Past-paper walk-through

Anatomy of a Class ▪ 3.3

eXercise

Questions in this set

- 2026 Q2
- 2025 Q2
- 2024 Q2
- 2023 Q2
- 2019 Q2
- 2018 Q3
- 2017 Q2
- 2016 Q1
- 2015 Q2
- 2015 Q4

Questions in this set

- 2014 Q2
- 2014 Q4

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

2026 ▪ Q2

The `Bottle` class, which you will write, represents a bottle that contains liquid. `Bottle` objects are created by calls to a constructor with a `double` parameter that represents the bottle's capacity, in milliliters (mL), which is the maximum amount of liquid in the bottle. Assume that this value will be greater than or equal to 0. When `Bottle` objects are constructed, each is filled to its capacity.

Question 2

The `Bottle` class contains an `updateAmount` method, which updates the amount of liquid in the bottle. This method has a `double` parameter that represents the amount of liquid, in mL, to be removed from the bottle, with a precondition that the parameter is always greater than zero and less than or equal to the amount of liquid remaining in the bottle. If updating the amount of liquid in the bottle would cause it to be less than 25% of the capacity, the bottle is filled and reset to the capacity. The `updateAmount` method returns a `double` that represents the amount remaining in the bottle, in mL, after the amount has been updated.
The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `Bottle`.

Statement	Return Value (blank if no value)	Explanation
`double amt;`		
`Bottle water = new Bottle(1000.0);`		A `Bottle` object named `water` is constructed with a capacity of 1,000 mL of liquid.
`amt = water.updateAmount(400.0);`	600.0	400 mL are used, so 600 mL remain in the bottle.
`amt = water.updateAmount(100.0);`	500.0	100 mL are used, so 500 mL remain in the bottle.
`amt = water.updateAmount(300.0);`	1000.0	300 mL are used, so 200 mL remain in the bottle. Since this amount is less than 250 mL (25% of the capacity of 1,000 mL), the bottle is immediately filled and reset to the initial amount of 1,000 mL.
`Bottle shampoo = new`		A `Bottle` object named `shampoo` is constructed with a capacity of 40 mL of liquid.

Question 2

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

Question 2

Statement	Return Value (blank if no value)	Explanation
`double amt;`		
`Bottle water = new Bottle(1000.0);`		A `Bottle` object named `water` is constructed with a capacity of 1,000 mL of liquid.
`amt = water.updateAmount(400.0);`	`600.0`	400 mL are used, so 600 mL remain in the bottle.
`amt = water.updateAmount(100.0);`	`500.0`	100 mL are used, so 500 mL remain in the bottle.
`amt = water.updateAmount(300.0);`	`1000.0`	300 mL are used, so 200 mL remain in the bottle. Since this amount is less than 250 mL (25% of the capacity of 1,000 mL), the bottle is immediately filled and reset to the initial amount of 1,000 mL.
`Bottle shampoo =`		A `Bottle` object named `shampoo` is

Question 2

`Bottle shampoo = new Bottle(40.0);`		A Bottle object named shampoo is constructed with a capacity of 40 mL of liquid.
`amt = shampoo.updateAmount(30.0);`	`10.0`	30 mL are used, so 10 mL remain in the bottle.
`amt = shampoo.updateAmount(1.0);`	`40.0`	1 mL is used, so 9 mL remain in the bottle. Since this amount is less than 10 mL (25% of the capacity of 40 mL), the bottle is immediately filled and reset to the initial amount of 40 mL.

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

2024 ▪ Q2

This question involves a scoreboard for a game. The game is played between two teams who alternate turns so that at any given time, one team is active and the other is inactive. During a turn, a team makes one or more plays. Each play can score one or more points and the team's turn continues, or the play can fail, in which case no points are scored and the team's turn ends. The `Scoreboard` class, which you will write, is used to keep track of the score in a game.

The `Scoreboard` class contains a constructor and two methods.

Question 2

- The constructor has two parameters. The first parameter is a `String` containing the name of team 1, and the second parameter is a `String` containing the name of team 2. The game always begins with team 1 as the active team.
- The `recordPlay` method has a single nonnegative integer parameter that is equal to the number of points scored on a play or `0` if the play failed. If the play results in one or more points scored, the active team's score is updated and that team remains active. If the value of the parameter is `0`, the active team's turn ends and the inactive team becomes the active team. The `recordPlay` method does not return a value.
- The `getScore` method has no parameters. The method returns a `String` containing information about the current state of the game. The returned string begins with the score of team 1, followed by a hyphen (`"-"`), followed by the score of team 2, followed by a hyphen, followed by the name of the team that is currently active.

