

1. This question involves the `Account` class, which is used to represent user accounts for a website. The `Account` class contains a helper method, `isAvailable`, which determines whether a username is available. You will write a constructor and a method in the `Account` class.

```
public class Account
{
    private String username; // To be initialized in part (a)

    /**
     * Creates a username based on the parameter requestedName. If the
     * username is unavailable, appends successive integers, beginning
     * with 1, to requestedName until an available username is found,
     * as described in part (a).
     */
    public Account(String requestedName)
    { /* to be implemented in part (a) */ }

    /**
     * Returns true if the parameter str is an available username;
     * returns false otherwise.
     */
    public static boolean isAvailable(String str)
    { /* implementation not shown */ }

    /**
     * Returns a shortened version of username with each hyphen ("-")
     * and the character before it removed, as described in part (b)
     * Preconditions: username does not start or end with a hyphen.
     *                  username does not contain consecutive hyphens.
     *                  username.length() >= 2
     * Postcondition: username is unchanged.
     */
    public String getShortenedName()
    { /* to be implemented in part (b) */ }

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

The following information applies to part A.

In part A, you will write the `Account` constructor, which creates a username based on the parameter `requestedName`.

A helper method, `isAvailable`, has been provided to determine whether a username is available. The `isAvailable` method returns `true` if its parameter is an available username and returns `false` otherwise.

If `requestedName` is an available username, the `Account` constructor will assign `requestedName` to the instance variable `username`. If not, the constructor will try different variations of `requestedName` until an available username is found and assigned to the instance variable `username`. The variations consist of `requestedName` followed by an integer, as in the following example.

- If `requestedName` is "Luis-Cruz" and this username is not available, then the constructor will check the availability of "Luis-Cruz1", "Luis-Cruz2", "Luis-Cruz3", and so on, until an available username is found. The first available username found is assigned to `username`.
- If `requestedName` is "PSmith" and this username is available, then "PSmith" is assigned to `username`.

The following information applies to part B.

In part B, you will write the `getShortenedName` method, which returns a shortened version of the instance variable `username` with each hyphen ("-") and the character immediately before it removed. If no hyphens appear in `username`, the value of `username` is returned.

For example, if `username` is "Amy-Marie-Lin", `getShortenedName` should return "AmMariLin".

As another example, if `username` is "SammyB3", `getShortenedName` should return "SammyB3".

Part A

Complete the `Account` constructor. You must use `isAvailable` appropriately in order to receive full credit.

```
/**
 * Creates a username based on the parameter requestedName. If the
 * username is unavailable, appends successive integers, beginning
 * with 1, to requestedName until an available username is found,
 * as described in part (a).
 */
public Account(String requestedName)
```

Part B

Complete method `getShortenedName`.

```
/**
 * Returns a shortened version of username with each hyphen ("-")
 * and the character before it removed, as described in part (b)
 * Preconditions: username does not start or end with a hyphen.
 *                username does not contain consecutive hyphens.
 *                username.length() >= 2
 * Postcondition: username is unchanged.
 */
public String getShortenedName()
```

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	$3 \times 5 + 3 = 18$
8	3	2	$2 \times 5 = 10$
9	3	2	$2 \times 5 + 3 = 13$
10	3	3	$3 \times 5 + 3 = 18$
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)
```

