Past-paper walk-through

String Manipulation ▪ 1.15

eXercise

Questions in this set

- 2025 Q2
- 2023 Q2
- 2021 Q1
- 2018 Q3
- 2017 Q3
- 2016 Q2
- 2016 Q4
- 2014 Q1

Reading the mark scheme

- **B1** a mark that stands on its own – the point is either made or not.
- **M1** a *method* mark: the right approach, even if the number is wrong.
- **A1** an *answer* mark, and it usually needs its M mark first.
- **C1** a working mark on the way to the answer.
- // separates answers the examiner accepts equally.
- **ecf** = error carried forward: a later part is marked on your own earlier answer.
- **owtte** = "or words to that effect" – the wording need not match.

Question 2

2025 ▪ Q2

This question involves the `SignedText` class, which contains methods that are used to include a signature as part of a string of text. You will write the complete `SignedText` class, which contains a constructor and two methods.

The `SignedText` constructor takes two `String` parameters. The first parameter is a first name and the second parameter is a last name. The length of the second parameter is always greater than or equal to 1.

The `getSignature` method takes no parameters and returns a formatted signature string constructed from the first and last names according to the following rules.

- If the first name is an empty string, the returned signature string contains just the last name.
- If the first name is not an empty string, the returned signature string is the first letter of the first name, a dash ("-"), and the last name, in that order.

Question 2

The `addSignature` method returns a possibly revised copy of its `String` parameter. The parameter will contain at most one occurrence of the object's signature, at either the beginning or the end of the parameter. The returned string is created from the parameter according to the following rules.

- If the object's signature does not occur in the `String` parameter of the method, the returned `String` is the value of the parameter with the signature added to the end.
- If the object's signature occurs at the end of the `String` parameter, the returned `String` is the unchanged value of the parameter.
- If the object's signature occurs at the beginning of the `String` parameter, the returned `String` is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

Question 2

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `SignedText`.

Statement	Method Call Return Value (blank if none)	Explanation
`SignedText st1 = new SignedText("", "Wong");`		The `SignedText` object `st1` has an empty first name and last name "Wong".
`String temp = st1.getSignature();`	"Wong"	
`SignedText st2 = new SignedText("henri", "dubois");`		The `SignedText` object `st2` has first name "henri" and last name "dubois".
`temp = st2.getSignature();`	"h-dubois"	
`SignedText st3 = new SignedText("GRACE", "LOPEZ");`		The `SignedText` object `st3` has first name "GRACE" and last name "LOPEZ".
`temp = st3.getSignature();`	"G-LOPEZ"	
`SignedText st4 = new SignedText("", "FOX");`		The `SignedText` object `st4` has an empty first name and last name "FOX".
`String text = "Dear";`		
`temp = st4.addSignature(text);`	"DearFOX"	The signature does not occur in the `addSignature` parameter, so the returned string is the value of the parameter with the signature appended.
`text = "Best wishesFOX";`		
`temp = st4.addSignature(text);`	"Best wishesFOX"	The signature occurs at the end of the `addSignature` parameter, so the returned string is the unchanged value of the parameter.
`text = "FOXThanks";`		
`temp = st4.addSignature(text);`	"ThanksFOX"	The signature occurs at the beginning of the `addSignature` parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.
`text = "G-LOPEZHello";`		
`temp = st3.addSignature(text);`	"HelloG-LOPEZ"	The signature occurs at the beginning of the `addSignature` parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

Question 2

Write the complete `SignedText` class. Your implementation must meet all specifications and conform to the examples in the table.

