COMPUTER SCIENCE A SECTION II

Time—1 hour and 45 minutes
Number of questions—4
Percent of total grade—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN Java.

Write your answers in pencil only in the booklet provided.

Notes:

- Assume that the classes in the Quick Reference have been imported where needed.
- Unless otherwise stated, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

1. This question involves reasoning about the code from the GridWorld Case Study. A Quick Reference to the case study is provided as part of this exam.

   Consider modifying the `Critter` class so that a `Critter` has the capability of breeding. Every time it acts, it removes the neighboring critters, as before. Then, if this `Critter` meets the probability requirements for breeding, it breeds into each empty neighboring location by placing a new `Critter` there. If the `Critter` does not breed, it moves, as before, into one of the empty neighboring locations. A partial definition of the modified `Critter` class is shown below.

```
public class Critter extends Actor
{
    private double myProbBreeding;
    private boolean myBreed;

    /**
     * Constructor. Creates a Critter with the given probability of breeding.
     * myBreed is set to false.
     * @param probBreeding the probability of breeding
     */
    public Critter(double probBreeding)
    { /* to be implemented in part (a) */ }

    /**
     * A critter breeds by getting a list of its empty neighboring
     * locations, and placing a new Critter in each.
     */
    private void breed()
    { /* to be implemented in part (b) */ }

    /**
     * A critter acts by getting a list of its neighbors, and processing them.
     * If it passes the probability test for breeding, then it breeds into each
     * of the empty neighboring locations.
     * If not, it gets the possible locations to move to, selects one of them,
     * and moves to the selected location.
     */
    public void act()
    { /* to be implemented in part (c) */ }

    //methods getActors, processActors, getMoveLocations,
    // selectMoveLocation, and makeMove remain unchanged, and are
    // not shown ...
}
```

(a) Write the constructor for the modified `Critter` class. The constructor creates a blue `Critter` facing north, whose probability of breeding is as specified in its parameter. The private instance variable `myBreed` is set to false.

Complete the `Critter` constructor below.

```
/**
 * Constructor. Creates a Critter with the given probability of breeding.
 * myBreed is set to false.
 * @param probBreeding the probability of breeding
 */
public Critter(double probBreeding)
```

(b) Write the private helper method `breed`. This method places a new blue `Critter` facing north into each of the empty neighboring adjacent locations. Each new `Critter` has the same probability of breeding as its parent critter.

Complete method `breed` below.

```
/**
 * A critter breeds by getting a list of its empty neighboring
 * locations, and placing a new Critter in each.
 */
private void breed()
```

(c) Write the modified `act` method in the `Critter` class. The `act` method processes actors as before, "eating" each neighboring adjacent `Actor` that is neither a `Rock` nor another `Critter`. Then it uses a random number to determine whether this `Critter` will breed or not. If the random number is within the probability range for breeding, the `Critter` will breed; otherwise it will move as in the original `act` method.

Complete method `act` below.

```
/**
 * A critter acts by getting a list of its neighbors, and processing them.
 * If it passes the probability test for breeding, then it breeds into each
 * of the empty neighboring locations.
 * If not, it gets the possible locations to move to, selects one of them,
 * and moves to the selected location.
 */
public void act()
```

2. Consider a class, `NoteKeeper`, that is designed to store and manipulate a list of short notes. Here are some typical notes:

```
pick up drycleaning
special dog chow
car registration
dentist Monday
dog license
```

The incomplete class declaration is shown below:

```
public class NoteKeeper
{
    private ArrayList<String> noteList;

    //Postcondition: Prints all notes in noteList, one per line.
    //               Notes are numbered 1, 2, ...
    //               Each number is followed by a period and a space.
    public void printNotes()
    { /* to be implemented in part (a) */ }

    //Postcondition: All notes with specified word have been removed
    //               from noteList, leaving the order of the
    //               remaining notes unchanged. If none of the notes in
    //               noteList contains word, the list remains unchanged.
    public void removeNotes(String word)
    { /* to be implemented in part (b) */ }

    //constructor and other methods not shown ...
}
```

(a) Write the `NoteKeeper` method `printNotes`. The `printNotes` method prints all of the notes in `noteList`, one per line, and numbers the notes, starting at 1. The output should look like this:

```
1. pick up drycleaning
2. special dog chow
3. car registration
4. dentist Monday
5. dog license
```

Complete method `printNotes` below.

```
//Postcondition: Prints all notes in noteList, one per line.
//               Notes are numbered 1, 2, ...
//               Each number is followed by a period and a space.
public void printNotes()
```

(b) Write the `NoteKeeper` method `removeNotes`. Method `removeNotes` removes all notes from `noteList` that contain the word specified by the parameter. The ordering of the remaining notes should be left unchanged. For example, suppose that a `NoteKeeper` variable, `notes`, has a `noteList` containing

```
[pick up drycleaning, special dog chow, car registration,
    dentist Monday, dog license]
```

the method call `notes.removeNotes("dog")` should modify the `noteList` of `notes` to be

```
[pick up drycleaning, car registration, dentist Monday]
```

