Question 1: Methods and Control Structures — 9 points

Canonical solution

a. **4 points**

```
public int walkDogs(int hour)
{
   int dogsToWalk = company.numAvailableDogs(hour);

   if (dogsToWalk > maxDogs)
   {
      dogsToWalk = maxDogs;
   }

   company.updateDogs(hour, dogsToWalk);
   return dogsToWalk;
}
```

b. **5 points**

```
public int dogWalkShift(int startHour, int endHour)
{
   int totalPay = 0;

   for (int hour = startHour; hour <= endHour; hour++)
   {
      int dogs = walkDogs(hour);
      int hourPay = 5 * dogs;
      if (dogs == maxDogs || (hour >= 9 && hour <= 17))
      {
         hourPay += 3;
      }
      totalPay += hourPay;
   }
   return totalPay;
}
```


a. `walkDogs`

	Scoring Criteria	Decision Rules	
1	Calls `DogWalkCompany` method(s) on `company`	Responses **can** still earn the point even if they • fail to save or use the returned value • call `numAvailableDogs` more than once • only call one of the methods Responses **will not** earn the point if they • use the incorrect parameter types • call either `numAvailableDogs` or `updateDogs` on nothing or on something other than `company`	**1 point**
2	Compares available dogs and `maxDogs` to determine number of dogs to walk	Responses **can** still earn the point even if they • call `numAvailableDogs` incorrectly • calculate an incorrect number of dogs that were walked	**1 point**
3	Calls `numAvailableDogs` with `hour` and `updateDogs` with `hour` and correct number of dogs to walk (*algorithm*)	Responses **can** still earn the point even if they • call either method on nothing or on something other than `company` Responses **will not** earn the point if they • calculate an incorrect number of dogs that were walked	**1 point**
4	Returns calculated value	Responses **can** still earn the point even if they • calculate an incorrect number of dogs that were walked • call `numAvailableDogs` multiple times Responses **will not** earn the point if they • return something other than an `int` • fail to return a value in some cases • print a value in addition to or instead of returning it	**1 point**

b. `dogWalkShift`

	Scoring Criteria	Decision Rules	
5	Loops from `startHour` to `endHour`, inclusive	Responses **can** still earn the point even if they • return from the loop early, as long as bounds would otherwise be correct	**1 point**
6	Calls `walkDogs` with `int` parameter (*in the context of a loop*)	Responses **can** still earn the point even if they • fail to save or use the returned value Responses **will not** earn the point if they • use an incorrect parameter type on any call to `walkDogs` • call the method on the class or on an object other than `this` (use of `this` is optional)	**1 point**
7	Calculates base pay for an hour	Responses **can** still earn the point even if they • fail to include the calculation in the loop • call `walkDogs` incorrectly	**1 point**
8	Calculates possible bonus for an hour's pay	Responses **can** still earn the point even if they • call `walkDogs` multiple times inside the loop • fail to include the calculation in the loop Responses **will not** earn the point if they • count the bonus twice when both conditions are true • count the bonus only when both conditions are true • count the bonus when it has not been earned • calculate bonus incorrectly	**1 point**
9	Accumulates total pay (*algorithm*)	Responses **can** still earn the point even if they • calculate base pay or bonus incorrectly • fail to return total pay (*return not assessed in this part*) Responses **will not** earn the point if they • fail to initialize variable used for total pay • call `walkDogs` multiple times inside the loop • do not accumulate base and bonus in the context of a loop • return from the loop early	**1 point**
	Question-specific penalties		
	None		

Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares `"int G=99, g=0;"`, then uses `"while (G < 10)"` instead of `"while (g < 10)"`, the context does **not** allow for the reader to assume the use of the lower case variable.*

2025 Digital Decision Rules

- Some non-ASCII characters are not printing correctly in ONE, particularly extended punctuation. Certain kinds of double-quotes may display as â□□; a character that displays as ï¼□ may be a parenthesis, comma, or semicolon (or other punctuation). If the badly displayed character makes sense as one of those, evaluate the response accordingly.
- If there are missing closed-double-quotes or closed-parentheses, assume they are at the end of the line where they opened, immediately before the semicolon or curly bracket (if any).
- Assume an open/left curly bracket `{` immediately after any method header or class header that does not already have one.
- Assume an appropriate amount of closing brackets before any method header to close all open brackets from the previous method or constructor.
- If there are missing curly brackets, clear indentation can "convey intent". Evaluate the response accordingly.
- Inside a method with left-justified code, indentation cannot "convey intent", so missing curly brackets cannot be assumed. Without bracketing or indentation, only the first line of a `while` / `if` / `for` is controlled by the statement; with open curly bracket and no indentation cues, the entire remainder of the method is "inside" the statement.

