, IndexOfItem, Location)
```

		print "You have dropped it." **return Items**	

13	1	def PlayDiceGame(Characters, Items, OtherCharacterName):
		  PlayerScore = 0
		  OtherCharacterScore = 0
		  DiceGamePossible, IndexOfPlayerDie, IndexOfOtherCharacter, IndexOfOtherCharacterDie = CheckIfDiceGamePossible(Items, Characters, OtherCharacterName)
		  if not DiceGamePossible:
		  print "You can't play a dice game."
		  else:
		  Position = GetPositionOfCommand(Items[IndexOfPlayerDie].Commands, "use")
		  ResultForCommand = GetResultForCommand(Items[IndexOfPlayerDie].Results, Position)
		  Results = [0, 0, 0]
		  for Count in range(3):
		  Results[Count] = RollDie(ResultForCommand[5], ResultForCommand[7])
		  print "You rolled a " + str(Results[Count]) + "."
		  Largest = Results[0]
		  Smallest = Results[0]
		  Middle = Results[0]
		  for Count in range(1,3):
		  if Results[Count] > Largest:
		  Middle = Largest
		  Largest = Results[Count]
		  elif Results[Count] < Smallest:
		  Middle = Smallest
		  Smallest = Results[Count]
		  else:
		  Middle = Results[Count]
		  PlayerScore = Largest * 100 + Middle * 10 + Smallest
		  Position = GetPositionOfCommand(Items[IndexOfOtherCharacterDie].Commands, "use")
		  ResultForCommand = GetResultForCommand(Items[IndexOfOtherCharacterDie].Results, Position)
		  for Count in range(3):
		  Results[Count] = RollDie(ResultForCommand[5], ResultForCommand[7])
		  print "They rolled a " + str(Results[Count]) + "."
		  OtherCharacterScore += Results[Count] * 10 ** Count
		  print "Your score:", PlayerScore
		  print "Their score:", OtherCharacterScore
		  if PlayerScore > OtherCharacterScore:
		  print "You win!"
		  Items = TakeItemFromOtherCharacter(Items, Characters[IndexOfOtherCharacter].ID)
		  elif PlayerScore < OtherCharacterScore:
		  print "You lose!"
		  Items = TakeRandomItemFromPlayer(Items, Characters[IndexOfOtherCharacter].ID)
		  else:
		  print "Draw!"
		  return Items
			8

Pascal

05	1	program Project1;
		
		{$mode objfpc}{$H+}
		
		uses
		  Classes, SysUtils;
		
		var
		  word1, word2: string;
		  characterCount: array['A' .. 'Z'] of integer;
		  character: char;
		  canBeMade: boolean;
		
		begin
		  for character := 'A' to 'Z' do
		  characterCount[character] := 0;
		  write('First word: ');
		  readln(word1);
		  write('Second word: ');
		  readln(word2);
		  for character in word2 do
		  inc(characterCount[character]);
		  canBeMade := true;
		  for character in word1 do
		  begin
		  dec(characterCount[character]);
		  if characterCount[character] < 0 then
		  canBeMade := false
		  end;
		  if canBeMade then
		  writeln(word1, ' can be made with the letters in ', word2)
		  else
		  writeln(word1, ' cannot be made with the letters in ', word2);
		  readln;
		end.
			12

10	1	procedure PlayGame(Characters: TCharacterArray; Items: TItemArray; Places: TPlaceArray); ... else if Command = 'quit' then begin Say('You decide to give up, try again another time'); StopGame := true; end else begin **if random < 0.5 then** writeln('Sorry, you don''t know how to ', Command, '.') **else** **writeln('Sorry, I don’t know what ', Command, ' means.');** end; end; readln; end;	4
11	1	procedure GetItem(Items: TItemArray; ItemToGet: string; CurrentLocation: integer; var StopGame: boolean); var ResultForCommand: string;	7