1. This question simulates birds or possibly a bear eating at a bird feeder. The following `Feeder` class contains information about how much food is in the bird feeder and simulates how much food is eaten. You will write two methods of the `Feeder` class.

```
public class Feeder
{
    /**
     * The amount of food, in grams, currently in the bird feeder; initialized in the constructor and
     * always greater than or equal to zero
     */
    private int currentFood;

    /**
     * Simulates one day with numBirds birds or possibly a bear at the bird feeder,
     * as described in part (a)
     * Precondition: numBirds > 0
     */
    public void simulateOneDay(int numBirds)
    { /* to be implemented in part (a) */ }

    /**
     * Returns the number of days birds or a bear found food to eat at the feeder in this simulation,
     * as described in part (b)
     * Preconditions: numBirds > 0, numDays > 0
     */
    public int simulateManyDays(int numBirds, int numDays)
    { /* to be implemented in part (b) */ }

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

GO ON TO THE NEXT PAGE.

- (a) Write the `simulateOneDay` method, which simulates `numBirds` birds or possibly a bear at the feeder for one day. The method determines the amount of food taken from the feeder on this day and updates the `currentFood` instance variable. The simulation accounts for normal conditions, which occur 95% of the time, and abnormal conditions, which occur 5% of the time.

Under normal conditions, the simulation assumes that on any given day, only birds visit the feeder and that each bird at the feeder consumes the same amount of food. This standard amount consumed is between 10 and 50 grams of food, inclusive, in 1-gram increments. That is, on any given day, each bird might eat 10, 11, . . . , 49, or 50 grams of food. The amount of food eaten by each bird on a given day is randomly generated and each integer from 10 to 50, inclusive, has an equal chance of being chosen.

For example, a run of the simulation might predict that for a certain day under normal conditions, each bird coming to the feeder will eat 11 grams of food. If 10 birds come to the feeder on that day, then a total of 110 grams of food will be consumed.

If the simulated food consumed is greater than the amount of food in the feeder, the birds empty the feeder and the amount of food in the feeder at the end of the day is zero.

Under abnormal conditions, a bear empties the feeder and the amount of food in the feeder at the end of the day is zero.

The following examples show possible results of three calls to `simulateOneDay`.

- Example 1: If the feeder initially contains 500 grams of food, the call `simulateOneDay(12)` could result in 12 birds eating 20 grams of food each, leaving 260 grams of food in the feeder.
- Example 2: If the feeder initially contains 1,000 grams of food, the call `simulateOneDay(22)` could result in a bear eating all the food, leaving 0 grams of food in the feeder.
- Example 3: If the feeder initially contains 100 grams of food, the call `simulateOneDay(5)` could result in 5 birds attempting to eat 30 grams of food each. Since the feeder initially contains less than 150 grams of food, the feeder is emptied, leaving 0 grams of food in the feeder.

GO ON TO THE NEXT PAGE.

Complete the `simulateOneDay` method.

```
/**
 * Simulates one day with numBirds birds or possibly a bear at the bird feeder,
 * as described in part (a)
 * Precondition: numBirds > 0
 */
public void simulateOneDay(int numBirds)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Feeder
private int currentFood
public void simulateOneDay(int numBirds)
public int simulateManyDays(int numBirds, int numDays)
```

GO ON TO THE NEXT PAGE.

- (b) Write the `simulateManyDays` method. The method uses `simulateOneDay` to simulate `numBirds` birds or a bear coming to the feeder on at most `numDays` consecutive days. The simulation returns the number of days that birds or a bear found food at the feeder.

Consider the following examples.

Value of <code>currentFood</code> and Method Call	Possible Outcomes and Resulting Return Value
<code>currentFood: 2400</code> <code>simulateManyDays(10, 4)</code>	Day 1: <code>simulateOneDay</code> leaves 2100 grams of food in the feeder. Day 2: <code>simulateOneDay</code> leaves 1650 grams of food in the feeder. Day 3: <code>simulateOneDay</code> leaves 1500 grams of food in the feeder. Day 4: <code>simulateOneDay</code> leaves 1260 grams of food in the feeder. The simulation returns 4 because, on all four days of the simulation, birds or a bear found food at the feeder. The instance variable <code>currentFood</code> has the value 1260.
<code>currentFood: 250</code> <code>simulateManyDays(10, 5)</code>	Day 1: <code>simulateOneDay</code> leaves 150 grams of food in the feeder. Day 2: <code>simulateOneDay</code> leaves 0 grams of food in the feeder. The simulation returns 2 because, on two of the five simulated days, birds or a bear found food at the feeder. The instance variable <code>currentFood</code> has the value 0.
<code>currentFood: 0</code> <code>simulateManyDays(5, 10)</code>	The simulation returns 0 because no food was found at the feeder on any day. The instance variable <code>currentFood</code> has the value 0.

GO ON TO THE NEXT PAGE.

Complete the `simulateManyDays` method. Assume that `simulateOneDay` works as intended, regardless of what you wrote in part (a). You must use `simulateOneDay` appropriately in order to receive full credit.