Question 2

Statement	Method Call Return Value (blank if none)	Explanation
`SignedText st1 = new SignedText("", "Wong");`		The `SignedText` object `st1` has an empty first name and last name `"Wong"`.
`String temp = st1.getSignature();`	`"Wong"`	
`SignedText st2 = new SignedText("henri", "dubois");`		The `SignedText` object `st2` has first name `"henri"` and last name `"dubois"`.
`temp = st2.getSignature();`	`"h-dubois"`	
`SignedText st3 = new SignedText("GRACE", "LOPEZ");`		The `SignedText` object `st3` has first name `"GRACE"` and last name `"LOPEZ"`.
`temp = st3.getSignature();`	`"G-LOPEZ"`	
`SignedText st4 = new SignedText("", "FOX");`		The `SignedText` object `st4` has an empty first name and last name

Question 2

`SignedText("", "FOX");`		an empty first name and last name `"FOX"`.
`String text = "Dear";`		
`temp = st4.addSignature(text);`	`"DearFOX"`	The signature does not occur in the `addSignature` parameter, so the returned string is the value of the parameter with the signature appended.
`text = "Best wishesFOX";`		
`temp = st4.addSignature(text);`	`"Best wishesFOX"`	The signature occurs at the end of the `addSignature` parameter, so the returned string is the unchanged value of the parameter.

Question 2

Statement	Method Call Return Value (blank if none)	Explanation
`text = "FOXThanks";`		
`temp = st4.addSignature(text);`	`"ThanksFOX"`	The signature occurs at the beginning of the `addSignature` parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.
`text = "G-LOPEZHello";`		
`temp = st3.addSignature(text);`	`"HelloG-LOPEZ"`	The signature occurs at the beginning of the `addSignature` parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

Solution 2

- Canonical solution:
- “‘java
- public class SignedText
- {
- private String firstName;
- private String lastName;
- public SignedText(String first, String last)
- {

Solution 2

- firstName = first;
- lastName = last;
- }
- public String getSignature()
- {
- String sig = "";
- if (!firstName.equals(""))
- {
- sig += firstName.substring(0, 1) + "-";

Solution 2

- }
- sig += lastName;
- return sig;
- }
- public String addSignature(String textStr)
- {
- String sig = getSignature();
- int index = textStr.indexOf(sig);
- if (index == -1)

Solution 2

- {
- return textStr + sig;
- }
- else if (index == 0)
- {
- return textStr.substring(sig.length()) + sig;
- }
- else
- {

Solution 2

- return textStr;
- }
- }
- }
- “‘
- Scoring (9 points, 1 each): (1) declares class header ‘class SignedText‘; (2) declares appropriate private instance variable(s), constructor initializes them from its parameters; (3) declares constructor header ‘SignedText(String ____, String ____)‘; (4) declares method headers ‘public String getSignature()‘ and ‘public String addSignature(String ____)‘; (5) compares the first name to the empty string (or equivalent, e.g. length == 0); (6) determines the correct signature

Solution 2

string in both the empty- and non-empty-first-name cases (algorithm point) — signature is ‘lastName‘ alone, or ‘firstInitial + ”-” + lastName‘; (7) calls String methods (substring/indexOf/equals/etc.) with correct syntax throughout; (8) identifies the three required cases for ‘addSignature‘: no match (append signature), match at start (move signature to end), match elsewhere (leave unchanged); (9) ‘addSignature‘ returns the appropriate String in all three cases (algorithm point). An alternate canonical solution stores only the final signature string in one instance variable rather than firstName/lastName separately — either design earns full credit if the logic and String usage are correct.

Question 2

2023 ▪ Q2

This question involves methods that distribute text across lines of an electronic sign. The electronic sign and the text to be displayed on it are represented by the `Sign` class. You will write the complete `Sign` class, which contains a constructor and two methods. The `Sign` class constructor has two parameters. The first parameter is a `String` that contains the message to be displayed on the sign. The second parameter is an `int` that contains the *width* of each line of the sign. The width is the positive maximum number of characters that can be displayed on a single line of the sign.

Question 2

A sign contains as many lines as are necessary to display the entire message. The message is split among the lines of the sign without regard to spaces or punctuation. Only the last line of the sign may contain fewer characters than the width indicated by the constructor parameter.
The following are examples of a message displayed on signs of different widths. Assume that in each example, the sign is declared with the width specified in the first column of the table and with the message `"Everything on sale, please come in"`, which contains 34 characters.