```
  SubCommand: string;
  SubCommandParameter: string;
  IndexOfItem: integer;
  Position: integer;
  CanGet: Boolean;
  ItemsInInventory: integer;
  Item: TItem;
begin
  SubCommand := '';
  SubCommandParameter := '';
  CanGet := false;
  IndexOfItem := GetIndexOfItem(ItemToGet, -1, Items);
  if IndexOfItem = -1 then
    writeln('You can''t find ', ItemToGet, '.')
  else if Items[IndexOfItem].Location = Inventory then
    writeln('You have already got that!')
  else if pos('get', Items[IndexOfItem].Commands) = 0 then
    writeln('You can''t get ', ItemToGet, '.')
  else if (Items[IndexOfItem].Location >= MinimumIDForItem) and
(Items[GetIndexOfItem('', Items[IndexOfItem].Location,
Items)].Location <> CurrentLocation) then
    writeln('You can''t find ', ItemToGet, '.')
  else if (Items[INdexOfItem].Location < MinimumIDForItem) and
(Items[IndexOfItem].Location <> CurrentLocation) then
    writeln('You can''t find ', ItemToGet, '.')
  else
    CanGet := true;
  if CanGet then
    begin
      ItemsInInventory := 0;
      for Item in Items do
        if Item.Location = Inventory then
          inc(ItemsInInventory);
      if ItemsInInventory >= 5 then
        writeln('You have too many items in your inventory to carry
any more.');
      else
        begin
          Position :=
GetPositionOfCommand(Items[IndexOfItem].Commands, 'get');
          ResultForCommand :=
GetResultForCommand(Items[IndexOfItem].Results, Position);
          ExtractResultForCommand(SubCommand, SubCommandParameter,
ResultForCommand);
          if SubCommand = 'say' then
            Say(SubCommandParameter)
          else if SubCommand = 'win' then
            begin
              say('You have won the game');
              StopGame := true;
              exit;
            end;
          if pos('gettable', Items[IndexOfItem].Status) <> 0 then
            begin
              ChangeLocationOfItem(Items, IndexOfItem, Inventory);
              writeln('You have got that now.');
            end;
        end;
    end;
end;
```

12	1	procedure PlayGame(Characters: TCharacterArray; Items: TItemArray; Places: TPlaceArray); ... else if Command = 'quit' then begin Say('You decide to give up, try again another time'); StopGame := true; end **else if Command = 'drop' then** **DropItem(Items, Instruction, Characters[0].CurrentLocation)** else begin writeln('Sorry, you don''t know how to ', Command, '.') end; end; readln; end; **procedure DropItem(Items: TItemArray; ItemToDrop: string; CurrentLocation: integer);** **var** **ResultForCommand: string;** **SubCommand: string;** **SubCommandParameter: string;** **IndexOfItem: integer;** **Position: integer;** **CanDrop: boolean;** **IsFragile: boolean;** **begin** **SubCommand := '';** **SubCommandParameter := '';** **CanDrop := true;** **IsFragile := false;** **IndexOfItem := GetIndexOfItem(ItemToDrop, -1, Items);** **if IndexOfItem = -1 then** **begin** **writeln('You can''t find ', ItemToDrop, '.');** **CanDrop := false** **end** **else if Items[IndexOfItem].Location <> Inventory then** **begin** **writeln('You don''t have that!');** **CanDrop := false;** **end** **else if pos('fragile', Items[IndexOfItem].Status) <> 0 then** **IsFragile := true;** **if CanDrop then** **begin** **if IsFragile then** **begin** **Items[IndexOfItem].Location := -1;** **writeln(' It broke!')** **end** **else** **begin** **Items[IndexOfItem].Location := CurrentLocation;** **writeln('You have dropped it.');** **end;** **end;** **end;**	12

13	1	procedure PlayDiceGame(Characters: TCharacterArray; Items: TItemArray; OtherCharacterName: string); var PlayerScore: array[1 .. 3] of integer; PlayerScoreTotal: integer; OtherCharacterScore: array[1 .. 3] of integer; OtherCharacterScoreTotal: integer; IndexOfPlayerDie: integer; IndexOfOtherCharacterDie: integer; Position: integer; IndexOfOtherCharacter: integer; PlayerWins: boolean; ResultForCommand: string; DiceGamePossible: boolean; **i: integer;** **procedure SwapIfNeeded(i, j : integer);** **var** **temp: integer;** **begin** **if PlayerScore[i] > PlayerScore[j] then** **begin** **temp := PlayerScore[i];** **PlayerScore[i] := PlayerScore[j];** **PlayerScore[j] := temp;** **end;** **end;** begin **for i := 1 to 3 do** **begin** **PlayerScore[i] := 0;** **OtherCharacterScore[i] := 0;** **end;** PlayerWins := false; DiceGamePossible := CheckIfDiceGamePossible(Items, Characters, IndexOfPlayerDie, IndexOfOtherCharacter, IndexOfOtherCharacterDie, OtherCharacterName); if not DiceGamePossible then writeln('You can''t play a dice game.') else begin Position := GetPositionOfCommand(Items[IndexOfPlayerDie].Commands, 'use'); ResultForCommand := GetResultForCommand(Items[IndexOfPlayerDie].Results, Position); **for i := 1 to 3 do** **begin** **PlayerScore[i] := RollDie(ResultForCommand[6], ResultForCommand[8]);** **writeln('You rolled a ', inttostr(PlayerScore[i]), '.');** **end;** Position := GetPositionOfCommand(Items[IndexOfOtherCharacterdie].Commands,	8