No Penalty

- `:` instead of `;` and vice versa
- `,` instead of `;` and vice versa

Question 2: Class — 9 points

Canonical solution

9 points

```
public class SignedText
{
   private String firstName;
   private String lastName;

   public SignedText(String first, String last)
   {
      firstName = first;
      lastName = last;
   }

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

   public String addSignature(String textStr)
   {
      String sig = getSignature();
      int index = textStr.indexOf(sig);

      if (index == -1)
      {
         return textStr + sig;
      }
      else if (index == 0)
      {
         return textStr.substring(sig.length()) + sig;
      }
      else
      {
         return textStr;
      }
   }
}
```

`SignedText`

	Scoring Criteria	Decision Rules	
1	Declares class header: `class SignedText`	Responses **will not** earn the point if they • declare the class as `private` • include extraneous code outside the class • include `()` with or without parameters in the class header	**1 point**
2	Declares all appropriate `private` instance variable(s) and constructor initializes instance variable(s) using appropriate parameters	Responses **can** still earn the point even if they • create and store the signature using only one `String` instance variable Responses **will not** earn the point if they • declare any instance variables `static` • omit `private` in instance variable declaration • declare variables outside the class • declare an instance variable inside the constructor • fail to use either parameter • assign values to instance variables from an unknown source • assign initial value to local variable instead of instance variable, even if the local variables are declared as `private` • assign parameter(s) to variable(s) with incompatible data type • return a value from the constructor • define the constructor inside a method or define a method inside the constructor	**1 point**
3	Declares constructor header: `SignedText(String ___, String ___)`	Responses **will not** earn the point if they • declare the constructor as `private` or `static`	**1 point**

4	Declares method headers:
	`public String getSignature()`
	`public String addSignature(String ___)`
		Responses **will not** earn the point if they
		• omit `public` in either method header or declare either method as something other than `public`
		• use incorrect method name
		• use incorrect return or parameter type
			1 point

5	Compares first name to the empty string	Responses **can** still earn the point even if they • perform the comparison implicitly (e.g., by comparing the string's length to 0)	**1 point**

6	Determines appropriate signature string in both cases (*algorithm*)	Responses **can** still earn the point even if they • fail to return a value in some cases (*return from `getSignature` not assessed*) Responses **will not** earn the point if they • construct the correct string but return something else • define another method inside the signature method or vice versa	**1 point**
7	Calls `String` methods using correct syntax throughout the class	Responses **can** still earn the point even if they • call `substring` only one time • call `indexOf`, `contains`, `charAt`, or other appropriate methods Responses **will not** earn the point if they • only call `length` and/or `equals` without using other `String` methods • fail to call at least one `String` method which accesses elements of a string	**1 point**
8	Identifies the three required cases for `addSignature` using appropriate conditions	Responses **can** still earn the point even if they • return an incorrect string in one or more cases	**1 point**
9	`addSignature` returns appropriate `String` in all three cases (*algorithm*)	Responses **can** still earn the point even if they • identify one or more of the three cases incorrectly, as long as they are unambiguously no-match, match-start, and match-end • implement `getSignature` incorrectly Responses **will not** earn the point if they • call `getSignature` incorrectly, or incorrectly implement its functionality in `addSignature` • fail to identify three distinct cases • build correct strings but assign them to cases incorrectly • print the string instead of or in addition to returning it • define another method inside `addSignature` or vice versa	**1 point**

	Question-specific penalties
	None

Alternate canonical:

```
public class SignedText
{
   private String signature;

   public SignedText(String first, String last)
   {
      signature = last;
      if (first.length() > 0)
      {
         signature = first.substring(0, 1) + "-" + last;
      }
   }

   public String getSignature()
   {
      return signature;
   }

   public String addSignature(String textStr)
   {
      String result = textStr;
      int index = textStr.indexOf(signature);
      if (index < 0)
      {
         result += signature;
      }
      else if (index == 0)
      {
         result = result.substring(signature.length()) + signature;
      }
      return result;
   }
}
```


Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares "`int G=99, g=0;`", then uses "`while (G < 10)`" instead of "`while (g < 10)`", the context does **not** allow for the reader to assume the use of the lower case variable.*

2025 Digital Decision Rules

- Some non-ASCII characters are not printing correctly in ONE, particularly extended punctuation. Certain kinds of double-quotes may display as â□□; a character that displays as ï¼□ may be a parenthesis, comma, or semicolon (or other punctuation). If the badly displayed character makes sense as one of those, evaluate the response accordingly.
- If there are missing closed-double-quotes or closed-parentheses, assume they are at the end of the line where they opened, immediately before the semicolon or curly bracket (if any).
- Assume an open/left curly bracket `{` immediately after any method header or class header that does not already have one.
- Assume an appropriate amount of closing brackets before any method header to close all open brackets from the previous method or constructor.
- If there are missing curly brackets, clear indentation can "convey intent". Evaluate the response accordingly.
- Inside a method with left-justified code, indentation cannot "convey intent", so missing curly brackets cannot be assumed. Without bracketing or indentation, only the first line of a `while` / `if` / `for` is controlled by the statement; with open curly bracket and no indentation cues, the entire remainder of the method is "inside" the statement.

