A trip consists of a sequence of flights and is represented by the `Trip` class. The `Trip` class contains an `ArrayList` of `Flight` objects that are stored in chronological order. You may assume that for each flight after the first flight in the list, the departure time of the flight is later than the arrival time of the preceding flight in the list. A partial declaration of the `Trip` class is shown below. You will write two methods for the `Trip` class.

```
public class Trip
{
  private ArrayList<Flight> flights;
     // stores the flights (if any) in chronological order

  /** @return the number of minutes from the departure of the first flight to the arrival
   *          of the last flight if there are one or more flights in the trip;
   *          0, if there are no flights in the trip
   */
  public int getDuration()
  {  /* to be implemented in part (a) */  }

  /** Precondition: the departure time for each flight is later than the arrival time of its
   *                 preceding flight
   *  @return the smallest number of minutes between the arrival of a flight and the departure
   *          of the flight immediately after it, if there are two or more flights in the trip;
   *          -1, if there are fewer than two flights in the trip
   */
  public int getShortestLayover()
  {  /* 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) Complete method `getDuration` below.

```
/** @return  the number of minutes from the departure of the first flight to the arrival
 *           of the last flight if there are one or more flights in the trip;
 *           0, if there are no flights in the trip
 */
public int getDuration()
```

(b) Write the `Trip` method `getShortestLayover`. A layover is the number of minutes from the arrival of one flight in a trip to the departure of the flight immediately after it. If there are two or more flights in the trip, the method should return the shortest layover of the trip; otherwise, it should return -1.

For example, assume that the instance variable `flights` of a `Trip` object `vacation` contains the following flight information.

	Departure Time	**Arrival Time**	**Layover (minutes)**
Flight 0	11:30 a.m.	12:15 p.m.	
			} 60
Flight 1	1:15 p.m.	3:45 p.m.	
			} 15
Flight 2	4:00 p.m.	6:45 p.m.	
			} 210
Flight 3	10:15 p.m.	11:00 p.m.	

The call `vacation.getShortestLayover()` should return 15.

Complete method `getShortestLayover` below.

```
/** Precondition: the departure time for each flight is later than the arrival time of its
 *                preceding flight
 *  @return  the smallest number of minutes between the arrival of a flight and the departure
 *           of the flight immediately after it, if there are two or more flights in the trip;
 *           -1, if there are fewer than two flights in the trip
 */
public int getShortestLayover()
```

GO ON TO THE NEXT PAGE.

2. Consider a method of encoding and decoding words that is based on a *master string*. This master string will contain all the letters of the alphabet, some possibly more than once. An example of a master string is `"sixtyzipperswerequicklypickedfromthewovenjutebag"`. This string and its indexes are shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	i	x	t	y	z	i	p	p	e	r	s	w	e	r	e	q	u	i	c	k	l	y	p

24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
i	c	k	e	d	f	r	o	m	t	h	e	w	o	v	e	n	j	u	t	e	b	a	g

An encoded string is defined by a list of *string parts*. A string part is defined by its starting index in the master string and its length. For example, the string `"overeager"` is encoded as the list of string parts [(37, 3), (14, 2), (46, 2), (9, 2)] denoting the substrings `"ove"`, `"re"`, `"ag"`, and `"er"`.

String parts will be represented by the `StringPart` class shown below.

```
public class StringPart
{
   /** @param start  the starting position of the substring in a master string
    *  @param length  the length of the substring in a master string
    */
   public StringPart(int start, int length)
   {  /* implementation not shown */  }


   /** @return  the starting position of the substring in a master string
    */
   public int getStart()
   {  /* implementation not shown */  }


   /** @return  the length of the substring in a master string
    */
   public int getLength()
   {  /* implementation not shown */  }

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

GO ON TO THE NEXT PAGE.

3. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

   An opossum is an animal whose defense is to pretend to be dead. The `OpossumCritter` class, shown below, will be used to represent the opossum in the grid. An `OpossumCritter` classifies its neighbors as friends, foes, or neither. It is possible that a neighbor is neither a friend nor a foe; however, no neighbor is both a friend and a foe. If the `OpossumCritter` has more foes than friends surrounding it, it will simulate playing dead by changing its color to black and remaining in the same location. Otherwise, it will behave like a `Critter`. If the `OpossumCritter` plays dead for three consecutive steps, it is removed from the grid.