```
'use');
      ResultForCommand :=
GetResultForCommand(Items[IndexOfOtherCharacterDie].Results,
Position);
      for i := 1 to 3 do
        begin
          OtherCharacterScore[i] :=
RollDie(ResultForCommand[6], ResultForCommand[8]);
          writeln('They rolled a ','
inttostr(OtherCharacterScore[i]), '.');
        end;
      SwapIfNeeded(1, 2);
      SwapIfNeeded(2, 3);
      SwapIfNeeded(1, 2);
      PlayerScoreTotal := 100 * PlayerScore[3] + 10 *
PlayerScore[2] + PlayerScore[1];
      OtherCharacterScoreTotal := 100 * OtherCharacterScore[3]
+ 10 * OtherCharacterScore[2] + OtherCharacterScore[1];
      if PlayerScoreTotal > OtherCharacterScoreTotal then
        begin
          writeln('You win!');
          TakeItemFromOtherCharacter(Items,
Characters[IndexOfOtherCharacter].ID);
        end
      else if PlayerScoreTotal < OtherCharacterScoreTotal then
        begin
          writeln('You lose!');
          TakeRandomItemFromPlayer(Items,
Characters[IndexOfOtherCharacter].ID);
        end
      else
        writeln('Draw!');
    end;
end;
```

C#

05	1	`string word1, word2;`
		`bool CanBeMadeFromSecondWord = true;`
		`Console.Write("Enter the first word: ");`
		`word1 = Console.ReadLine();`
		`Console.Write("Enter the second word: ");`
		`word2 = Console.ReadLine();`
		`for (int i = 0; i < word1.Length; i++)`
		`{`
		`  if (word1.Split(word1[i]).Length - 1 >`
		`Word2.Split(word1[i]).Length - 1)`
		`  {`
		`  CanBeMadeFromSecondWord = false;`
		`  }`
		`}`
		`if (CanBeMadeFromSecondWord)`
		`{`
		`  Console.WriteLine("Yes");`
		`}`
		`else`
		`{`
		`  Console.WriteLine("No");`
		`}`
		`Console.ReadLine();`
		
		Alternative answer
		`string word1, word2;`
		`bool CanBeMadeFromSecondWord = true;`
		`int Loc = 0;`
		`Console.Write("Enter the first word: ");`
		`word1 = Console.ReadLine();`
		`Console.Write("Enter the second word: ");`
		`word2 = Console.ReadLine();`
		`for (int i = 0; i < word1.Length; i++)`
		`{`
		`  if (word2.Contains(word1[i]))`
		`  {`
		`  Loc = word2.IndexOf(word1[i]);`
		`  word2 = word2.Remove(Loc, 1);`
		`  }`
		`  else`
		`  {`
		`  CanBeMadeFromSecondWord = false;`
		`  }`
		`}`
		`if (CanBeMadeFromSecondWord)`
		`{`
		`  Console.WriteLine("Yes");`
		`}`
		`else`
		`{`
		`  Console.WriteLine("No");`
		`}`
		`Console.ReadLine();`
			12

		Alternative answer
		`string word1, word2;`
		`bool CanBeMadeFromSecondWord = true;`
		`int[,] counts = new int[26, 2];`
		`Console.Write("Enter the first word: ");`
		`word1 = Console.ReadLine();`
		`Console.Write("Enter the second word: ");`
		`word2 = Console.ReadLine();`
		`foreach (var ch in word1)`
		`{`
		`  counts[ch - 65, 0]++;`
		`}`
		`foreach (var ch in word2)`
		`{`
		`  counts[ch - 65, 1]++;`
		`}`
		`int pos = 0;`
		`while (pos <= 25)`
		`{`
		`  if (counts[pos, 0] > counts[pos, 1])`
		`  {`
		`  CanBeMadeFromSecondWord = false;`
		`  }`
		`  pos++;`
		`}`
		`if (CanBeMadeFromSecondWord)`
		`{`
		`  Console.WriteLine("Yes");`
		`}`
		`else`
		`{`
		`  Console.WriteLine("No");`
		`}`
		`Console.ReadLine();`	

10	1	`private static void PlayGame(List<Character> characters, List<Item> items, List<Place> places)`
		`...`
		`  case "playdice":`
		`  PlayDiceGame(characters, items, instruction);`
		`  break;`
		`  case "quit":`
		`  Say("You decide to give up, try again another time.");`
		`  stopGame = true;`
		`  break;`
		`  default:`
		`  Random rnd = new Random();`
		`  int rno = rnd.Next(0, 2);`
		`  if (rno == 1)`
		`  {`
		`  Console.WriteLine("Sorry, you don't know how to " + command + ".");`
		`  }`
		`  else`
		`  {`
		`  Console.WriteLine("Sorry, I don't know what " + command + " means.");`
		`  }`
		`  break;`
			4

