BREAKTHROUGH!

**Theory Questions (Mark Scheme)**

| **Question** | | **Suggested Solution** | **Total Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- |
| **1** | (a) | e.g. AdditionalScore // CardScore | 1 mark | **A:** Similar names with meaning to explain the score.  **R:** Spaces in names.  **I:** case. |
| (b) | **4 marks:** Ignoring a return value is not good practice **[1]**… One alternative would be to create a new method called MoveCardWithScore **[1]**… which takes the current player Score as a parameter and returns the updated Score **[1]**… and to remove the return value from the current method **[1]**.  **Examples of answers worth less than full marks:**  **3 marks:** make the return type void **[1]**… and move the logic to the place where the card is played to the sequence **[1]**… which is the only time the score is needed **[1]**.  **2 marks:** remove the scoring **[1]**… and create a separate getScore() method **[1]**.  **1 mark:** always check the return value as ignoring it is bad practice **[1]**. | 4 marks | **A:** any reasonable suggestion.  **A:** answers without passing Score as a parameter and dealing with the extra score as per now.  **A:** answers where there is a scoring method in CardCollection which ‘knows’ whether to score or not.  **A:** passing score in by reference and having a new attribute on CardCollection to indicate if a card added/played should affect the score. |
| **2** | (a) | The sequence only allows cards to be added to the end and taken from the same end which is a LIFO structure **[1]**… and a stack is a LIFO structure that would be appropriate **[1]**… For the discard pile, sometimes you need to peek at the whole stack but generally just play cards to the top and then the whole pile is shuffled back in **[1]**… a stack could be suitable for this or the current structure is definitely adequate as it is easy to iterate through and print out when inspecting **[1]**… I would recommend a stack for the sequence and no change for the discard pile **[1]**. | 4 marks | 1 mark for each point (MAX. 4)  **A:** stack for discard pile.  **R:** queue for either. |
| (b) | Create a new class called StackCardCollection or SequenceCardCollection that inherits from CardCollection **[1]**… and adds and overrides methods to ensure that it behaves as a stack; for example, changing RemoveCard so that you can only remove the card at the end **[1]**… or adding an isEmpty() method **[1]**. | 2 marks | 1 mark for each point (MAX. 2)  **A:** any suitable method for a stack, not just isEmpty() |
| (c) | It would mean that when interacting with the sequence you could just remove a card until it’s empty **[1]**… and that the implementation of this would be hidden **[1]**… avoiding the need to know which position to remove a card from and what constitutes the start or end **[1]**. | 2 marks | 1 mark for each point (MAX. 2) |

| **Question** | | **Suggested Solution** | **Total Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- |
| **3** | (a) | Splitting the deck into two halves (or using pointers to do the same thing) **[1]**… Choosing a number of cards from one half and add them to the combined deck (**A:** circular deck with counting solutions) **[1]**… Choosing a number of cards from the other half and add them to the combined deck **[1]**… Using a random number of cards (0 to 5, A. 1 to 5) **[1]**… Taking cards from the bottom of the split decks rather than the top **[1]**… Repeating until the deck is fully combined **[1]**. | 6 marks | 1 mark for each point | |
| (b) | Most algorithms will have a space complexity twice that of the random shuffle that existed before, i.e. storing the deck split and a new merged/combined deck. | 1 mark | **A:** any version of the idea for 1 mark.  **A**: circular deck solutions with space complexity of the same as the storage for the deck as long as they are explained properly. | |
| **4** | (a) | |  |  |  | | --- | --- | --- | | Count | SequenceAsString | Return value | |  | “” |  | | 5 | “P a” |  | | 4 | “, P a” |  | |  | “F a, P a” |  | | 3 | “, F a, P a” |  | |  | “P a, F a, P a” | True | | 5 marks | 1 mark for the Count column (**I:** spaces)  1 mark for a final return value of True  1 mark for the first value in the SequenceAsString column  1 mark for the last value in the SequenceAsString column  1 mark for the correct middle values in the SequenceAsString column (between the first and last value)  **DPT:** -1 only for a missing space | |
| (b) | |  |  |  | | --- | --- | --- | | Count | SequenceAsString | Return value | |  | “” |  | | 5 | “P a” |  | | 4 | “, P a” |  | |  | “F a, P a” | True | | 3 marks | 1 mark for each column  **DPT:** -1 only for a missing space (note that this is across parts (a) and (b) combined, total of -1 for a missing space across the two parts). | |
| **5** | (a) | When a challenge is solved, it is marked as met by changing the value of the protected attribute Met to True **[1]**… When a lock is solved, these challenges are not reset to False **[1]**… Therefore, when a new random lock is chosen it will appear but the moment that a command is executed, GetLockSolved will be called which will mark the lock as solved by setting the attribute LockSolved of the Breakthrough class to True **[1]**… The lock is not removed from the list of locks from which a random lock is chosen when it is solved **[1]**. | 2 marks | 1 mark for each point (MAX. 2) | |