```
/**
 * Returns the number of days birds or a bear found food to eat at the feeder in this simulation,
 * as described in part (b)
 * Preconditions: numBirds > 0, numDays > 0
 */
public int simulateManyDays(int numBirds, int numDays)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Feeder
private int currentFood
public void simulateOneDay(int numBirds)
public int simulateManyDays(int numBirds, int numDays)
```

1. This question involves the `AppointmentBook` class, which provides methods for students to schedule appointments with their teacher. Appointments can be scheduled during one of eight class periods during the school day, numbered 1 through 8. A requested appointment has a duration, which is the number of minutes the appointment will last. The 60 minutes within a period are numbered 0 through 59. In order for an appointment to be scheduled, the teacher must have a block of consecutive, available minutes that contains at least the requested number of minutes in a requested period. Scheduled appointments must start and end within the same period.

The `AppointmentBook` class contains two helper methods, `isMinuteFree` and `reserveBlock`. You will write two additional methods of the `AppointmentBook` class.

```
public class AppointmentBook
{
    /**
     * Returns true if minute in period is available for an appointment and returns
     * false otherwise
     * Preconditions: 1 <= period <= 8; 0 <= minute <= 59
     */
    private boolean isMinuteFree(int period, int minute)
    { /* implementation not shown */ }

    /**
     * Marks the block of minutes that starts at startMinute in period and
     * is duration minutes long as reserved for an appointment
     * Preconditions: 1 <= period <= 8; 0 <= startMinute <= 59;
     * 1 <= duration <= 60
     */
    private void reserveBlock(int period, int startMinute, int duration)
    { /* implementation not shown */ }

    /**
     * Searches for the first block of duration free minutes during period, as described in
     * part (a). Returns the first minute in the block if such a block is found or returns -1 if no
     * such block is found.
     * Preconditions: 1 <= period <= 8; 1 <= duration <= 60
     */
    public int findFreeBlock(int period, int duration)
    { /* to be implemented in part (a) */ }

    /**
     * Searches periods from startPeriod to endPeriod, inclusive, for a block
     * of duration free minutes, as described in part (b). If such a block is found,
     * calls reserveBlock to reserve the block of minutes and returns true; otherwise
     * returns false.
     * Preconditions: 1 <= startPeriod <= endPeriod <= 8; 1 <= duration <= 60
     */
    public boolean makeAppointment(int startPeriod, int endPeriod,
                                   int duration)
    { /* to be implemented in part (b) */ }

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

- (a) Write the `findFreeBlock` method, which searches `period` for the first block of free minutes that is `duration` minutes long. If such a block is found, `findFreeBlock` returns the first minute in the block. Otherwise, `findFreeBlock` returns `-1`. The `findFreeBlock` method uses the helper method `isMinuteFree`, which returns `true` if a particular minute is available to be included in a new appointment and returns `false` if the minute is unavailable.

Consider the following list of unavailable and available minutes in period 2.

Minutes in Period 2	Available?
0–9 (10 minutes)	No
10–14 (5 minutes)	Yes
15–29 (15 minutes)	No
30–44 (15 minutes)	Yes
45–49 (5 minutes)	No
50–59 (10 minutes)	Yes

The method call `findFreeBlock(2, 15)` would return 30 to indicate that a 15-minute block starting with minute 30 is available. No steps should be taken as a result of the call to `findFreeBlock` to mark those 15 minutes as unavailable.

The method call `findFreeBlock(2, 9)` would also return 30. Whenever there are multiple blocks that satisfy the requirement, the earliest starting minute is returned.