```
        }
    }
```

Alternative answer

```
default:
    if (GetRandomNumber(0, 1) == 1)
    {
        Console.WriteLine("Sorry, you don't know how to " +
command + ".");
    }
    else
    {
        Console.WriteLine("Sorry, I don't know what " + command
+ " means.");
    }
    break;
```

11 | 1 | 7

```
private static void GetItem(List<Item> items, string itemToGet,
int currentLocation, ref bool stopGame)
{
    string resultForCommand, subCommand = "",
subCommandParameter = "";
    int indexOfItem, position;
    bool canGet = false;
    indexOfItem = GetIndexOfItem(itemToGet, -1, items);
    if (indexOfItem == -1)
    {
        Console.WriteLine("You can't find " + itemToGet + ".");
    }
    else if (items[indexOfItem].Location == Inventory)
    {
        Console.WriteLine("You have already got that!");
    }
    else if (!items[indexOfItem].Commands.Contains("get"))
    {
        Console.WriteLine("You can't get " + itemToGet + ".");
    }
    else if (items[indexOfItem].Location >= MinimumIDForItem &&
items[GetIndexOfItem("", items[indexOfItem].Location,
items)].Location != currentLocation)
    {
        Console.WriteLine("You can't find " + itemToGet + ".");
    }
    else if (items[indexOfItem].Location < MinimumIDForItem &&
items[indexOfItem].Location != currentLocation)
    {
        Console.WriteLine("You can't find " + itemToGet + ".");
    }
    else
    {
      canGet = true;
    }
    if (canGet)
    {
        int noOfItems = 0;
        foreach (var thing in items)
        {
```

```
            if (thing.Location == Inventory)
            {
                noOfItems++;
            }
        }
        if (noOfItems >= 5)
        {
            Console.WriteLine("You can't carry anything else.");
        }
        else
        {
            position =
GetPositionOfCommand(items[indexOfItem].Commands, "get");
            resultForCommand =
GetResultForCommand(items[indexOfItem].Results, position);
            ExtractResultForCommand(ref subCommand, ref
subCommandParameter, resultForCommand);
            if (subCommand == "say")
            {
                Say(subCommandParameter);
            }
            else if (subCommand == "win")
            {
                Say("You have won the game");
                stopGame = true;
                return;
            }
            if (items[indexOfItem].Status.Contains("gettable"))
            {
                ChangeLocationOfItem(items, indexOfItem,
Inventory);
                Console.WriteLine("You have got that now.");
            }
        }
    }
}
```

Alternative answer

```
int indexOfItem, position;
bool canGet = false;
int itemsCount = 0;
foreach (var thing in items)
{
    if (thing.Location == Inventory)
    {
        itemsCount++;
    }
}
indexOfItem = GetIndexOfItem(itemToGet, -1, items);
if (itemsCount >= 5)
{
    Console.WriteLine("You already have five items, you cannot
carry any more");
}
else if (indexOfItem == -1)
```

12	1	`private static void PlayGame(List<Character> characters, List<Item> items, List<Place> places)`	12