Complete method `removeNotes` below.

```
//Postcondition: All notes with specified word have been removed
//               from noteList, leaving the order of the
//               remaining notes unchanged. If none of the notes in
//               noteList contains word, the list remains unchanged.
public void removeNotes(String word)
```

3. Refer to the `Coin` and `Wallet` classes below.

```
/* Represents a Coin with a name and value */
public class Coin
{
    private double myValue;
    private String myName;

    //constructor
    public Coin(double value, String name)
    {
        myValue = value;
        myName = name;
    }

    //accessors

    public double getValue()
    { return myValue; }

    public String getName()
    { return myName; }

    //Postcondition: Returns true if this Coin has the same name
    //               and value as Coin c, false otherwise.
    public boolean equals(Coin c)
    { /* to be implemented in part (a) */ }

}


/* A Wallet holds a collection of Coins */
public class Wallet
{
    private ArrayList<Coin> myCoins;

    //Postcondition: Returns total value of cents in Wallet.
    public int getTotal()
    { /* to be implemented in part (b) */ }

    //Postcondition: Returns total value in Wallet,
    //               rounded to the nearest dollar.
    public int roundTotal()
    { /* to be implemented in part (c) */ }


    //Postcondition: Returns number of coins in Wallet that
    //               are equal to Coin c.
    public int howMany(Coin c)
    { /* to be implemented in part (d) */ }

    //constructor and other methods not shown ...
}
```

(a) Write the `equals` method for the `Coin` class. Two `Coin` objects are equal if and only if they have the same name and value.

Complete method `equals` below.

```
//Postcondition: Returns true if this Coin has the same name
//               and value as Coin c, false otherwise.
public boolean equals(Coin c)
```

(b) Write the `getTotal` method for the `Wallet` class. Method `getTotal` should find the total value of all coins in the `Wallet`, in cents.

Complete method `getTotal` below.

```
//Postcondition: Returns total value of cents in Wallet.
public int getTotal()
```

(c) Write the `roundTotal` method for the `Wallet` class. This method should return the total value of coins in the `Wallet`, rounded to the nearest dollar. The table below shows the results of some calls to `roundTotal`.

Total value of cents in `Wallet`	Result returned by `roundTotal`
39	0
650	7
649	6
1000	10

Complete method `roundTotal` below.

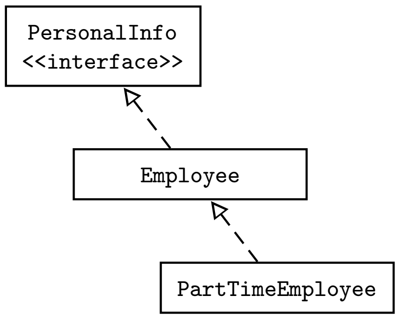
```
//Postcondition: Returns total value in Wallet,
//               rounded to the nearest dollar.
public int roundTotal()
```

(d) Write the `howMany` method for the `Wallet` class. The `howMany` method returns the number of coins in `Wallet` that are equal to `Coin c`. In writing method `howMany`, you should use the `equals` method defined in part (a). Assume that `equals` works as specified, irrespective of what you wrote in part (a).

Complete method `howMany` below.

```
//Postcondition: Returns number of coins in Wallet that
//               are equal to Coin c.
public int howMany(Coin c)
```

4. Consider the hierarchy of classes shown in the following diagram.



`PersonalInfo` is an interface that is implemented by both `Inspector` and `Employee`. A `PartTimeEmployee` *is-a* `Employee`.
Here is the declaration for `PersonalInfo`.

```
public interface PersonalInfo
{
    String getName();
    String getCity();
    boolean getCitizenStatus();
}
```

(a) Given the class hierarchy shown above, write a complete class declaration for the class `Employee`, including implementation of methods and a constructor with parameters. An `Employee`, in addition to implementing `PersonalInfo`, has a data field to store annual salary, and an accessor method that returns the annual salary of the `Employee`.

Write the `Employee` class below.

(b) Given the class hierarchy shown above, write a complete class declaration for the `PartTimeEmployee` class. A `PartTimeEmployee` has two additional data fields:

- A real number that indicates the fraction that the part-time employee works—for example, 0.5, 0.8, and so on.
- A `boolean` field that stores whether the employee is a union member or not.

There are three additional methods:

- An accessor that returns the real number fraction that the `PartTimeEmployee` works.
- An accessor that returns a value indicating whether the employee is a union member or not.
- A mutator that switches the status of that employee's union membership, but only if the employee's salary is greater than $15,000. In other words, if his salary is greater than $15,000, and he is a union member, the mutator switches him to *not* being a union member. If his salary is greater than $15,000, and he is not a union member, the mutator switches him to being a union member.

Write the `PartTimeEmployee` class below.