The method call `findFreeBlock(2, 20)` would return `-1`, since no 20-minute block of available minutes exists in period 2.

Complete method `findFreeBlock`. You must use `isMinuteFree` appropriately in order to receive full credit.

```
/**
 * Searches for the first block of duration free minutes during period, as described in
 * part (a). Returns the first minute in the block if such a block is found or returns -1 if no
 * such block is found.
 * Preconditions: 1 <= period <= 8; 1 <= duration <= 60
 */
public int findFreeBlock(int period, int duration)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

- (b) Write the `makeAppointment` method, which searches the periods from `startPeriod` to `endPeriod`, inclusive, for the earliest block of `duration` available minutes in the lowest-numbered period. If such a block is found, the `makeAppointment` method calls the helper method `reserveBlock` to mark the minutes in the block as unavailable and returns `true`. If no such block is found, the `makeAppointment` method returns `false`.

Consider the following list of unavailable and available minutes in periods 2, 3, and 4 and three successive calls to `makeAppointment`.

Period	Minutes	Available?
2	0–24 (25 minutes)	No
2	25–29 (5 minutes)	Yes
2	30–59 (30 minutes)	No
3	0–14 (15 minutes)	Yes
3	15–40 (26 minutes)	No
3	41–59 (19 minutes)	Yes
4	0–4 (5 minutes)	No
4	5–29 (25 minutes)	Yes
4	30–43 (14 minutes)	No
4	44–59 (16 minutes)	Yes

The method call `makeAppointment(2, 4, 22)` returns `true` and results in the minutes 5 through 26, inclusive, in period 4 being marked as unavailable.

The method call `makeAppointment(3, 4, 3)` returns `true` and results in the minutes 0 through 2, inclusive, in period 3 being marked as unavailable.

The method call `makeAppointment(2, 4, 30)` returns `false`, since there is no block of 30 available minutes in periods 2, 3, or 4.

The following shows the updated list of unavailable and available minutes in periods 2, 3, and 4 after the three example method calls are complete.

Period	Minutes	Available?
2	0–24 (25 minutes)	No
2	25–29 (5 minutes)	Yes
2	30–59 (30 minutes)	No
3	0–2 (3 minutes)	No
3	3–14 (12 minutes)	Yes
3	15–40 (26 minutes)	No
3	41–59 (19 minutes)	Yes
4	0–26 (27 minutes)	No
4	27–29 (3 minutes)	Yes
4	30–43 (14 minutes)	No
4	44–59 (16 minutes)	Yes

Complete method `makeAppointment`. Assume that `findFreeBlock` works as intended, regardless of what you wrote in part (a). You must use `findFreeBlock` and `reserveBlock` appropriately in order to receive full credit.

```
/**
 * Searches periods from startPeriod to endPeriod, inclusive, for a block
 * of duration free minutes, as described in part (b). If such a block is found,
 * calls reserveBlock to reserve the block of minutes and returns true; otherwise
 * returns false.
 * Preconditions: 1 <= startPeriod <= endPeriod <= 8; 1 <= duration <= 60
 */
public boolean makeAppointment(int startPeriod, int endPeriod,
                               int duration)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class AppointmentBook
private boolean isMinuteFree(int period, int minute)
private void reserveBlock(int period, int startMinute,
                           int duration)
public int findFreeBlock(int period, int duration)
public boolean makeAppointment(int startPeriod, int endPeriod,
                               int duration)
```

1. This question involves simulation of the play and scoring of a single-player video game. In the game, a player attempts to complete three levels. A level in the game is represented by the `Level` class.

```
public class Level
{
    /** Returns true if the player reached the goal on this level and returns false otherwise */
    public boolean goalReached()
    { /* implementation not shown */ }

    /** Returns the number of points (a positive integer) recorded for this level */
    public int getPoints()
    { /* implementation not shown */ }

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

GO ON TO THE NEXT PAGE.