```
...
        instruction = GetInstruction();
        command = ExtractCommand(ref instruction);
        switch (command)
        {
            case "drop":
                DropItem(items, instruction,
characters[0].CurrentLocation);
                break;
            case "get":
                GetItem(items, instruction,
characters[0].CurrentLocation, ref stopGame);
                break;
...
            default:
                Console.WriteLine("Sorry, you don't know how to
" + command + ".");
                break;
        }
    }
    Console.ReadLine();
}

private static void DropItem(List<Item> items, string
itemToDrop, int location)
{
    int indexOfItem;
    bool canDrop = true;
    indexOfItem = GetIndexOfItem(itemToDrop, -1, items);
    if (indexOfItem == -1 || items[indexOfItem].Location !=
1001)
    {
        Console.WriteLine("You don't have that!");
        canDrop = false;
    }
    if (canDrop)
    {
        if (items[indexOfItem].Status.Contains("fragile"))
        {
            items.RemoveAt(indexOfItem);
            Console.WriteLine("It broke!");
        }
        else
        {
            ChangeLocationOfItem(items, indexOfItem, location);
            Console.WriteLine("You have dropped it.");
        }
    }
}
```

Note: Incorrect if '|' used instead of '||' for the OR operator in the first if statement.

13	1	`private static void PlayDiceGame(List<Character> characters, List<Item> items, string otherCharacterName)`	8

```
...
    if (!DiceGamePossible)
    {
        Console.WriteLine("You can't play a dice game.");
    }
    else
    {
        position =
GetPositionOfCommand(items[indexOfPlayerDie].Commands, "use");
        ResultForCommand =
GetResultForCommand(items[indexOfPlayerDie].Results, position);
        int[] results = new int[3];
        for (int i = 0; i < results.Length; i++)
        {
            results[i] = RollDie(ResultForCommand[5].ToString(),
ResultForCommand[7].ToString());
            Console.WriteLine("You rolled a " + results[i] +
".");
        }
        int largest = results[0];
        int smallest = results[0];
        int middle = results[0];
        for (int i = 1; i < results.Length; i++)
        {
            if (results[i] > largest)
            {
                middle = largest;
                largest = results[i];
            }
            else if (results[i] < smallest)
            {
                middle = smallest;
                smallest = results[i];
            }
            else
            {
                middle = results[i];
            }
        }
        playerScore = largest * 100 + middle * 10 + smallest;
        position =
GetPositionOfCommand(items[indexOfOtherCharacterDie].Commands,
"use");
        ResultForCommand =
GetResultForCommand(items[indexOfOtherCharacterDie].Results,
position);
        for (int i = 0; i < results.Length; i++)
        {
            results[i] = RollDie(ResultForCommand[5].ToString(),
ResultForCommand[7].ToString());
            Console.WriteLine("They rolled a " + results[i] +
".");
            otherCharacterScore += results[i] *
(int)Math.Pow(10, i);
        }
        Console.WriteLine("Your score: " + playerScore);
        Console.WriteLine("Their score: " +
otherCharacterScore);
        if (playerScore > otherCharacterScore)
        {
            Console.WriteLine("You win!");
```

```
            TakeItemFromOtherCharacter(items,
characters[indexOfOtherCharacter].ID);
        }
        else if (playerScore < otherCharacterScore)
        {
            Console.WriteLine("You lose!");
            TakeRandomItemFromPlayer(items,
characters[indexOfOtherCharacter].ID);
        }
        else
        {
            Console.WriteLine("Draw!");
        }
    }
}
```

Java

05	1	`String word1, word2;`
		`boolean canBeMadeFromSecondWord = true;`
		`int pos;`
		`Console.write("Enter the first word: ");`
		`word1 = Console.readLine();`
		`Console.write("Enter the second word: ");`
		`word2 = Console.readLine();`
		`for (pos = 0; pos < word1.length(); pos++) {`
		`  if (word1.split(word1.substring(pos, pos + 1)).length >`
		`  word2.split(word1.substring(pos, pos + 1)).length) {`
		`  canBeMadeFromSecondWord = false;`
		`  }`
		`}`
		`if (canBeMadeFromSecondWord) {`
		`  Console.writeLine("Yes");`
		`} else {`
		`  Console.writeLine("No");`
		`}`
		`Console.readLine();`
		
		Alternative answer
		
		`String word1, word2;`
		`boolean canBeMadeFromSecondWord = true;`
		`int pos, loc;`
		`Console.write("Enter the first word: ");`
		`word1 = Console.readLine();`
		`Console.write("Enter the second word: ");`
		`word2 = Console.readLine();`
		`for (pos = 0; pos < word1.length(); pos++) {`
		`  if (word2.contains(word1.substring(pos, pos + 1))) {`
		`  loc = word2.indexOf(word1.substring(pos, pos + 1));`
		`  word2 = word2.replaceFirst(word1.substring(pos, pos + 1), "");`
		`  } else {`
		`  canBeMadeFromSecondWord = false;`
		`  }`
		`}`
		`if (canBeMadeFromSecondWord) {`
		`  Console.writeLine("Yes");`
		`} else {`
		`  Console.writeLine("No");`
		`}`
		`Console.readLine();`
		
		Alternative answer
		
		`String word1, word2;`
		`boolean canBeMadeFromSecondWord = true;`
		`int pos;`
		`int[][] counts = new int[26][2];`
		`Console.write("Enter the first word: ");`
		`word1 = Console.readLine();`
		`Console.write("Enter the second word: ");`
			12

