2. This question involves reasoning about the GridWorld case study. Reference materials are provided in the appendices.

A retro bug behaves like a regular bug. It also has the ability to revert to its previous location and direction. When a retro bug acts, it maintains information about its location and direction at the beginning of the act. The retro bug has a `restore` method that restores it to the location (if possible) and direction it faced at the beginning of its previous act. A retro bug only maintains information about its most recent act; therefore, multiple calls to `restore` that occur before its next act will use the same information. The `restore` method has no effect if it is called before a retro bug's first act.

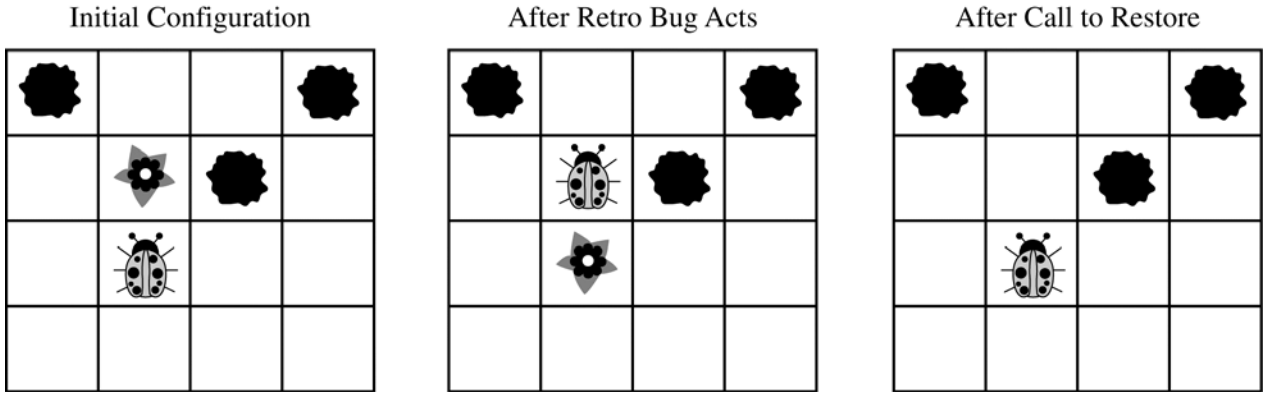
The `restore` method takes no parameters and does not return a value. The `restore` method has the following functionality.

- If the previous location of the retro bug is either unoccupied or contains a flower, the `restore` method places the retro bug in that previous location. The presence of any other type of actor in that location will prevent the retro bug from being placed in that location.
- The `restore` method always ends with the retro bug facing in the same direction that it had been facing at the beginning of its most recent act.

The following examples illustrate the behavior of the `restore` method.

Example 1

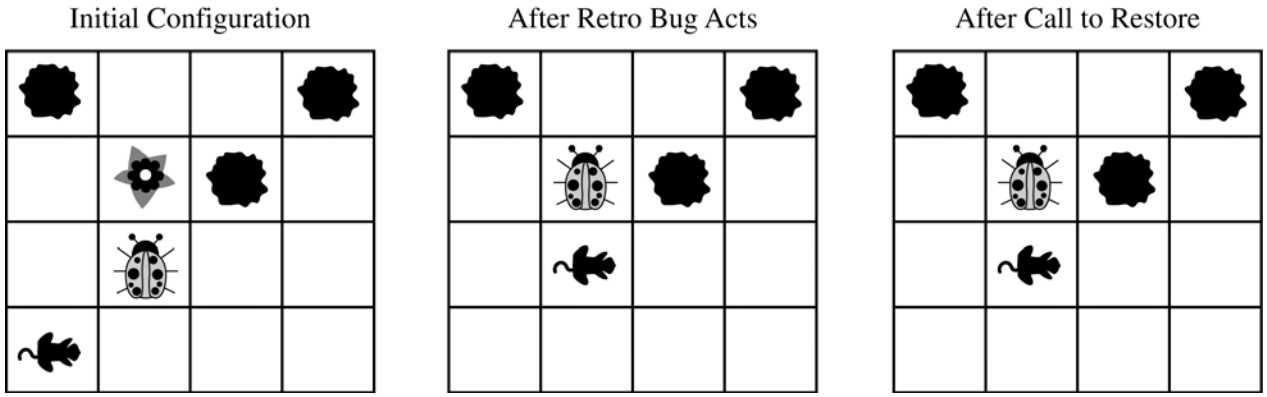
The retro bug acts once and later calls `restore`. Note that the flower that was originally in front of the retro bug is not replaced as a result of the call to `restore`. The retro bug is returned to its previous direction, which, in this case, is the same as the current direction.



Question 2 continues on the next page.