| **Question** | | **Suggested Solution** | **Total Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- |
| **5** | (b) | **Either:**  Once a lock is solved, remove it from the list of locks so that it cannot be selected again when a new random lock is chosen.  **Or:**  Once a lock is solved, iterate through all the challenges and set the attribute Met for each to False. | 2 marks | 2 marks available for either solution as long as the details are clearly explained, otherwise award 1 mark  **A:** any other reasonable solution including the idea of moving LockSolved into the Lock class and checking it when choosing a new lock. | |
| **6** | (a) | As the number of cards in the deck gets smaller then there will be more swaps than possible arrangements of the cards, which makes the additional swaps redundant and inefficient **[1]**…For example, with 2 cards there are only two combinations and with 6 only 720 but there are still 10000 swaps **[1]** …There is also the chance of the algorithm causing an exception with 0 or 1 cards but there is no check for this **[1]**. | 3 marks | Award 1 mark for each point  **A:** any expression of the idea that 10000 is more swaps than you need, which makes the extra ones unnecessary for the first mark.  **A:** any reference or example to decreasing combinations for the second mark. | |
| (b) | The number of swaps could be a measure of deck size or could be set to a lower threshold when the deck is small **[1]**.  This would be done by using a selection statement and setting a swaps variable to a lower value if the deck is short or to 10000 as now if the deck is large **[1]**. | 2 marks | Award 1 mark for each point  **A:** any expression of each concept for each mark.  **A:** any other reasonable suggestions that would give 10000 for large decks and a much lower number for small decks. | |
| **7** | (a) | When two arguments are supplied they are used to set the ToolType and Kit respectively **[1]**… The CardNumber is set using the next available CardNumber from the class/static variable NextCardNumber by the parent constructor when it is called **[1]**… In the case of a third argument being supplied, the parent constructor is not called and the CardNumber is set by the value of the parameter **[1]**. | 3 marks | 1 mark for each point (MAX. 3) | |
| (b) | A constructor is used to set the value of any attributes appropriately and return an object which is an instance of the class **[1]** | 1 mark | **A:** to instantiate the class with the correct values. | |
| **8** | (a) | An abstract class is a (parent) class that you would never create an instance of **[1]**… An example of this is the Card class with an empty Process method **[1]**.  A concrete class is a class that you would create an instance of **[1]**… An example of this is the ToolCard or DifficultyCard class **[1]**. | 4 marks | For each type of class, there needs to be a description for the first mark with an example for the second mark.  For the concrete class, pretty much any class from Breakthrough onwards in the program (except Card) can be given as an example. | |
| **8** | (b) | A class or static variable has the same value for every object and is changed in them all when it changes in one **[1]**… An attribute may start off the same but has a different value in each object and if changed in one will not affect the others **[1]**. | 2 marks | 1 mark for each point | |
| **9** | (a) | Inheritance is where a child gains the attributes and behaviours/methods of its parent. | 1 mark | **A:** other words with similar meanings. | |
| (b) | Aggregation associate is where one class contains another class but their lifespans are not linked, they can function independently. | 1 mark | **R:** answers that do not refer to lifespan in some way. | |
| (c) | A data structure for which the memory usage will shrink and grow over time according to the storage needs. | 1 mark |  | |
| **10** | (a) | ***Solution #1***  Code **[1]:**  self.\_\_MoveCard(self.\_\_Deck, self.\_\_Discard, self.\_\_Deck.GetCardNumberAt(0))  … because CardCollection treats DifficultyCards and ToolCards **[1]**…as cards **[1]**…which is an example of polymorphism but when the statement resolves any methods will actually execute on the child’s object class **[1]**.  ***Solution #2***  Code **[1]:**  def Process(self, Deck, Discard, Hand, Sequence, CurrentLock, Choice, CardChoice):    ChoiceAsInteger = None  … because this overrides the Process method **[1]**… in the parent Card class **[1]**… which will mean that DifficultyCards can be treated as cards but behave as themselves **[1]**.  ***Solution #3***  Code **[1]:**  if self.\_\_Deck.GetCardDescriptionAt(0) == "Dif":  … because the call treats it as a card (morph to parent) **[1]**… but when the call resolves the attribute it was the one set in the child’s constructor for DifficultyCard **[1]**… which means that methods can now be called on it as a DifficultyCard (morph to child) **[1]**. | 4 marks | For any one possible answer: 1 mark for each point and 1 mark for the code  **A:** any example of code related to inheritance for 1 mark provided the explanation gains at least 1 mark.  **R:** code only with no explanation. | |
| (b) | Interacting with objects using the interface of the parent **[1]**… but having the object resolve the methods for themselves and execute up the inheritance structure from the child **[1]**. | 2 marks | 1 mark for each point  **A:** execute the method in the child. | |