No Penalty

- `:` instead of `;` and vice versa
- `,` instead of `;` and vice versa

AP® Computer Science A 2016 Scoring Guidelines

2016 GENERAL SCORING GUIDELINES

Apply the question assessment rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

- v) Array/collection access confusion `([] get)`
- w) Extraneous code that causes side-effect (e.g., writing to output, failure to compile)
- x) Local variables used but none declared
- y) Destruction of persistent data (e.g., changing value referenced by parameter)
- z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators **(× • ÷ ≤ ≥ <> ≠)**
- `[]` vs. `()` vs. `<>`
- = instead of == and vice versa
- `length`/`size` confusion for array, String, List, or ArrayList; with or without ( )
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing ; where structure clearly conveys intent
- Missing { } where indentation clearly conveys intent
- Missing ( ) on parameter-less method or constructor invocations
- Missing ( ) around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context. For example,* "ArayList" *instead of* "ArrayList". *As a counter example, note that if the code declares* "Bug bug;"*, then uses* "Bug.move()" *instead of* "bug.move()"*, the context does **not** allow for the reader to assume the object instead of the class.*

2016 SCORING GUIDELINES

Question 1: Random String Chooser

Part (a)	`RandomStringChooser`	**7 points**

Intent: *Define implementation of class to choose a random string*

+1 Uses correct class, constructor, and method headers

+1 Declares appropriate `private` instance variable(s)

+1 Initializes all instance variable(s) (*point lost if parameter not used in any initialization*)

+4 Implements `getNext`

- **+1** Generates a random number in the proper range (*point lost for improper or missing cast*)
- **+1** Chooses a string from instance variable using generated random number
- **+1** Updates state appropriately (*point lost if constructor parameter is altered*)
- **+1** Returns chosen string or `"NONE"` as appropriate

Part (b)	`RandomLetterChooser`	**2 points**

Intent: *Define implementation of a constructor of a class that extends* `RandomStringChooser`

+1 `getSingleLetters(str)`

+1 `super(getSingleLetters(str));` (*point lost if not first statement in constructor*)

2016 CANONICAL SOLUTIONS

Question 1: Random String Chooser

Part (a):

```
public class RandomStringChooser
{
    private List<String> words;

    public RandomStringChooser(String[] wordArray)
    {
       words = new ArrayList<String>();

       for (String singleWord : wordArray)
       {
            words.add(singleWord);
       }
    }

    public String getNext()
    {
       if (words.size() > 0)
       {
            return words.remove((int)(Math.random() * words.size()));
       }
       return "NONE";
    }
}
```

Part (b):

```
public RandomLetterChooser(String str)
{
     super(getSingleLetters(str));
}
```


2016 SCORING GUIDELINES

Question 2: Log Messages

Part (a) `LogMessage` constructor — **2 points**

Intent: *Initialize instance variables using passed parameter*

+1 Locates colon

+1 Initializes instance variables with correct parts of the parameter

Part (b) `containsWord` — **2 points**

Intent: *Determine whether description properly contains a keyword*

+1 Identifies at least one properly-contained occurrence of `keyword` in `description`

+1 Returns `true` if and only if `description` properly contains `keyword`
Returns `false` otherwise (*no bounds errors*)

Part (c) `removeMessages` — **5 points**

Intent: *Remove log messages containing keyword from system log list and return these messages in a new list*

+1 Accesses all items in `messageList` (*no bounds errors; point lost if no removal attempted*)

+1 Identifies keyword-containing entry using `containsWord`

+1 Adds all and only identified entries to new list (*point lost if original order not maintained*)

+1 Removes all identified entries from `messageList` (*point lost if* `messageList` *reordered)*

+1 Constructs and returns `new ArrayList<LogMessage>`

2016 CANONICAL SOLUTIONS

Question 2: Log Messages

Part (a):

```
public LogMessage(String message)
{
      int colon = message.indexOf(":");
      machineId = message.substring(0, colon);
      description = message.substring(colon + 1);
}
```

Part (b):

```
public boolean containsWord(String keyword)
{
   if (description.equals(keyword))
   {   return true;   }
   if (description.indexOf(keyword + " ") == 0)
   {   return true;   }
   if (description.indexOf(" " + keyword + " ") != -1)
   {   return true;   }
   if (description.length() > keyword.length())
   {
       if ((description.substring(description.length() -
                                  keyword.length() - 1).equals(
                                  " " + keyword)))
       {
          return true;
       }
   }
   return false;
}
```

