Athlete (Abstract Class)

PROMPT

The **Athlete** abstract class describes a general athlete. Other classes will use the **Athlete** class to define a specific type of athlete, such as a football player, a basketball player, a golfer, etc.

A partial definition of the **Athlete** abstract class is shown below.

```java
public abstract class Athlete
{
    private String name;  // name of the athlete
    private String sport; // name of the sport
    private int score;    // current score

    public Athlete(String name, String sport)
    {
        this.name = name;
        this.sport = sport;
    }

    /**
     * returns the name of the athlete
     */
    public String getName()
    {
        return name;
    }

    /**
     * returns the sport of the athlete
     */
    public String getSport()
    {
        return sport;
    }
    /**
     * This method is an abstract method to return the number of points
     * (sometimes called runs, strokes, goals, etc. depending on the sport).
     * Points are determined by using the score type of the sport and sometimes
     * the second score parameter (if necessary for the sport).
     * @param scoreType is the description of the type of score in the sport
     * @param scoreParm is a second parameter that is needed to determine the
     * number of points in some sports (not necessary for all sports)
     */
    public abstract int determinePoints(String scoreType, int scoreParm);

    /**
     * Updates the current score by adding the points to the score.  If the
     * result is a negative score, then the score should be zero.
     * @param points = the number of points, runs, strokes, goals, etc.
     * Postcondition:  The score will not be negative.
     */
    public void updateScore(int points)
    {
        // To be implemented in PART A) below
    }

    // other instance variables and methods not shown
}
```

The **Quarterback** class represents a football quarterback and inherits from the **Athlete** class. A partial definition of the **Quarterback** class is shown below.

```
public class Quarterback extends Athlete
{
    private int jerseyNumber;
    private double completionPercent;

    /**
     * This constructor method accepts 4 parameters:
     * @param name is the athlete's name
     * @param sport is the sport of the athlete
     * @param jerseyNum is the jersey number of the player
     * @param completions is the completion percentage of the player
     */
    // The constructor method is to be defined in PART B) below

    public int determinePoints(String scoreType, int scoreParm)
    {
        // implementation not shown
    }

    // instance variables and other methods not shown
}
```

The **Golfer** class represents a golfer and inherits from the **Athlete** class. A partial definition of the **Golfer** class is shown below.

```
public class Golfer extends Athlete
{
    /**
     * returns the number of strokes by using the score type and adding or
     * deducting strokes from par (second parameter) according to the following:
     * if the score type is:
     * eagle: subtract 2 from the par parameter
     * birdie: subtract 1 from the par parameter
     * par: add zero to the par parameter
     * bogey: add 1 to the par parameter
     * double bogey: add 2 to the par parameter
     * triple bogey: add 3 to the par parameter
     * @param scoreType is the description of the score
     * @param par is the number of strokes for par on the specified hole
     */
    public int determinePoints(String scoreType, int par)
    {
        // to be implemented in PART C) below
    }

    // instance variables, constructors, and other methods not shown
}
```

PART A)

The **updateScore** method in the **Athlete** class is used to update the athlete's score. The score is computed by adding the **points** parameter to the current score. The score cannot be negative; if so then make it zero.

Complete the **updateScore** method below:

```
/**
 * Updates the current score by adding the points to the score.  If the
 * result is a negative score, then the score should be zero.
 * @param points = the number of points, runs, strokes, goals, etc.
 * Postcondition:  The score will not be negative.
 */
public void updateScore(int points)
{


}
```

PART B)

The **Quarterback** class represents a football quarterback and inherits from the **Athlete** class. The **Quarterback** constructor method accepts four parameters: in addition to the name and sport of the athlete, it also accepts two additional parameters for the jersey number of the quarterback (integer format) and the completion percentage of the quarterback (in decimal format). You can use any other methods from the **Athlete** or **Quarterback** classes while coding the constructor method.

Code the entire **Quarterback** constructor method below:

```
public class Quarterback extends Athlete
{
    private int jerseyNumber;
    private double completionPercent;

    /**
     * This constructor method accepts 4 parameters:
     * @param name is the athlete's name
     * @param sport is the sport of the athlete
     * @param jerseyNum is the jersey number of the player
     * @param completions is the completion percentage of the player
     */
    // Define the constructor method here:


}
```

PART C)

The **determinePoints** method in the **Golfer** class overrides the same method from the **Athlete** class. This method returns the number of strokes by using the score type and adding or deducting strokes from par (second parameter) according to the following:

If the score type is, then return the following:

- eagle: subtract 2 from the par parameter
- birdie: subtract 1 from the par parameter
- par: add zero to the par parameter
- bogey: add 1 to the par parameter
- double bogey: add 2 to the par parameter
- triple bogey: add 3 to the par parameter

Complete the **determinePoints** method in the **Golfer** class:

```
/**
 * method defined above
 * @param scoreType is the description of the score
 * @param par is the number of strokes for par on the specified hole
 */
public int determinePoints(String scoreType, int par)
{


}
```

Room (Interface)

PROMPT

A **Room** is an interface that specifies three methods used for every type of room. None of these methods accepts a parameter:
1) **sqFootage** returns the square footage of the room in integer format
2) **getNumOutlets** returns the number of electrical outlets in the floor of the room (in integer format)
3) **electricityOk** returns **true** if the electrical capabilities of the room are ok, or **false** otherwise

There are different kinds of rooms: standard rooms, computer rooms, conference rooms, and large meeting rooms. Each room has electrical outlets that are installed in the floor instead of the walls. Many of the rooms have retractable walls so that rooms can be combined when necessary to form larger rooms.