| **Question** | | **Suggested Solution** | **Total Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- |
| **11** |  | Create a new AdvancedLock class **[1]**… that inherits from Lock **[1]**… and override the GetLockSolved method **[1]**… so that when the basic challenges are solved it unlocks the final challenge and returns False instead of True **[1]**… this will then mean that it can refer to a private attribute such as SecretUnlocked **[1]**… when checking GetLockSolved the next time **[1]**. | 6 marks | 1 mark for each point | |
| **12** |  | The ‘K a’ card is in position 2 meaning that when GetCardFromDeck is called, CardChoice will be 2 **[1]**… The card in position 2 will have already been moved from the hand to the deck as MoveCard is called by PlayCardToSequence before GetCardFromDeck **[1]**… The hand is redisplayed the player enters 3 to choose ‘K c’ as the key to discard because it is now in position 3 **[1]**… When Process is called CardChoice is 2 and Choice is 3 **[1]**… Choice is converted from a string to an integer, range checked (1‒5) successfully and stored in ChoiceAsInteger **[1]**… as 3 >= 2, ChoiceAsInteger is decremented by 1 **[1]**… and as ChoiceAsInteger is still > 0 it is decremented by 1 again to give 1 **[1]**… The selection statement checks against the card in the hand at index ChoiceAsInteger (which is 1) and that card is now ‘F b’ so the condition is False **[1]**… and 5 cards are discarded from the deck to the discard pile **[1]**. | 8 marks | 1 mark for each point (MAX. 8) | |
| **13** | (a) | Using a constant **[1]**… which would be declared once at the top of program and could be changed in that single place **[1]**. | 2 marks | 1 mark for each point | |
| (b) | It is possible to miss out values **[1]** ,… update the wrong values **[1]**,… or mistype values **[1]**… constants have an identifier, making a value easier to understand **[1]**. | 2 marks | 1 mark for each point (MAX. 2)  **A:** opposite points. | |
| **14** | (a) | File handling can always generate exceptions **[1]**… because files could be locked **[1]**… removed/unavailable/inaccessible **[1]** | 2 marks | 1 mark for each point (MAX. 2) | |
| (b) | Converting an inputted string to an integer **[1]**… because if it fails you want to catch the error and ask the user to input again **[1]**. | 2 marks | 1 mark for each point  **A:** any example of validation.  **A:** any reasonable example including custom exceptions. | |
| **15** |  | It is used as a condition for the inside loop because it needs to exit if the player runs out of cards and not just when a lock is solved **[1]**… for the outside loop it is based on the return value of CheckIfPlayerHasLost which also checks if the deck has 0 cards so this needs to exit the main game loop **[1]**. | 2 marks | 1 mark for each point  **A:** answers that don’t refer to CheckIfPlayerHasLost as long as they refer to where GameOver is set. | |