Play of the game is represented by the `Game` class. You will write two methods of the `Game` class.

```
public class Game
{
    private Level levelOne;
    private Level levelTwo;
    private Level levelThree;

    /** Postcondition: All instance variables have been initialized. */
    public Game()
    { /* implementation not shown */ }

    /** Returns true if this game is a bonus game and returns false otherwise */
    public boolean isBonus()
    { /* implementation not shown */ }

    /** Simulates the play of this Game (consisting of three levels) and updates all relevant
     * game data
     */
    public void play()
    { /* implementation not shown */ }

    /** Returns the score earned in the most recently played game, as described in part (a) */
    public int getScore()
    { /* to be implemented in part (a) */ }

    /** Simulates the play of num games and returns the highest score earned, as
     * described in part (b)
     * Precondition: num > 0
     */
    public int playManyTimes(int num)
    { /* to be implemented in part (b) */ }

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

GO ON TO THE NEXT PAGE.

- (a) Write the `getScore` method, which returns the score for the most recently played game. Each game consists of three levels. The score for the game is computed using the following helper methods.
- The `isBonus` method of the `Game` class returns `true` if this is a bonus game and returns `false` otherwise.
 - The `goalReached` method of the `Level` class returns `true` if the goal has been reached on a particular level and returns `false` otherwise.
 - The `getPoints` method of the `Level` class returns the number of points recorded on a particular level. Whether or not recorded points are earned (included in the game score) depends on the rules of the game, which follow.

The score for the game is computed according to the following rules.

- Level one points are earned only if the level one goal is reached. Level two points are earned only if both the level one and level two goals are reached. Level three points are earned only if the goals of all three levels are reached.
- The score for the game is the sum of the points earned for levels one, two, and three.
- If the game is a bonus game, the score for the game is tripled.

GO ON TO THE NEXT PAGE.

The following table shows some examples of game score calculations.

	Level One Results	Level Two Results	Level Three Results	isBonus Return Value	Score Calculation
goalReached Return Value: getPoints Return Value:	true 200	true 100	true 500	true	$(200 + 100 + 500) \times 3 = 2,400$ The recorded points for levels one, two, and three are earned because the goals were reached in all three levels. The earned points are multiplied by 3 because isBonus returns true.
goalReached Return Value: getPoints Return Value:	true 200	true 100	false 500	false	$200 + 100 = 300$ The recorded points for level one and level two are earned because the goal was reached in levels one and two. The recorded points for level three are not earned because the goal was not reached in level three.
goalReached Return Value: getPoints Return Value:	true 200	false 100	true 500	true	$200 \times 3 = 600$ The recorded points for only level one are earned because the goal was not reached in level two. The earned points are multiplied by 3 because isBonus returns true.
goalReached Return Value: getPoints Return Value:	false 200	true 100	true 500	false	0 Because the goal in level one was not reached, no points are earned for any level.

Complete the getScore method.

```
/** Returns the score earned in the most recently played game, as described in part (a) */
public int getScore()
```

Begin your response at the top of a new page in the Free Response booklet and fill in the appropriate circle indicating the question number.
If there are multiple parts to this question, write the part letter with your response.

GO ON TO THE NEXT PAGE.

- (b) Write the `playManyTimes` method, which simulates the play of `num` games and returns the highest game score earned. For example, if the four plays of the game that are simulated as a result of the method call `playManyTimes(4)` earn scores of 75, 50, 90, and 20, then the method should return 90.

Play of the game is simulated by calling the helper method `play`. Note that if `play` is called only one time followed by multiple consecutive calls to `getScore`, each call to `getScore` will return the score earned in the single simulated play of the game.

Complete the `playManyTimes` method. Assume that `getScore` works as intended, regardless of what you wrote in part (a). You must call `play` and `getScore` appropriately in order to receive full credit.