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 `Scoreboard`.

Statement	Value Returned (blank if none)	Explanation
`String info;`		
`Scoreboard game =  new Scoreboard("Red", "Blue");`		game is a new Scoreboard for a game played between team 1, whose name is "Red", and team 2, whose name is "Blue". The active team is set to team 1.
`info = game.getScore();`	"0-0-Red"	
`game.recordPlay(1);`		Team 1 earns 1 point because the game always begins with team 1 as the active team.
`info = game.getScore();`	"1-0-Red"	
`game.recordPlay(0);`		Team 1's play failed, so team 2 is now active.
`info = game.getScore();`	"1-0-Blue"	
`info = game.getScore();`	"1-0-Blue"	The score and state of the game are unchanged since the last call to getScore.
`game.recordPlay(3);`		Team 2 earns 3 points.
`info = game.getScore();`	"1-3-Blue"	
`game.recordPlay(1);`		Team 2 earns 1 point.
`game.recordPlay(0);`		Team 2's play failed, so team 1 is now active.
`info = game.getScore();`	"1-4-Red"	
`game.recordPlay(0);`		Team 1's play failed, so team 2 is now active.
`game.recordPlay(4);`		Team 2 earns 4 points.
`game.recordPlay(0);`		Team 2's play failed, so team 1 is now active.
`info = game.getScore();`	"1-8-Red"	
`Scoreboard match =  new Scoreboard("Lions", "Tigers");`		match is a new and independent Scoreboard object.
`info = match.getScore();`	"0-0-Lions"	
`info = game.getScore();`	"1-8-Red"	

Question 2

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

Question 2

`info = game.getScore();`	`"1-0-Red"`	
`game.recordPlay(0);`		Team 1's play failed, so team 2 is now active.
`info = game.getScore();`	`"1-0-Blue"`	
`info = game.getScore();`	`"1-0-Blue"`	The score and state of the game are unchanged since the last call to `getScore`.
`game.recordPlay(3);`		Team 2 earns 3 points.
`info = game.getScore();`	`"1-3-Blue"`	
`game.recordPlay(1);`		Team 2 earns 1 point.
`game.recordPlay(0);`		Team 2's play failed, so team 1 is now active.
`info = game.getScore();`	`"1-4-Red"`	
`game.recordPlay(0);`		Team 1's play failed, so team 2 is now active.
`game.recordPlay(4);`		Team 2 earns 4 points.
`game.recordPlay(0);`		Team 2's play failed, so team 1 is now active.
`info = game.getScore();`	`"1-8-Red"`	
`Scoreboard match = new Scoreboard("Lions", "Tigers");`		`match` is a new and independent `Scoreboard` object.
`info = match.getScore();`	`"0-0-Lions"`	
`info = game.getScore();`	`"1-8-Red"`	

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 2

2019 ▪ Q2

This question involves the implementation of a fitness tracking system that is represented by the `StepTracker` class. A `StepTracker` object is created with a parameter that defines the minimum number of steps that must be taken for a day to be considered *active*.
The `StepTracker` class provides a constructor and the following methods.

- `addDailySteps`, which accumulates information about steps, in readings taken once per day
- `activeDays`, which returns the number of active days
- `averageSteps`, which returns the average number of steps per day, calculated by dividing the total number of steps taken by the number of days tracked

Question 2

The following table contains a sample code execution sequence and the corresponding results.

Statements and Expressions	Value Returned (blank if no value)	Comment
`StepTracker tr = new StepTracker(10000);`		Days with at least 10,000 steps are considered active. Assume that the parameter is positive.
`tr.activeDays();`	0	No data have been recorded yet.
`tr.averageSteps();`	0.0	When no step data have been recorded, the averageSteps method returns 0.0.
`tr.addDailySteps(9000);`		This is too few steps for the day to be considered active.
`tr.addDailySteps(5000);`		This is too few steps for the day to be considered active.
`tr.activeDays();`	0	No day had at least 10,000 steps.
`tr.averageSteps();`	7000.0	The average number of steps per day is (14000 / 2).
`tr.addDailySteps(13000);`		This represents an active day.
`tr.activeDays();`	1	Of the three days for which step data were entered, one day had at least 10,000 steps.
`tr.averageSteps();`	9000.0	The average number of steps per day is (27000 / 3).
`tr.addDailySteps(23000);`		This represents an active day.
`tr.addDailySteps(1111);`		This is too few steps for the day to be considered active.
`tr.activeDays();`	2	Of the five days for which step data were entered, two days had at least 10,000 steps.
`tr.averageSteps();`	10222.2	The average number of steps per day is (51111 / 5).