Width of the Sign	**Sign Display**
15	`Everything on s / ale, please com / e in`
17	`Everything on sal / e, please come in`
40	`Everything on sale, please come in`

Question 2

In addition to the constructor, the `Sign` class contains two methods.
The `numberOfLines` method returns an `int` representing the number of lines needed to display the text on the sign. In the previous examples, `numberOfLines` would return 3, 2, and 1, respectively, for the sign widths shown in the table.
The `getLines` method returns a `String` containing the message broken into lines separated by semicolons (`;`) or returns `null` if the message is the empty string. The constructor parameter that contains the message to be displayed will not include any semicolons. As an example, in the first row of the preceding table, `getLines` would return `"Everything on s;ale, please com;e in"`. No semicolon should appear at the end of the `String` returned by `getLines`.

Question 2

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `Sign`.

Statement	Method Call Return Value (blank if none)	Explanation
`String str;`		
`int x;`		
`Sign sign1 = new Sign("ABC222DE", 3);`		The message for `sign1` contains 8 characters, and the sign has lines of width 3.
`x = sign1.numberOfLines();`	`3`	The sign needs three lines to display the 8-character message on a sign with lines of width 3.
`str = sign1.getLines();`	`"ABC;222;DE"`	Semicolons separate the text displayed on the first, second, and third lines of the sign.
`str = sign1.getLines();`	`"ABC;222;DE"`	Successive calls to `getLines` return the same value.
`Sign sign2 = new Sign("ABCD", 10);`		The message for `sign2` contains 4 characters, and the sign has lines of width 10.
`x = sign2.numberOfLines();`	`1`	The sign needs one line to display the 4-character message on a sign with lines of width 10.
`str = sign2.getLines();`	`"ABCD"`	No semicolon appears, since the text to be displayed fits on the first line of the sign.
`Sign sign3 = new Sign("ABCDEF", 6);`		The message for `sign3` contains 6 characters, and the sign has lines of width 6.
`x = sign3.numberOfLines();`	`1`	The sign needs one line to display the 6-character message on a sign with lines of width 6.
`str = sign3.getLines();`	`"ABCDEF"`	No semicolon appears, since the text to be displayed fits on the first line of the sign.
`Sign sign4 = new Sign("", 4);`		The message for `sign4` is an empty string.
`x = sign4.numberOfLines();`	`0`	There is no text to display.
`str = sign4.getLines();`	`null`	There is no text to display.
`Sign sign5 = new Sign("AB_CD_EF", 2);`		The message for `sign5` contains 8 characters, and the sign has lines of width 2.
`x = sign5.numberOfLines();`	`4`	The sign needs four lines to display the 8-character message on a sign with lines of width 2.
`str = sign5.getLines();`	`"AB;_C;D_;EF"`	Semicolons separate the text displayed on the four lines of the sign.

Question 2

Write the complete `Sign` class. Your implementation must meet all specifications and conform to the examples shown in the preceding table.

Question 2

Width of the Sign	Sign Display
15	`Everything on s ale, please com e in`
17	`Everything on sal e, please come in`
40	`Everything on sale, please come in`

Question 2

Statement	Method Call Return Value (blank if none)	Explanation
`String str;`		
`int x;`		
`Sign sign1 = new Sign("ABC222DE", 3);`		The message for `sign1` contains 8 characters, and the sign has lines of width 3

Question 2

		displayed fits on the first line of the sign.
`Sign sign4 = new Sign("", 4);`		The message for `sign4` is an empty string.
`x = sign4.numberOfLines();`	`0`	There is no text to display.
`str = sign4.getLines();`	`null`	There is no text to display.
`Sign sign5 = new`		The message for `sign5` contains 8 characters, and the sign has lines of width 2

Question 1

2021 ▪ Q1

This question involves the `WordMatch` class, which stores a secret string and provides methods that compare other strings to the secret string. You will write two methods in the `WordMatch` class.

```
k+kdpublic+w k+kdclass n+ncWordMatch
p
+w    c+cm/** The secret string. */
+w    k+kdprivate+w nString+w nsecretp;

+w    c+cm/** Constructs a WordMatch object with the given secret string of lowercase letters. */
+w    k+kdpublic+w n+nfWordMatchp(nString+w nwordp)
+w    p
+w        c+cm/* implementation not shown */
+w    p
+w    c+cm/** Returns a score for guess, as described in part (a).
c+cm     *  Precondition: 0  guess.length() = secret.length()
c+cm     */
+w    k+kdpublic+w k+ktint+w n+nfscoreGuessp(nString+w nguessp)
+w    p+w   c+cm/* to be implemented in part (a) */+w   p

+w    c+cm/** Returns the better of two guesses, as determined by scoreGuess and the rules for a
```