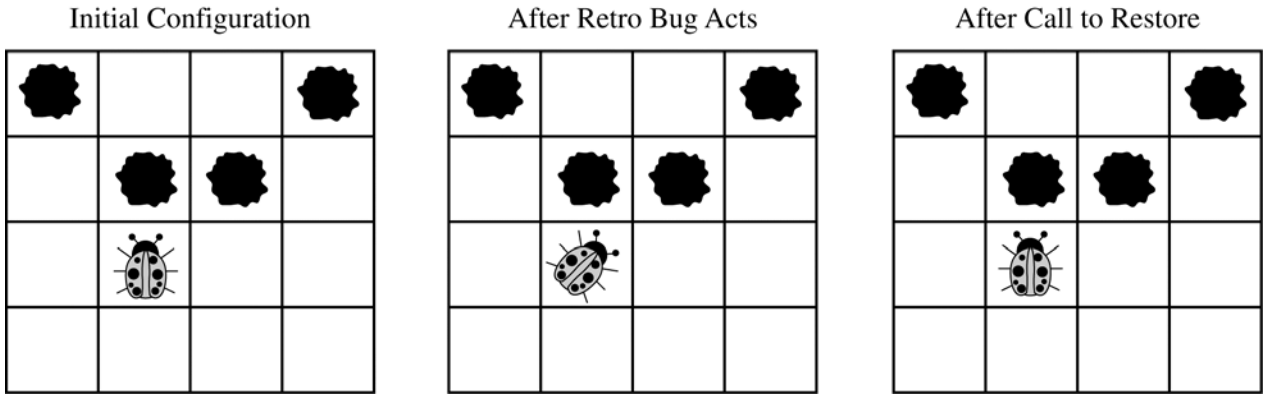
Example 2

The retro bug acts once and then some other actor moves into the location that the retro bug originally held. The call to `restore` results in the retro bug staying in its current location. The retro bug is returned to its previous direction (in this case it is the same as the current direction).



Example 3

The retro bug acts once and later calls `restore`. Because the retro bug is blocked from moving forward, it turns as its first act. The `restore` method results in the retro bug staying in its current location (the same as its previous location) and returning to its previous direction.



WRITE YOUR SOLUTION ON THE NEXT PAGE.

Write the entire `RetroBug` class, including all necessary instance variables and methods.

3. Consider a software system that models a horse barn. Classes that represent horses implement the following interface.

```
public interface Horse
{
  /** @return the horse's name */
  String getName();

  /** @return the horse's weight */
  int getWeight();

  // There may be methods that are not shown.
}
```

A horse barn consists of N numbered spaces. Each space can hold at most one horse. The spaces are indexed starting from 0; the index of the last space is $N - 1$. No two horses in the barn have the same name.

The declaration of the `HorseBarn` class is shown below. You will write two unrelated methods of the `HorseBarn` class.

```
public class HorseBarn
{
  /** The spaces in the barn. Each array element holds a reference to the horse
   *  that is currently occupying the space. A null value indicates an empty space.
   */
  private Horse[] spaces;


  /** Returns the index of the space that contains the horse with the specified name.
   *  Precondition: No two horses in the barn have the same name.
   *  @param name the name of the horse to find
   *  @return the index of the space containing the horse with the specified name;
   *          -1 if no horse with the specified name is in the barn.
   */
  public int findHorseSpace(String name)
  {  /* to be implemented in part (a) */  }


  /** Consolidates the barn by moving horses so that the horses are in adjacent spaces,
   *  starting at index 0, with no empty space between any two horses.
   *  Postcondition: The order of the horses is the same as before the consolidation.
   */
  public void consolidate()
  {  /* to be implemented in part (b) */  }

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

Part (a) begins on page 14.

AP® COMPUTER SCIENCE A
2012 SCORING GUIDELINES

Question 2: RetroBug (GridWorld)

Class:	`RetroBug`	**9 points**

Intent: *Define extension to* `Bug` *class that implements a restore method to revert to previous location and direction*

+1 Provides properly formed class header for `RetroBug` that extends `Bug` class

+1 Overrides at least one `Bug` method, other than constructor, and maintains all `Bug` behaviors

+2 Saves state at beginning of `act`

- **+1** Remembers location or direction in `RetroBug` instance variable at beginning of `act` method and nowhere else
  (*point awarded only if instance variable is explicitly declared*)
- **+1** Remembers both location and direction in `RetroBug` instance variables

+5 Implements `restore`

- **+½** Provides correct method header: `public void restore()`
- **+½** Guards against any effect if called before first invocation of `act`
- **+1** Always restores remembered direction
- **+1** Moves to remembered location
- **+1** Moves if remembered location is empty (*must check for empty location*)
- **+1** Moves if remembered location is occupied only by a flower
  (*must check for flower at location*)

Question-Specific Penalties

- **-1** (r) Use of `"RetroBug."` instead of `"this."`
- **-1** (v) Confused use of location and direction
  (*e.g., saved location used as direction and vice versa*)
- **-1** (z) Attempts to return a value from `restore`
- **-0** Missing `public` qualifier on class header

AP® COMPUTER SCIENCE A
2012 CANONICAL SOLUTIONS

Question 2: RetroBug (GridWorld)

```
public class RetroBug extends Bug {
   Location savedLocation;
   int savedDirection;


   public void act() {
      savedLocation = getLocation();
      savedDirection = getDirection();
      super.act();
   }


   public void restore() {
      if (savedLocation == null) return;
      setDirection(savedDirection);
      if ( getGrid().get(savedLocation) == null
          || getGrid().get(savedLocation) instanceof Flower ) {
         moveTo(savedLocation);
      }
   }

}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.