Question 2

Write the complete `StepTracker` class, including the constructor and any required instance variables and methods. Your implementation must meet all specifications and conform to the example.

Question 2

Statements and Expressions	Value Returned (blank if no value)	Comment
`StepTracker tr = new StepTracker(10000);`		Days with at least 10,000 steps are considered active. Assume that the parameter is positive.
`tr.activeDays();`	`0`	No data have been recorded yet.
`tr.averageSteps();`	`0.0`	When no step data have been recorded, the `averageSteps` method returns `0.0`.
`tr.addDailySteps(9000);`		This is too few steps for the day to be considered active.
`tr.addDailySteps(5000);`		This is too few steps for the day to be considered active.
`tr.activeDays();`	`0`	No day had at least 10,000 steps.
`tr.averageSteps();`	`7000.0`	The average number of steps per day is (14000 / 2).
`tr.addDailySteps(13000);`		This represents an active day.
`tr.activeDays();`	`1`	Of the three days for which step data were entered, one day had at least 10,000 steps.
`tr.averageSteps();`	`9000.0`	The average number of steps per day is (27000 / 3).
`tr.addDailySteps(23000);`		This represents an active day.
[illegible]		This is too few steps for the day to be considered

Question 2

`tr.addDailySteps(1111);`		This is too few steps for the day to be considered active.
`tr.activeDays();`	`2`	Of the five days for which step data were entered, two days had at least 10,000 steps.
`tr.averageSteps();`	`10222.2`	The average number of steps per day is (51111 / 5).

Solution 2

- Model solution for the full StepTracker class:
- public class StepTracker {
- private int minSteps;
- private int totalSteps;
- private int numDays;
- private int numActiveDays;
- public StepTracker(int threshold) {
- minSteps = threshold;

Solution 2

- totalSteps = 0;
- numDays = 0;
- numActiveDays = 0;
- }
- public void addDailySteps(int steps) {
- totalSteps += steps;
- numDays++;
- if (steps >= minSteps) { numActiveDays++; }
- }

Solution 2

- public int activeDays() {
- return numActiveDays;
- }
- public double averageSteps() {
- if (numDays == 0) { return 0.0; }
- else { return (double) totalSteps / numDays; }
- }
- }

Solution 2

- How the 9 points are earned: +1 declares all appropriate private instance variables (e.g. minSteps, totalSteps, numDays, numActiveDays – must be private and inside the class); Constructor (+2 total): +1 declares the header public StepTracker(int ____), +1 uses the parameter and appropriate values to initialize every instance variable (failing to initialize one, or assigning parameters to themselves instead of instance variables, forfeits this point); addDailySteps method (+3 total): +1 declares the header public void addDailySteps(int ____), +1 identifies active days (compares the parameter against the threshold) and increments an active-day count, +1 updates the other instance variables appropriately (e.g. running total and day count) on every call, not just active days; activeDays method (+1 total): declares and implements public int activeDays() returning the correct count, given correct updates elsewhere;

Solution 2

averageSteps method (+2 total): +1 declares the header public double averageSteps(), +1 returns the correctly calculated double average of steps per day, including handling the special case of zero tracked days (e.g. returning 0.0) without failing – using integer division instead of double division forfeits this point.

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.

Question 2

2017 ▪ Q2

This question involves the design of a class that will be used to produce practice problems. The following `StudyPractice` interface represents practice problems that can be used to study some subject.

```
k+kdpublic+w k+kdinterface n+ncStudyPractice
p
+w    c+cm/** Returns the current practice problem. */
+w    nString+w n+nfgetProblemp(p)p;

+w    c+cm/** Changes to the next practice problem. */
+w    k+ktvoid+w n+nfnextProblemp(p)p;
p
```

Question 2

The `MultPractice` class is a `StudyPractice` that produces multiplication practice problems. A `MultPractice` object is constructed with two integer values: *first integer* and *initial second integer*. The first integer is a value that remains constant and is used as the first integer in every practice problem. The initial second integer is used as the starting value for the second integer in the practice problems. This second value is incremented for each additional practice problem that is produced by the class.
For example, a `MultPractice` object created with the call `new MultPractice(7, 3)` would be used to create the practice problems `"7 TIMES 3"`, `"7 TIMES 4"`, `"7 TIMES 5"`, and so on.