Question 1

(a) Write the `WordMatch` method `scoreGuess`. To determine the score to be returned, `scoreGuess` finds the number of times that `guess` occurs as a substring of `secret` and then multiplies that number by the square of the length of `guess`. Occurrences of `guess` may overlap within `secret`.
Assume that the length of `guess` is less than or equal to the length of `secret` and that `guess` is not an empty string.
The following examples show declarations of a `WordMatch` object. The tables show the outcomes of some possible calls to the `scoreGuess` method.

```
WordMatch game = new WordMatch("mississippi");
```

Value of guess	Number of Substring Occurrences	Score Calculation: (Number of Substring Occurrences) x (Square of the Length of guess)	Return Value of game.scoreGuess(guess)
"i"	4	4 * 1 * 1 = 4	4
"iss"	2	2 * 3 * 3 = 18	18
"issipp"	1	1 * 6 * 6 = 36	36
"mississippi"	1	1 * 11 * 11 = 121	121

Question 1

```
WordMatch game = new WordMatch("aaaabb");
```

Value of guess	Number of Substring Occurrences	Score Calculation: (Number of Substring Occurrences) x (Square of the Length of guess)	Return Value of game.scoreGuess(guess)
“a”	4	4 * 1 * 1 = 4	4
“aa”	3	3 * 2 * 2 = 12	12
“aaa”	2	2 * 3 * 3 = 18	18
“aabb”	1	1 * 4 * 4 = 16	16
“c”	0	0 * 1 * 1 = 0	0

Question 1

Complete the scoreGuess method.

```
c+cm/** Returns a score for guess, as described in part (a).
c+cm *  Precondition: 0  guess.length() = secret.length()
c+cm */
k+kdpublic+w k+ktint+w n+nfscoreGuessp(nString+w nguessp)
```

[Class information box for this question, repeated for reference:]

Question 1

```
k+kdpublic+w k+kdclass n+ncWordMatch

k+kdprivate+w nString+w nsecret

k+kdpublic+w n+nfWordMatchp(nString+w nwordp)
k+kdpublic+w k+ktint+w n+nfscoreGuessp(nString+w nguessp)
k+kdpublic+w nString+w n+nffindBetterGuessp(nString+w nguess1p,+w nString+w nguess
```

Question 1

2021 ▪ Q1

This question involves the `WordMatch` class, which stores a secret string and provides methods that compare other strings to the secret string. You will write two methods in the `WordMatch` class.

```
k+kdpublic+w k+kdclass n+ncWordMatch
p
+w    c+cm/** The secret string. */
+w    k+kdprivate+w nString+w nsecretp;

+w    c+cm/** Constructs a WordMatch object with the given secret string of lowercase letters. */
+w    k+kdpublic+w n+nfWordMatchp(nString+w nwordp)
+w    p
+w        c+cm/* implementation not shown */
+w    p
+w    c+cm/** Returns a score for guess, as described in part (a).
c+cm     *  Precondition: 0  guess.length() = secret.length()
c+cm     */
+w    k+kdpublic+w k+ktint+w n+nfscoreGuessp(nString+w nguessp)
+w    p+w    c+cm/* to be implemented in part (a) */+w    p
```

Question 1

(b) Write the `WordMatch` method `findBetterGuess`, which returns the better guess of its two String parameters, `guess1` and `guess2`. If the `scoreGuess` method returns different values for `guess1` and `guess2`, then the guess with the higher score is returned. If the `scoreGuess` method returns the same value for `guess1` and `guess2`, then the alphabetically greater guess is returned.
The following example shows a declaration of a `WordMatch` object and the outcomes of some possible calls to the `scoreGuess` and `findBetterGuess` methods.

```
WordMatch game = new WordMatch("concatenation");
```