```
/** Simulates the play of num games and returns the highest score earned, as
 * described in part (b)
 * Precondition: num > 0
 */
public int playManyTimes(int num)
```

Begin your response at the top of a new page in the Free Response booklet and fill in the appropriate circle indicating the question number.
If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class Level
{
    public boolean goalReached()
    public int getPoints()
}

public class Game
{
    private Level levelOne
    private Level levelTwo
    private Level levelThree

    public Game()
    public boolean isBonus()
    public void play()
    public int getScore()
    public int playManyTimes(int num)
```

GO ON TO THE NEXT PAGE.

1. The `APCalendar` class contains methods used to calculate information about a calendar. You will write two methods of the class.

```
public class APCalendar
{
    /** Returns true if year is a leap year and false otherwise. */
    private static boolean isLeapYear(int year)
    { /* implementation not shown */ }

    /** Returns the number of leap years between year1 and year2, inclusive.
     * Precondition: 0 <= year1 <= year2
     */
    public static int numberOfLeapYears(int year1, int year2)
    { /* to be implemented in part (a) */ }

    /** Returns the value representing the day of the week for the first day of year,
     * where 0 denotes Sunday, 1 denotes Monday, ..., and 6 denotes Saturday.
     */
    private static int firstDayOfYear(int year)
    { /* implementation not shown */ }

    /** Returns n, where month, day, and year specify the nth day of the year.
     * Returns 1 for January 1 (month = 1, day = 1) of any year.
     * Precondition: The date represented by month, day, year is a valid date.
     */
    private static int dayOfYear(int month, int day, int year)
    { /* implementation not shown */ }

    /** Returns the value representing the day of the week for the given date
     * (month, day, year), where 0 denotes Sunday, 1 denotes Monday, ...,
     * and 6 denotes Saturday.
     * Precondition: The date represented by month, day, year is a valid date.
     */
    public static int dayOfWeek(int month, int day, int year)
    { /* to be implemented in part (b) */ }

    // There may be instance variables, constructors, and other methods not shown.
}
```

- (a) Write the static method `numberOfLeapYears`, which returns the number of leap years between `year1` and `year2`, inclusive.

In order to calculate this value, a helper method is provided for you.

- `isLeapYear(year)` returns `true` if `year` is a leap year and `false` otherwise.

Complete method `numberOfLeapYears` below. You must use `isLeapYear` appropriately to receive full credit.

```
/** Returns the number of leap years between year1 and year2, inclusive.
 * Precondition: 0 <= year1 <= year2
 */
public static int numberOfLeapYears(int year1, int year2)
```

- (b) Write the static method `dayOfWeek`, which returns the integer value representing the day of the week for the given date (`month`, `day`, `year`), where 0 denotes Sunday, 1 denotes Monday, ..., and 6 denotes Saturday. For example, 2019 began on a Tuesday, and January 5 is the fifth day of 2019. As a result, January 5, 2019, fell on a Saturday, and the method call `dayOfWeek(1, 5, 2019)` returns 6.

As another example, January 10 is the tenth day of 2019. As a result, January 10, 2019, fell on a Thursday, and the method call `dayOfWeek(1, 10, 2019)` returns 4.

In order to calculate this value, two helper methods are provided for you.

- `firstDayOfYear(year)` returns the integer value representing the day of the week for the first day of year, where 0 denotes Sunday, 1 denotes Monday, ..., and 6 denotes Saturday. For example, since 2019 began on a Tuesday, `firstDayOfYear(2019)` returns 2.
- `dayOfYear(month, day, year)` returns n , where `month`, `day`, and `year` specify the n th day of the year. For the first day of the year, January 1 (`month` = 1, `day` = 1), the value 1 is returned. This method accounts for whether `year` is a leap year. For example, `dayOfYear(3, 1, 2017)` returns 60, since 2017 is not a leap year, while `dayOfYear(3, 1, 2016)` returns 61, since 2016 is a leap year.