```
word2 = Console.readLine();
for (pos = 0; pos < word1.length(); pos++) {
  counts[(int)word1.charAt(pos) - 65][0]++;
}
for (pos = 0; pos < word2.length(); pos++) {
  counts[(int)word2.charAt(pos) - 65][1]++;
}
pos = 0;
while (pos <= 25) {
  if (counts[pos][0] > counts[pos][1]) {
    canBeMadeFromSecondWord = false;
  }
  pos++;
}
if (canBeMadeFromSecondWord) {
  Console.writeLine("Yes");
} else {
  Console.writeLine("No");
}
Console.readLine();
```

10 | 1 | 4

```
void playGame(ArrayList<Character> characters, ArrayList<Item>
items, ArrayList<Place> places) {
  …
    case "quit":
      say("You decide to give up, try again another time.");
      stopGame = true;
      break;
    default:
      Random rnd = new Random();
      int rNo = (int)(rnd.nextDouble()*2);
      if (rNo == 0) {
        Console.writeLine("Sorry, you don't know how to " +
command + ".");
      } else {
        Console.writeLine("Sorry, I don't know what " + command +
" means.");
      }
    }
  }
  Console.readLine();
}
```

Alternative answer

```
default:
  if (getRandomNumber(0, 1) == 0) {
    Console.writeLine("Sorry, you don't know how to " + command +
".");
  } else {
    Console.writeLine("Sorry, I don't know what " + command + "
means.");
  }
}
```

11 | 1 | 7

```
boolean getItem(ArrayList<Item> items, String itemToGet, int
```

```
currentLocation) {
  boolean stopGame = false, canGet = false;
  String resultForCommand, subCommand = "", subCommandParameter =
"";
  int indexOfItem, position;
  indexOfItem = getIndexOfItem(itemToGet, -1, items);
  if (indexOfItem == -1) {
    Console.writeLine("You can't find " + itemToGet + ".");
  } else if (items.get(indexOfItem).location == INVENTORY) {
    Console.writeLine("You have already got that!");
  } else if (!items.get(indexOfItem).commands.contains("get")) {
    Console.writeLine("You can't get " + itemToGet + ".");
  } else if (items.get(indexOfItem).location >=
MINIMUM_ID_FOR_ITEM && items.get(getIndexOfItem("",
items.get(indexOfItem).location, items)).location !=
currentLocation) {
    Console.writeLine("You can't find " + itemToGet + ".");
  } else if (items.get(indexOfItem).location < MINIMUM_ID_FOR_ITEM
&& items.get(indexOfItem).location != currentLocation) {
    Console.writeLine("You can't find " + itemToGet + ".");
  } else {
    canGet = true;
  }
  if (canGet) {
    int noOfItems = 0;
    for (Item thing : items) {
      if (thing.location == INVENTORY) {
        noOfItems++;
      }
    }
    if (noOfItems >= 5) {
       Console.writeLine("You can't carry anything else.");
    } else {
      position =
getPositionOfCommand(items.get(indexOfItem).commands, "get");
      resultForCommand =
getResultForCommand(items.get(indexOfItem).results, position);
      String[] returnArray = extractResultForCommand(subCommand,
subCommandParameter, resultForCommand);
      subCommand = returnArray[0];
      subCommandParameter = returnArray[1];
      if (subCommand.equals("say")) {
        say(subCommandParameter);
      } else if (subCommand.equals("win")) {
        say("You have won the game");
        stopGame = true;
        return stopGame;
      }
      if (items.get(indexOfItem).status.contains("gettable")) {
        changeLocationOfItem(items, indexOfItem, INVENTORY);
        Console.writeLine("You have got that now.");
      }
    }
  }
  return stopGame;
}
```

Alternative answer