Method Call	Return Value	Explanation
`game.scoreGuess("ten")`	9	1 * 3 * 3
`game.scoreGuess("nation")`	36	1 * 6 * 6
`game.findBetterGuess("ten", "nation")`	"nation"	Since `scoreGuess` returns 36 for "nation" and 9 for "ten", the guess with the greater score, "nation", is returned.
`game.scoreGuess("con")`	9	1 * 3 * 3
`game.scoreGuess("cat")`	9	1 * 3 * 3
`game.findBetterGuess("con", "cat")`	"con"	Since `scoreGuess` returns 9 for both "con" and "cat", the alphabetically greater guess, "con", is returned.

Question 1

Complete method `findBetterGuess`.
Assume that `scoreGuess` works as specified, regardless of what you wrote in part (a). You must use `scoreGuess` appropriately to receive full credit.

```
c+cm/** Returns the better of two guesses, as determined by scoreGuess and the rul
c+cm *  tiebreaker that are described in part (b).
c+cm *  Precondition: guess1 and guess2 contain all lowercase letters.
c+cm *                guess1 is not the same as guess2.
c+cm */
k+kdpublic+w nString+w n+nffindBetterGuessp(nString+w nguess1p,+w nString+w nguess
```

Question 1

[Class information box for this question, repeated for reference:]

```
k+kdpublic+w k+kdclass n+ncWordMatch

k+kdprivate+w nString+w nsecret

k+kdpublic+w n+nfWordMatchp(nString+w nwordp)
k+kdpublic+w k+ktint+w n+nfscoreGuessp(nString+w nguessp)
k+kdpublic+w nString+w n+nffindBetterGuessp(nString+w nguess1p,+w nString+w nguess
```

Solution 1

(a) Mark scheme

- Model solution (scoreGuess, 5 pts):
- “‘java
- public int scoreGuess(String guess)
- {
- int count = 0;
- for (int i = 0; i <= secret.length() - guess.length(); i++)
- {
- if (secret.substring(i, i + guess.length()).equals(guess))

Solution 1

- {
- count++;
- }
- }
- return count * guess.length() * guess.length();
- }
- ‘‘‘
- Counts every (possibly overlapping) occurrence of ‘guess‘ as a substring of ‘secret‘, then returns ‘count * guess.length()^2‘.
- Points (1 each, 5 total):

Solution 1

- 1. Compares ‘guess‘ to a substring of ‘secret‘ (calling ‘secret.indexOf(guess)‘ alone still earns this).
- 2. Uses a substring of ‘secret‘ with the correct length (== guess.length()) for the comparison.
- 3. Loops through all necessary substrings of ‘secret‘ with no bounds errors and without skipping overlapping occurrences.
- 4. Counts the number of identified occurrences of ‘guess‘ within ‘secret‘ (in the context of a condition involving both ‘secret‘ and ‘guess‘).
- 5. Calculates and returns the correct final score: must use a loop, compare ‘guess‘ to multiple substrings of ‘secret‘, not count the same matching substring more than once, and use the correct (unchanged) ‘guess‘ length when computing the score.

Solution 1

(b) Mark scheme

- Model solution (findBetterGuess, 4 pts):
- “‘java
- public String findBetterGuess(String guess1, String guess2)
- {
- if (scoreGuess(guess1) > scoreGuess(guess2))
- {
- return guess1;
- }

Solution 1

- if (scoreGuess(guess2) $>$ scoreGuess(guess1))
- {
- return guess2;
- }
- if (guess1.compareTo(guess2) $>$ 0)
- {
- return guess1;
- }
- return guess2;
- }

Solution 1

- “‘
- Returns the guess with the higher scoreGuess() value; if scores tie, returns the alphabetically greater guess (by String.compareTo).
- Points (1 each, 4 total):
- 6. Calls ‘scoreGuess‘ (with parameters, on ‘this‘) to get scores for ‘guess1‘ and ‘guess2‘.
- 7. Compares the two scores using the comparison result in a conditional statement (not just ‘==‘/‘!=‘).
- 8. Determines which of ‘guess1‘/‘guess2‘ is alphabetically greater (reversing the comparison direction is still acceptable; must not reimplement compareTo incorrectly or treat its result as a boolean).