Class information for this question

```
public class APCalendar
```

```
private static boolean isLeapYear(int year)
public static int numberOfLeapYears(int year1, int year2)
private static int firstDayOfYear(int year)
private static int dayOfYear(int month, int day, int year)
public static int dayOfWeek(int month, int day, int year)
```

Complete method `dayOfWeek` below. You must use `firstDayOfYear` and `dayOfYear` appropriately to receive full credit.

```
/** Returns the value representing the day of the week for the given date
 * (month, day, year), where 0 denotes Sunday, 1 denotes Monday, ...,
 * and 6 denotes Saturday.
 * Precondition: The date represented by month, day, year is a valid date.
 */
public static int dayOfWeek(int month, int day, int year)
```

1. This question involves reasoning about a simulation of a frog hopping in a straight line. The frog attempts to hop to a goal within a specified number of hops. The simulation is encapsulated in the following `FrogSimulation` class. You will write two of the methods in this class.

```
public class FrogSimulation
{
    /** Distance, in inches, from the starting position to the goal. */
    private int goalDistance;

    /** Maximum number of hops allowed to reach the goal. */
    private int maxHops;

    /** Constructs a FrogSimulation where dist is the distance, in inches, from the starting
     * position to the goal, and numHops is the maximum number of hops allowed to reach the goal.
     * Precondition: dist > 0; numHops > 0
     */
    public FrogSimulation(int dist, int numHops)
    {
        goalDistance = dist;
        maxHops = numHops;
    }

    /** Returns an integer representing the distance, in inches, to be moved when the frog hops.
     */
    private int hopDistance()
    {
        /* implementation not shown */
    }

    /** Simulates a frog attempting to reach the goal as described in part (a).
     * Returns true if the frog successfully reached or passed the goal during the simulation;
     * false otherwise.
     */
    public boolean simulate()
    {
        /* to be implemented in part (a) */
    }

    /** Runs num simulations and returns the proportion of simulations in which the frog
     * successfully reached or passed the goal.
     * Precondition: num > 0
     */
    public double runSimulations(int num)
    {
        /* to be implemented in part (b) */
    }
}
```

- (a) Write the `simulate` method, which simulates the frog attempting to hop in a straight line to a goal from the frog's starting position of 0 within a maximum number of hops. The method returns `true` if the frog successfully reached the goal within the maximum number of hops; otherwise, the method returns `false`.

The `FrogSimulation` class provides a method called `hopDistance` that returns an integer representing the distance (positive or negative) to be moved when the frog hops. A positive distance represents a move toward the goal. A negative distance represents a move away from the goal. The returned distance may vary from call to call. Each time the frog hops, its position is adjusted by the value returned by a call to the `hopDistance` method.

The frog hops until one of the following conditions becomes true:

- The frog has reached or passed the goal.
- The frog has reached a negative position.
- The frog has taken the maximum number of hops without reaching the goal.

The following example shows a declaration of a `FrogSimulation` object for which the goal distance is 24 inches and the maximum number of hops is 5. The table shows some possible outcomes of calling the `simulate` method.

```
FrogSimulation sim = new FrogSimulation(24, 5);
```

	Values returned by <code>hopDistance()</code>	Final position of frog	Return value of <code>sim.simulate()</code>
Example 1	5, 7, -2, 8, 6	24	<code>true</code>
Example 2	6, 7, 6, 6	25	<code>true</code>
Example 3	6, -6, 31	31	<code>true</code>
Example 4	4, 2, -8	-2	<code>false</code>
Example 5	5, 4, 2, 4, 3	18	<code>false</code>

Class information for this question

```
public class FrogSimulation
{
    private int goalDistance
    private int maxHops

    private int hopDistance()
    public boolean simulate()
    public double runSimulations(int num)
}
```

Complete method `simulate` below. You must use `hopDistance` appropriately to receive full credit.

```
/** Simulates a frog attempting to reach the goal as described in part (a).
 * Returns true if the frog successfully reached or passed the goal during the simulation.
```

- (b) Write the `runSimulations` method, which performs a given number of simulations and returns the proportion of simulations in which the frog successfully reached or passed the goal. For example, if the parameter passed to `runSimulations` is 400, and 100 of the 400 `simulate` method calls returned `true`, then the `runSimulations` method should return 0.25.

