---

1. This question involves dog walkers who are paid through a dog-walking company to take dogs for one-hour walks. A dog-walking company has a varying number of dogs that need to be walked during each hour of the day. A dog-walking company is represented by the following `DogWalkCompany` class.

```
public class DogWalkCompany
{
  /**
   * Returns the number of dogs, always greater than 0, that are available
   * for a walk during the time specified by hour
   * Precondition: 0 <= hour <= 23
   */
  public int numAvailableDogs(int hour)
  { /* implementation not shown */ }

  /**
   * Decreases, by numberDogsWalked, the number of dogs available for a walk
   * during the time specified by hour
   * Preconditions: 0 <= hour <= 23
   *                numberDogsWalked > 0
   */
  public void updateDogs(int hour, int numberDogsWalked)
  { /* implementation not shown */ }

  /* There may be instance variables, constructors,
     and methods that are not shown. */
}
```

A dog walker is associated with a dog-walking company and is represented by the `DogWalker` class. You will write two methods of the `DogWalker` class.

```
public class DogWalker
{
  /** The maximum number of dogs this walker can walk simultaneously
      per hour */
  private int maxDogs;

  /** The dog-walking company this dog walker is associated with */
  private DogWalkCompany company;

  /**
   * Assigns max to maxDogs and comp to company
   * Precondition: max > 0
   */
  public DogWalker(int max, DogWalkCompany comp)
  { /* implementation not shown */ }

  /**
   * Takes at least one dog for a walk during the time specified by
   * hour, as described in part (a)
   * Preconditions: 0 <= hour <= 23
   *                maxDogs > 0
   */
  public int walkDogs(int hour)
  { /* to be implemented in part (a) */ }
```

```
  /**
   * Performs an entire dog-walking shift and returns the amount
   * earned, in dollars, as described in part (b)
   * Preconditions: 0 <= startHour <= endHour <= 23
   *                maxDogs > 0
   */
  public int dogWalkShift(int startHour, int endHour)
  { /* to be implemented in part (b) */ }

  /* There may be instance variables, constructors,
     and methods that are not shown. */
}
```

A. Write the `walkDogs` method, which updates and returns the number of dogs this dog walker walks during the time specified by `hour`. Values of `hour` range from `0` to `23`, inclusive.

A helper method, `numAvailableDogs`, has been provided in the `DogWalkCompany` class. The method returns the number of dogs available to be taken for a walk at a given hour. The dog walker will always walk as many dogs as the dog-walking company has available to walk, as long as the available number of dogs is not greater than the maximum number of dogs that the dog walker can handle, represented by the value of `maxDogs`.

Another helper method, `updateDogs`, has also been provided in the `DogWalkCompany` class. So that multiple dog walkers do not sign up to walk the same dogs, the `walkDogs` method should use `updateDogs` to update the dog-walking company with the number of dogs this dog walker will walk during the given hour. The parameters of the `updateDogs` method indicate how many dogs this dog walker will walk during the time specified by `hour`.

For example, if the dog-walking company has 10 dogs that need to be walked at the given hour but the dog walker's maximum is 4, the `updateDogs` method should be used to indicate that this dog walker will walk 4 of the 10 dogs during the given hour. As another example, if the dog-walking company has 3 dogs that need to be walked at the given hour and the dog walker's maximum is 4, the `updateDogs` method should be used to indicate that this dog walker will walk all 3 of the available dogs during the given hour.

The `walkDogs` method should return the number of dogs to be walked by this dog walker during the time specified by `hour`.

Complete method `walkDogs`. You must use `numAvailableDogs` and `updateDogs` appropriately to receive full credit.

```
/**
 * Takes at least one dog for a walk during the time specified by
 * hour, as described in part (a)
 * Preconditions: 0 <= hour <= 23
 *                maxDogs > 0
 */
public int walkDogs(int hour)
```

B. Write the `dogWalkShift` method, which performs a dog-walking shift of the hours in the range `startHour` to `endHour`, inclusive, and returns the total amount earned. For example, a dog-walking shift from `14` to `16` is composed of three one-hour dog walks starting at hours `14`, `15`, and `16`.

For each hour, the base pay is $5 per dog walked plus a bonus of $3 if at least one of the following is true.

- `maxDogs` dogs are walked
- the walk occurs between the peak hours of 9 and 17, inclusive

The following table shows an example of the calculated amount earned by walking dogs from hour 7 through hour 10, inclusive.

Hour	Maximum Number of Dogs	Number of Dogs Walked	Amount Earned, in Dollars
7	3	3	3 × 5 + 3 = 18
8	3	2	2 × 5 = 10
9	3	2	2 × 5 + 3 = 13
10	3	3	3 × 5 + 3 = 18
Total			59

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

```
/**
 * Performs an entire dog-walking shift and returns the amount
 * earned, in dollars, as described in part (b)
 * Preconditions: 0 <= startHour <= endHour <= 23
 *                maxDogs > 0
 */
public int dogWalkShift(int startHour, int endHour)
```

2. 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.

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 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