Solution 1

- 9. Returns the correctly identified ‘guess1‘ or ‘guess2‘ in every case (must not reverse a comparison, omit either comparison, or fail to return a guess in some case).

Question 3

2018 ▪ Q3

3. The `StringChecker` interface describes classes that check if strings are valid, according to some criterion.

```
public interface StringChecker
{
    /** Returns true if str is valid. */
    boolean isValid(String str);
}
```

A `CodeWordChecker` is a `StringChecker`. A `CodeWordChecker` object can be constructed with three parameters: two integers and a string. The first two parameters specify the minimum and maximum code word lengths, respectively, and the third parameter specifies a string that must not occur in the code word. A `CodeWordChecker` object can also be constructed with a single parameter that specifies a string that must not occur in the code word; in this case the minimum and maximum lengths will default to 6 and 20, respectively.

The following examples illustrate the behavior of `CodeWordChecker` objects.

Example 1

Question 3

```
StringChecker sc1 = new CodeWordChecker(5, 8, "$");
```

Valid code words have 5 to 8 characters and must not include the string `"$"`.

Method call	Return value	Explanation
`sc1.isValid("happy")`	`true`	The code word is valid.
`sc1.isValid("happy$")`	`false`	The code word contains `"$"`.
`sc1.isValid("Code")`	`false`	The code word is too short.
`sc1.isValid("happyCode")`	`false`	The code word is too long.

Example 2

```
StringChecker sc2 = new CodeWordChecker("pass");
```

Question 3

Valid code words must not include the string `"pass"`. Because the bounds are not specified, the length bounds are 6 and 20, inclusive.

Method call	Return value	Explanation
`sc2.isValid("MyPass")`	`true`	The code word is valid.
`sc2.isValid("Mypassport")`	`false`	The code word contains `"pass"`.
`sc2.isValid("happy")`	`false`	The code word is too short.
`sc2.isValid("1,000,000,000,000,000")`	`false`	The code word is too long.

Write the complete `CodeWordChecker` class. Your implementation must meet all specifications and conform to all examples.

Solution 3

- Model solution:
- “‘java
- public class CodeWordChecker implements StringChecker
- {
- private int minLength;
- private int maxLength;
- private String notAllowed;
- public CodeWordChecker(int minLen, int maxLen, String symbol)

Solution 3

- {
- minLength = minLen;
- maxLength = maxLen;
- notAllowed = symbol;
- }
- public CodeWordChecker(String symbol)
- {
- minLength = 6;
- maxLength = 20;

Solution 3

- notAllowed = symbol;
- }
- public boolean isValid(String str)
- {
- return str.length() >= minLength && str.length() <= maxLength &&
- str.indexOf(notAllowed) == -1;
- }
- }
- ‘‘‘

Solution 3

- Holistic point allocation (9 pts total, single class-implementation question, no (a)/(b) split): +1 declares the class header 'public class CodeWordChecker implements StringChecker'; +1 declares all appropriate 'private' instance variables (min length, max length, unwanted/notAllowed string) — not 'static', not outside the class. Constructors (+3 total): +1 declares correct headers for both required constructors — a 3-parameter constructor 'public CodeWordChecker(int ___, int ___, String ___)' and a 1-parameter constructor 'public CodeWordChecker(String ___)'; +1 uses all three parameters to initialize the instance variables in the 3-parameter constructor; +1 uses the given parameter plus default values (min length 6, max length 20) to initialize the instance variables in the 1-parameter constructor. 'isValid' method (+4 total): +1 declares the header 'public boolean isValid(String ___)'; +1 checks that the string's length is between min and max,

Solution 3

inclusive; +1 checks that the string does not contain the unwanted/notAllowed substring; +1 returns ‘true‘ when the length is in range AND the unwanted string is absent, and ‘false‘ otherwise, based on correct use of the checks even if a prior check point wasn't earned. Question-specific penalty: -1 if the constructor is written to return a value.