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

```
/** Runs num simulations and returns the proportion of simulations in which the frog
 * successfully reached or passed the goal.
 * @param num the number of simulations to run
 * @return the proportion of simulations in which the frog successfully reached or passed the goal
```

4. This question involves the process of taking a list of words, called `wordList`, and producing a formatted string of a specified length. The list `wordList` contains at least two words, consisting of letters only.

When the formatted string is constructed, spaces are placed in the gaps between words so that as many spaces as possible are evenly distributed to each gap. The equal number of spaces inserted into each gap is referred to as the *basic gap width*. Any *leftover spaces* are inserted one at a time into the gaps from left to right until there are no more leftover spaces.

The following three examples illustrate these concepts. In each example, the list of words is to be placed into a formatted string of length 20.

Example 1: `wordList`: ["AP", "COMP", "SCI", "ROCKS"]

Total number of letters in words: 14

Number of gaps between words: 3

Basic gap width: 2

Leftover spaces: 0

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																					
	A		P						C		O		M		P						S		C		I						R		O		C		K		S	

Example 2: `wordList`: ["GREEN", "EGGS", "AND", "HAM"]

Total number of letters in words: 15

Number of gaps between words: 3

Basic gap width: 1

Leftover spaces: 2

The leftover spaces are inserted one at a time between the words from left to right until there are no more leftover spaces. In this example, the first two gaps get an extra space.

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																					
	G		R		E		E		N						E		G		G		S						A		N		D				H		A		M	

Example 3: `wordList`: ["BEACH", "BALL"]

Total number of letters in words: 9

Number of gaps between words: 1

Basic gap width: 11

Leftover spaces: 0

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																											
	B		E		A		C		H																														B		A		L		L	

You will implement three `static` methods in a class named `StringFormatter` that is not shown.

- (a) Write the `StringFormatter` method `totalLetters`, which returns the total number of letters in the words in its parameter `wordList`. For example, if the variable `List<String> words` is `["A", "frog", "is"]`, then the call `StringFormatter.totalLetters(words)` returns 7. You may assume that all words in `wordList` consist of one or more letters.

Complete method `totalLetters` below.

```
/** Returns the total number of letters in wordList.  
 * Precondition: wordList contains at least two words, consisting of letters only.  
 */  
public static int totalLetters(List<String> wordList)
```

Part (b) begins on page 20.

- (b) Write the `StringFormatter` method `basicGapWidth`, which returns the basic gap width as defined earlier.

Class information for this question

```
public class StringFormatter

public static int totalLetters(List<String> wordList)
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
public static String format(List<String> wordList, int formattedLen)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

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

Complete method `basicGapWidth` below.

```
/** Returns the basic gap width when wordList is used to produce
 * a formatted string of formattedLen characters.
 * Precondition: wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 */
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
```

Part (c) begins on page 22.

- (c) Write the `StringFormatter` method `format`, which returns the formatted string as defined earlier. The `StringFormatter` class also contains a method called `leftoverSpaces`, which has already been implemented. This method returns the number of leftover spaces as defined earlier and is shown below.

```
/** Returns the number of leftover spaces when wordList is used to produce
 * a formatted string of formattedLen characters.
 * Precondition: wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 */
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
{ /* implementation not shown */ }
```

Class information for this question

```
public class StringFormatter

public static int totalLetters(List<String> wordList)
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
public static String format(List<String> wordList, int formattedLen)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `basicGapWidth` works as specified, regardless of what you wrote in part (b). You must use `basicGapWidth` and `leftoverSpaces` appropriately to receive full credit.

Complete method `format` below.

```
/** Returns a formatted string consisting of the words in wordList separated by spaces.
 * Precondition: The wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 * Postcondition: All words in wordList appear in the formatted string.
 * - The words appear in the same order as in wordList.
 * - The number of spaces between words is determined by basicGapWidth and the
 * distribution of leftoverSpaces from left to right, as described in the question.
 */
public static String format(List<String> wordList, int formattedLen)
```

STOP

END OF EXAM