```
boolean stopGame = false, canGet = false;
String resultForCommand, subCommand = "", subCommandParameter =
"";
int indexOfItem, position, itemCount = 0;
for (Item thing : items) {
  if (thing.location == INVENTORY) {
    itemCount++;
  }
}
indexOfItem = getIndexOfItem(itemToGet, -1, items);
if (indexOfItem == -1) {
  Console.writeLine("You can't find " + itemToGet + ".");
} else if (items.get(indexOfItem).location == INVENTORY) {
  Console.writeLine("You have already got that!");
} else if (!items.get(indexOfItem).commands.contains("get")) {
  Console.writeLine("You can't get " + itemToGet + ".");
} else if (items.get(indexOfItem).location >= MINIMUM_ID_FOR_ITEM
&& items.get(getIndexOfItem("", items.get(indexOfItem).location,
items)).location != currentLocation) {
  Console.writeLine("You can't find " + itemToGet + ".");
} else if (items.get(indexOfItem).location < MINIMUM_ID_FOR_ITEM
&& items.get(indexOfItem).location != currentLocation) {
  Console.writeLine("You can't find " + itemToGet + ".");
} else if (itemCount >=5) {
  Console.writeLine("You already have five items, you cannot carry
any more");
} else {
  canGet = true;
}
if (canGet) {
  position = getPositionOfCommand(items.get(indexOfItem).commands,
"get");
```

Alternative answer (Functional programming used to obtain the count of items)

```
boolean stopGame = false, canGet = false;
String resultForCommand, subCommand = "", subCommandParameter =
"";
int indexOfItem, position, itemCount = 0;
itemCount = items.stream().filter((thing) -> (thing.location ==
INVENTORY)).map((_item) -> 1).reduce(itemCount, Integer::sum);
indexOfItem = getIndexOfItem(itemToGet, -1, items);
if (indexOfItem == -1) {
  Console.writeLine("You can't find " + itemToGet + ".");
} else if (items.get(indexOfItem).location == INVENTORY) {
  Console.writeLine("You have already got that!");
} else if (!items.get(indexOfItem).commands.contains("get")) {
  Console.writeLine("You can't get " + itemToGet + ".");
} else if (items.get(indexOfItem).location >= MINIMUM_ID_FOR_ITEM
&& items.get(getIndexOfItem("", items.get(indexOfItem).location,
items)).location != currentLocation) {
  Console.writeLine("You can't find " + itemToGet + ".");
} else if (items.get(indexOfItem).location < MINIMUM_ID_FOR_ITEM
&& items.get(indexOfItem).location != currentLocation) {
  Console.writeLine("You can't find " + itemToGet + ".");
} else if (itemCount >=5) {
  Console.writeLine("You already have five items, you cannot carry
any more");
```

```
} else {
  canGet = true;
}
if (canGet) {
  position = getPositionOfCommand(items.get(indexOfItem).commands,
"get");
```

12 | 1

```
void playGame(ArrayList<Character> characters, ArrayList<Item>
items, ArrayList<Place> places) {
  boolean stopGame = false, moved = true;
  String instruction, command;
  int resultOfOpenClose;
  while (!stopGame) {
    if (moved) {
      Console.writeLine();
      Console.writeLine();
Console.writeLine(places.get(characters.get(0).currentLocation -
1).description);
      displayGettableItemsInLocation(items,
characters.get(0).currentLocation);
        moved = false;
      }
      instruction = getInstruction();
      String[] returnStrings = extractCommand(instruction);
      command = returnStrings[0];
      instruction = returnStrings[1];
      switch (command)
      {
        case "drop":
          dropItem(items, instruction,
characters.get(0).currentLocation);
          break;
        case "get":
          getItem(items, instruction.instruction,
characters.get(0).currentLocation, stopGame);
          break;
        case "use":
          useItem(items, instruction.instruction,
characters.get(0).currentLocation, stopGame, places);
          break;
        case "go":
          moved = go(characters.get(0), instruction.instruction,
places.get(characters.get(0).currentLocation - 1));
          break;
        case "read":
          readItem(items, instruction.instruction,
characters.get(0).currentLocation);
          break;
        case "examine":
          examine(items, characters, instruction.instruction,
characters.get(0).currentLocation);
          break;
        case "open":
          resultOfOpenClose = openClose(true, items, places,
instruction.instruction, characters.get(0).currentLocation);
          displayOpenCloseMessage(resultOfOpenClose, true);
          break;
        case "close":
```

12

