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.

GO ON TO THE NEXT PAGE.

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.
}
```

GO ON TO THE NEXT PAGE.

(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.

GO ON TO THE NEXT PAGE.

(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

GO ON TO THE NEXT PAGE.

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)
```

GO ON TO THE NEXT PAGE.

Question 1: Methods and Control Structures — 9 points

Canonical solution

(a) — **5 points**

```
public int findFreeBlock(int period, int duration)
{
   int blockLength = 0;

   for (int minute = 0; minute < 60; minute++)
   {
      if (isMinuteFree(period, minute))
      {
         blockLength++;
         if (blockLength == duration)
         {
            return minute - blockLength + 1;
         }
      }
      else
      {
         blockLength = 0;
      }
   }
   return -1;
}
```

(b) — **4 points**

```
public boolean makeAppointment(int startPeriod,
                               int endPeriod,
                               int duration)
{
   for (int period = startPeriod;
        period <= endPeriod;
        period++)
   {
      int minute = findFreeBlock(period, duration);
      if (minute != -1)
      {
         reserveBlock(period, minute, duration);
         return true;
      }
   }
   return false;
}
```

(a) `findFreeBlock`

	Scoring Criteria	Decision Rules	
1	Loops over necessary minutes in an hour	Responses **can** still earn the point even if they • loop over fewer than 60 minutes as long as at least (60 – duration + 1) minutes are included • loop over 60 minutes and use a `boolean` to indicate that a free block has been found	**1 point**
2	Calls `isMinuteFree` with `period` and another `int` parameter	Responses **can** still earn the point even if they • call `isMinuteFree` with invalid parameters due to incorrect loop bounds Responses **will not** earn the point if they • use incorrect parameter types • order the parameters incorrectly • call the method on the class or on an object other than `this` (use of `this` is optional)	**1 point**
3	Keeps track of contiguous free minutes in a block (*algorithm*)	Responses **can** still earn the point even if they • call `isMinuteFree` incorrectly Responses **will not** earn the point if they • fail to reset when a nonfree minute is found • call `isMinuteFree` with minutes `>= 60`	**1 point**
4	Checks whether a valid block of `duration` minutes has been found	Responses **can** still earn the point even if they • maintain a `boolean` instead of accumulating the block length	**1 point**
5	Calculates and returns starting minute and `-1` appropriately based on identified block (*algorithm*)	Responses **will not** earn the point if they • are off by one on the returned value	**1 point**
		Total for part (a)	**5 points**

(b) `makeAppointment`

	Scoring Criteria	Decision Rules	
6	Loops over periods from `startPeriod` through `endPeriod` (*no bounds errors*)		**1 point**
7	Calls `findFreeBlock` and `reserveBlock` with correct number of `int` parameters, representing a period and minute as appropriate, and `duration`	Responses **can** still earn the point even if they • use incorrect parameter values Responses **will not** earn the point if they • use incorrect parameter types • order the parameters incorrectly • call the methods on the class or on an object other than `this` (use of `this` is optional)	**1 point**
8	Guards call to method to reserve a block by determining that starting minute is not `-1`		**1 point**
9	Books correct appointment and returns appropriate `boolean` (*algorithm*)	Responses **can** still earn the point even if they • have incorrect bounds in the loop • call `findFreeBlock` or `reserveBlock` incorrectly Responses **will not** earn the point if they • fail to return `true` or `false` • return before the call to `reserveBlock`	**1 point**
			4 points

Question-specific penalties

None

Total for question 1 **9 points**

Alternate Canonical for Part (a)

```
public int findFreeBlock(int period, int duration)
{
   for (int startMin = 0; startMin < 60 - duration + 1; startMin++)
   {
      boolean isBlockFree = true;
      for (int min = 0; min < duration; min++)
      {
         if (!isMinuteFree(period, min + startMin))
         {
            isBlockFree = false;
         }
      }
      if (isBlockFree)
      {
         return startMin;
      }
   }
   return -1;
}
```

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.*

		• return an incorrect number of lines for the message, as long as the number returned is exactly the number of lines produced by `getLines` • use a method name inconsistent with the examples, as long as it is recognizably equivalent Responses **will not** earn the point if they • incorrectly account for the final line	
7	`getLines` returns `null` appropriately	Responses **can** still earn the point even if they • identify `null` case in a method other than `getLines` • use an invalid call to `length` or `==` in guard for `null` return • use a method name inconsistent with the examples, as long as it is recognizably equivalent Responses **will not** earn the point if they • guard the return with incorrect logic	**1 point**
8	Calls `substring` and `length` (or equivalent) on `String` objects	Responses **can** still earn the point even if they • calculate `substring` parameter values incorrectly • call `substring` and/or `length` from a method other than `getLines` • use a method name inconsistent with the examples, as long as it is recognizably equivalent Responses **will not** earn the point if they • fail to call `substring` or `length` on `String` objects • call `substring` or `length` with an incorrect number of parameters, with a parameter of an incorrect type, or with incorrectly ordered parameters, anywhere in the class	**1 point**
9	`getLines` constructs the delimited sign output appropriately (*algorithm*)	Responses **can** still earn the point even if they • call `substring` and/or `length` incorrectly • fail to return the constructed `String` *(return not assessed)* • handle the empty string /`null` case incorrectly • construct the output in the constructor • use a method name inconsistent with the examples, as long as it is recognizably equivalent	**1 point**