   You will implement two of the methods in the following `OpossumCritter` class.

```
public class OpossumCritter extends Critter
{
  private int numStepsDead;

  public OpossumCritter()
  {
    numStepsDead = 0;
    setColor(Color.ORANGE);
  }

  /** Whenever actors contains more foes than friends, this OpossumCritter plays dead.
   *  Postcondition: (1) The state of all actors in the grid other than this critter and the
   *  elements of actors is unchanged. (2) The location of this critter is unchanged.
   *  @param actors a group of actors to be processed
   */
  public void processActors(ArrayList<Actor> actors)
  {  /* to be implemented in part (a) */  }


  /** Selects the location for the next move.
   *  Postcondition: (1) The returned location is an element of locs, this critter's current location,
   *  or null. (2) The state of all actors is unchanged.
   *  @param locs the possible locations for the next move
   *  @return the location that was selected for the next move, or null to indicate
   *          that this OpossumCritter should be removed from the grid.
   */
  public Location selectMoveLocation(ArrayList<Location> locs)
  {  /* to be implemented in part (b) */  }
```

GO ON TO THE NEXT PAGE.

```
    /** @param other  the actor to check
     *  @return true if other is a friend; false otherwise
     */
    private boolean isFriend(Actor other)
    {  /* implementation not shown */  }

    /** @param other  the actor to check
     *  @return true if other is a foe; false otherwise
     */
    private boolean isFoe(Actor other)
    {  /* implementation not shown */  }
}
```

GO ON TO THE NEXT PAGE.

(a) Override the `processActors` method for the `OpossumCritter` class. This method should look at all elements of `actors` and determine whether or not to play dead according to the types of the actors. If there are more foes than friends, the `OpossumCritter` indicates that it is playing dead by changing its color to `Color.BLACK`. When not playing dead, it sets its color to `Color.ORANGE`. The instance variable `numStepsDead` should be updated to reflect the number of consecutive steps the `OpossumCritter` has played dead.

Complete method `processActors` below.

```
/** Whenever actors contains more foes than friends, this OpossumCritter plays dead.
 *  Postcondition: (1) The state of all actors in the grid other than this critter and the
 *  elements of actors is unchanged. (2) The location of this critter is unchanged.
 *  @param actors a group of actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
```

(b) Override the `selectMoveLocation` method for the `OpossumCritter` class. When the `OpossumCritter` is not playing dead, it behaves like a `Critter`. The next location for an `OpossumCritter` that has been playing dead for three consecutive steps is `null`. Otherwise, an `OpossumCritter` that is playing dead remains in its current location.

Complete method `selectMoveLocation` below.

```
/** Selects the location for the next move.
 *  Postcondition: (1) The returned location is an element of locs, this critter's current location,
 *  or null. (2) The state of all actors is unchanged.
 *  @param locs the possible locations for the next move
 *  @return the location that was selected for the next move, or null to indicate
 *          that this OpossumCritter should be removed from the grid.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
```

GO ON TO THE NEXT PAGE.

4. A *checker* is an object that examines strings and *accepts* those strings that meet a particular criterion.

The `Checker` interface is defined below.

```
public interface Checker
{
  /** @param text  a string to consider for acceptance
   *  @return true if this Checker accepts text; false otherwise
   */
  boolean accept(String text);
}
```

In this question, you will write two classes that implement the `Checker` interface. You will then create a `Checker` object that checks for a particular acceptance criterion.

(a) A `SubstringChecker` accepts any string that contains a particular substring. For example, the following `SubstringChecker` object `broccoliChecker` accepts all strings containing the substring `"broccoli"`.

```
Checker broccoliChecker = new SubstringChecker("broccoli");
```

The following table illustrates the results of several calls to the `broccoliChecker accept` method.

Method Call	Result
`broccoliChecker.accept("broccoli")`	`true`
`broccoliChecker.accept("I like broccoli")`	`true`
`broccoliChecker.accept("carrots are great")`	`false`
`broccoliChecker.accept("Broccoli Bonanza")`	`false`

Write the `SubstringChecker` class that implements the `Checker` interface. The constructor should take a single `String` parameter that represents the particular substring to be matched.

GO ON TO THE NEXT PAGE.