```
          resultOfOpenClose = openClose(false, items, places,
instruction.instruction, characters.get(0).currentLocation);
          displayOpenCloseMessage(resultOfOpenClose, false);
          break;
        case "move":
          moveItem(items, instruction.instruction,
characters.get(0).currentLocation);
          break;
        case "say":
          say(instruction.instruction);
          break;
        case "playdice":
          playDiceGame(characters, items,
instruction.instruction);
          break;
        case "quit":
          say("You decide to give up, try again another time.");
          stopGame = true;
          break;
        default:
          if(getRandomNumber(0, 1) == 0) {
            Console.writeLine("Sorry, you don't know how to " +
command + ".");
          } else {
            Console.writeLine("Sorry, I don't know what " +
command + " means.");
          }
      }
    }
  Console.readLine();
  }

void dropItem(ArrayList<Item> items, String itemToDrop, int
currentLocation) {
  int indexOfItem;
  boolean canDrop = true;
  indexOfItem = getIndexOfItem(itemToDrop, -1, items);
  if (indexOfItem == -1 || items.get(indexOfItem).location !=
1001) {
    Console.writeLine("You don't have that!");
    canDrop = false;
  }
  if (canDrop) {
    if (items.get(indexOfItem).status.contains("fragile")) {
      items.remove(indexOfItem);
      Console.writeLine("It broke!");
    } else {
      changeLocationOfItem(items, indexOfItem, currentLocation);
      Console.writeLine("You have dropped it.");
    }
  }
}
```

Note: Incorrect if '|' used instead of '||' for the OR operator in the first if statement.

13	1	void playDiceGame(ArrayList<Character> characters, ArrayList<Item>	8

```
items, String otherCharacterName) {
  int playerScore = 0, otherCharacterScore = 0,
indexOfOtherCharacter = 0, indexOfOtherCharacterDie = 0,
indexOfPlayerDie = 0, position;
  boolean diceGamePossible = false;
  String resultForCommand;
  int[] returnArray = checkIfDiceGamePossible(items, characters,
indexOfPlayerDie, indexOfOtherCharacter, indexOfOtherCharacterDie,
otherCharacterName);
  if (returnArray[0] == 1) {
    diceGamePossible = true;
  }
  indexOfPlayerDie = returnArray[1];
  indexOfOtherCharacter = returnArray[2];
  indexOfOtherCharacterDie = returnArray[3];
  if (!diceGamePossible) {
    Console.writeLine("You can't play a dice game.");
  } else {
    position =
getPositionOfCommand(items.get(indexOfPlayerDie).commands, "use");
    resultForCommand =
getResultForCommand(items.get(indexOfPlayerDie).results,
position);
    //playerScore = rollDie(resultForCommand.substring(5, 6),
resultForCommand.substring(7, 8));
    //Console.writeLine("You rolled a " + playerScore + ".");
    int[] results = new int[3];
    for (int count = 0; count < 3; count++) {
      results[count] = rollDie(resultForCommand.substring(5, 6),
resultForCommand.substring(7, 8));
      Console.writeLine("You rolled a " + results[count] + ".");
    }
    int largest = results[0];
    int smallest = results[0];
    int middle = results[0];
    for (int count = 0; count < 3; count++) {
      if (results[count] > largest) {
        middle = largest;
        largest = results[count];
      } else if (results[count] < smallest) {
        middle = smallest;
        smallest = results[count];
      } else {
        middle = results[count];
      }
    }
    playerScore = largest * 100 + middle * 10 + smallest;
    position =
getPositionOfCommand(items.get(indexOfOtherCharacterDie).commands,
"use");
    resultForCommand =
getResultForCommand(items.get(indexOfOtherCharacterDie).results,
position);
    for (int count = 0; count < 3; count++) {
      results[count] = rollDie(resultForCommand.substring(5, 6),
resultForCommand.substring(7, 8));
      Console.writeLine("They rolled a " + results[count] + ".");
      otherCharacterScore += results[count] * Math.pow(10, count);
    }
    Console.writeLine("Your score: " + playerScore + ".")
    Console.writeLine("They rolled a " + otherCharacterScore +
```

```
".");
    if (playerScore > otherCharacterScore) {
      Console.writeLine("You win!");
      takeItemFromOtherCharacter(items,
characters.get(indexOfOtherCharacter).id);
    } else if (playerScore < otherCharacterScore) {
      Console.writeLine("You lose!");
      takeRandomItemFromPlayer(items,
characters.get(indexOfOtherCharacter).id);
    } else {
      Console.writeLine("Draw!");
    }
  }
```