Part (c):

```
public List<LogMessage> removeMessages(String keyword)
{
      List<LogMessage> removals = new ArrayList<LogMessage>();

      for (int i = 0; i < messageList.size(); i++)
      {
            if (messageList.get(i).containsWord(keyword))
            {
                  removals.add(messageList.remove(i));
                  i--;
            }
      }
      return removals;
}
```


2016 SCORING GUIDELINES

Question 3: Crossword

Part (a) `toBeLabeled` **3 points**

Intent: *Return a `boolean` value indicating whether a crossword grid square should be labeled with a positive number*

+1 Checks `blackSquares[r][c]`

+1 Checks for black square/border above and black square/border to the left (*no bounds errors*)

+1 Returns `true` if square should be labeled with positive number; returns `false` otherwise

Part (b) `Crossword` constructor **6 points**

Intent: *Initialize each square in a crossword puzzle grid to have a color (`boolean`) and an integer label*

+1 `puzzle = new Square[blackSquares.length][blackSquares[0].length];` (*or equivalent*)

+1 Accesses all locations in `puzzle` (*no bounds errors*)

+1 Calls `toBeLabeled` with appropriate parameters

+1 Creates and assigns new `Square` to location in `puzzle`

+1 Numbers identified squares consecutively, in row-major order, starting at 1

+1 On exit: All squares in `puzzle` have correct color and number (*minor errors covered in previous points ok*)

Question-Specific Penalties

-2 (p) Consistently uses incorrect name instead of `puzzle`

-1 (q) Uses *array*`[].length` instead of *array*`[`*num*`].length`

2016 CANONICAL SOLUTIONS

Question 3: Crossword

Part (a):

```
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
{
      return (!(blackSquares[r][c]) &&
             (r == 0 || c == 0 || blackSquares[r - 1][c] ||
              blackSquares[r][c - 1]));
}
```

Part (b):

```
public Crossword(boolean[][] blackSquares)
{
      puzzle = new Square[blackSquares.length][blackSquares[0].length];
      int num = 1;

      for (int r = 0; r < blackSquares.length; r++)
      {
            for (int c = 0; c < blackSquares[0].length; c++)
            {
                  if (blackSquares[r][c])
                  {
                        puzzle[r][c] = new Square(true, 0);
                  }
                  else
                  {
                        if (toBeLabeled(r, c, blackSquares))
                        {
                              puzzle[r][c] = new Square(false, num);
                              num++;
                        }
                        else
                        {
                              puzzle[r][c] = new Square(false, 0);
                        }
                  }
            }
      }
}
```


2016 SCORING GUIDELINES

Question 4: String Formatter

Part (a)	`totalLetters`	**2 points**

Intent: *Calculate the total number of letters in a list of words*

+1 Accesses all strings in `wordList` and adds length of each to accumulated count (*no bounds errors*)

+1 Initializes and returns accumulated count

Part (b)	`basicGapWidth`	**2 points**

Intent: *Calculate the minimum number of spaces (basic gap width) to be placed between each word in the formatted string*

+1 Calls `totalLetters` correctly and uses result

+1 Returns correct calculated value

Part (c)	`format`	**5 points**

Intent: *Return a formatted string consisting of words from `wordList` separated by one or more spaces*

+1 Calls `basicGapWidth` and `leftoverSpaces` correctly and uses results

+1 Adds all strings in `wordList` to formatted string in original order (*no bounds errors*)

+1 Inserts *basicGapWidth* spaces between each pair of words in formatted string

+1 Inserts one space between first *leftoverSpaces* pairs of words in formatted string

+1 Initializes and returns formatted string (*no extra or deleted characters*)

2016 CANONICAL SOLUTIONS

Question 4: String Formatter

Part (a):

```
public static int totalLetters(List<String> wordList)
{
      int total = 0;

      for (String word : wordList)
      {
            total += word.length();
      }
      return total;
}
```

Part (b):

```
public static int basicGapWidth(List<String> wordList, int formattedLen)
{
      return (formattedLen - totalLetters(wordList)) / (wordList.size()-1);
}
```

Part (c):

```
public static String format(List<String> wordList, int formattedLen)
{
      String formatted = "";
      int gapWidth = basicGapWidth(wordList, formattedLen);
      int leftovers = leftoverSpaces(wordList, formattedLen);

      for (int w = 0; w < wordList.size() - 1; w++)
      {
            formatted = formatted + wordList.get(w);
            for (int i = 0; i < gapWidth; i++)
            {
                  formatted = formatted + " ";
            }
            if (leftovers > 0)
            {
                  formatted = formatted + " ";
                  leftovers--;
            }
      }
      formatted = formatted + wordList.get(wordList.size() - 1);

      return formatted;
}
```