Question 2

In the `MultPractice` class, the `getProblem` method returns a string in the format of `"first integer TIMES second integer"`. The `nextProblem` method updates the state of the `MultPractice` object to represent the next practice problem.
The following examples illustrate the behavior of the `MultPractice` class. Each table shows a code segment and the output that would be produced as the code is executed.
Example 1

Code segment	**Output produced**
`StudyPractice p1 = new MultPractice(7, 3); System.out.println(p1.getProblem());`	`7 TIMES 3`
`p1.nextProblem(); System.out.println(p1.getProblem());`	`7 TIMES 4`
`p1.nextProblem(); System.out.println(p1.getProblem());`	`7 TIMES 5`
`p1.nextProblem(); System.out.println(p1.getProblem());`	`7 TIMES 6`

Question 2

Example 2

Code segment	Output produced
`StudyPractice p2 = new MultPractice(4, 12); p2.nextProblem(); System.out.println(p2.getProblem()); p2.nextProblem(); p2.nextProblem(); System.out.println(p2.getProblem()); p2.nextProblem(); System.out.println(p2.getProblem());`	`4 TIMES 13 4 TIMES 14 4 TIMES 15 4 TIMES 16`

Interface information for this question

Question 2

```
k+kdpublic+w k+kdinterface n+ncStudyPractice

nString+w n+nfgetProblemp(p)
k+ktvoid+w n+nfnextProblemp(p)
```

Write the complete `MultPractice` class. Your implementation must be consistent with the specifications and the given examples.

Question 2

Code segment	Output produced
`StudyPractice p1 = new MultPractice(7, 3);` `System.out.println(p1.getProblem());`	`7 TIMES 3`
`p1.nextProblem();` `System.out.println(p1.getProblem());`	`7 TIMES 4`
`p1.nextProblem();` `System.out.println(p1.getProblem());`	`7 TIMES 5`
`p1.nextProblem();` `System.out.println(p1.getProblem());`	`7 TIMES 6`

Example 2

Code segment	Output produced

Question 2

Code segment	Output produced
`StudyPractice p2 = new MultPractice(4, 12);`	
`p2.nextProblem();`	
`System.out.println(p2.getProblem());`	`4 TIMES 13`
`System.out.println(p2.getProblem());`	`4 TIMES 13`
`p2.nextProblem();`	
`p2.nextProblem();`	
`System.out.println(p2.getProblem());`	`4 TIMES 15`
`p2.nextProblem();`	
`System.out.println(p2.getProblem());`	`4 TIMES 16`

Question 2 continues on page 10.

Question 2

Interface information for this question

```
public interface StudyPractice

String getProblem()
void nextProblem()
```

Solution 2

- MultPractice class implementing StudyPractice (9 pts total, holistic class design question). Model solution:
- “‘java
- public class MultPractice implements StudyPractice
- {
- private int first;
- private int second;
- public MultPractice(int num1, int num2)

Solution 2

- {
- first = num1;
- second = num2;
- }
- public String getProblem()
- {
- return first + " TIMES " + second;
- }
- public void nextProblem()

Solution 2

- {
- second++;
- }
- }
- “‘
- Point breakdown: +1 class header ‘public class MultPractice implements StudyPractice‘; +1 declares all necessary private instance variables (the two factors); +2 constructor — +1 header ‘public MultPractice(int ___, int ___)‘, +1 initializes all instance variables from the parameters; +3 getProblem method — +1 header ‘public String getProblem()‘, +1 builds a string using the current values of the instance variables, +1 returns the constructed string; +2

Solution 2

nextProblem method — +1 header ‘public void nextProblem()‘, +1 updates the instance variable(s) so the second factor increments each call. Grade holistically against these 9 criteria since student code may organize helper logic differently.

Question 1

2016 ▪ Q1

This question involves the implementation and extension of a `RandomStringChooser` class.

(a) A `RandomStringChooser` object is constructed from an array of non-`null` `String` values. When the object is first constructed, all of the strings are considered available. The `RandomStringChooser` class has a `getNext` method, which has the following behavior. A call to `getNext` returns a randomly chosen string from the available strings in the object. Once a particular string has been returned from a call to `getNext`, it is no longer available to be returned from subsequent calls to `getNext`. If no strings are available to be returned, `getNext` returns `"NONE"`.