The **StandardRoom** class represents a standard room and uses the **Room** interface. Below is a partial implementation of the **StandardRoom** class:

```java
public class StandardRoom implements Room
{
    /**
     * returns the total number of floor-based electrical outlets
     * in the room
     */
    public int getNumOutlets()
    {
        // implementation not shown
    }

    /**
     * returns the square footage of the room which is calculated
     * by multiplying the length times the width
     */
    public int sqFootage()
    {
        // implementation not shown
    }

    // Instance variables, constructors and some methods not shown
}
```

The **LargeMeetingRoom** class uses the **Room** interface and represents a meeting room used for large company gatherings. It is composed of a number of smaller rooms whose internal walls are temporarily retracted to form this large meeting room. The **rooms** list in this class contains a list of all the rooms used to comprise this large meeting room. Below is a partial implementation of the **LargeMeetingRoom** class:

```
public class LargeMeetingRoom implements Room
{
    private List<Room> rooms; // list of rooms combined to create this large meeting

    /**
     * returns the total number of floor-based electrical outlets
     * in the large meeting room
     */
    public int getNumOutlets()
    {
        // implementation not shown
    }

    /**
     * returns the square footage of the large meeting
     * room which is the total square footage from all
     * the rooms that comprise this large meeting room
     */
    public int sqFootage()
    {
        // implementation not shown
    }

    // Instance variables, constructors and some methods not shown
}
```

PART A)

Define the complete **Room** interface:

PART B)

The electricity capabilities of a standard room are considered ok if there are at least two electrical outlets per 100 square feet of the room. Otherwise, the electrical capabilities are not ok. You can use any other methods from the **StandardRoom** class while coding this method.

Define the complete **electricityOk** method in the **StandardRoom** class:

PART C)

A large meeting room has adequate electricity if all of the rooms that make up this meeting room have adequate electricity and the square footage of the large meeting room is less than 10,000 square feet. Otherwise the electricity capabilities of this room are not ok. You can use any other methods from the classes that implement **Room** while coding this method.

Define the **electricityOk** method in the **LargeMeetingRoom** class:

Chef Certification

PROMPT

A chef can obtain a certification level depending on how many times they have prepared a certain food or a combination of foods. Three different classes are used here: **Food**, **Meal**, and **Chef**.

A **Food** is defined by a food type and the description of the food itself. A food type is the category of food, such as meat, vegetable, appetizer, dessert, etc. The food description is more specific, such as steak, lamb, corn, pumpkin pie, etc. A partial definition of the **Food** class is shown below.

```java
public class Food
{
    /**
     * Constructor method for a food
     * @param foodType is the category of food
     * @param foodDesc is the description of the food
     */
    public Food(String foodType, String foodDesc)
    {
        // implementation not shown
    }

    /**
     * returns the category of the food
     */
    public String getFoodType()
    {
        // implementation not shown
    }

    /**
     * returns the description of the food
     */
    public String getFoodDesc()
    {
        // implementation not shown
    }

    // instance variables and other methods not shown
}
```

A **Meal** is defined by the name of the meal and a list of **Food** prepared for the meal. A partial definition of the **Meal** class is shown below.

```
public class Meal
{
    /**
     * Constructor method for a meal
     * @param mealDesc is the text description of the meal
     * @param foods is an array of food that make up the meal
     */
    public Meal(String mealDesc, Food[] foods)
    {
        // implementation not shown
    }

    /**
     * returns the description of the meal
     */
    public String getMealDesc()
    {
        // implementation not shown
    }

    /**
     * returns an array of food prepared for the meal
     * @return
     */
    public Food[] getFoods()
    {
        // implementation not shown
    }

    // instance variables and other methods not shown
}
```

A **Chef** is defined by the name of the chef, the chef's number of years of experience, and a list of all the meals the chef has ever prepared. Every time that a chef has prepared a meal, it is added to the list. Therefore, if the chef prepared the same meal last week three times and prepared it today two times, then the **meals** array list would contain the meal at least five times. A partial definition of the **Chef** class is shown below.

```
public class Chef
{
    private String name;      // name of the chef
    private List<Meal> meals; // list of all meals prepared by the chef.  If the chef
                              // prepared the meal 20 times, the Meal will
                              // exist 20 times in this list
    private int yearsExperience;    // number of years of experience

    /**
     * Constructor method for a Chef
     * @param name is the name of the chef
     * @param meals is a list of all meals prepared by the chef
     * @param yrs is the number of years of experience for the chef
     */    public Chef(String name, List<Meal> meals, int yrs)
    {
        // implementation not shown
    }
```

```
    /**
     * This method returns the certification level (either A, B, or C)
     * that the chef has obtained for preparing this type of food.
     * If the chef has prepared this food at least 100 times, then they
     * would receive a certification level of "A".  If they have prepared
     * the food from 50-99 times, they would receive a certification level
     * of "B".  Otherwise they would receive a certification level of "C".
     * @param foodType is the category of the food
     * @param foodDesc is the description of the food
     */
    public String foodCertification(String foodType, String foodDesc)
    {
        // to be implemented in PART A) below
    }

    /**
     * This method returns true if the chef is considered a master chef
     * for a certain meal combination.  Otherwise it returns false. A chef
     * is considered a master chef for a meal only if they have an "A"
     * certification for every food in the meal.
     * @param mealCombo is a Meal (combination of foods)
     */
    public boolean mealMasterChef(Meal mealCombo)
    {
        // to be implemented in PART B) below
    }

    // other methods not shown
}
```

PART A)

The **foodCertification** method in the **Chef** class accepts two parameters: the type (category) of food, and the more specific description of the food. This method needs to determine and return the certification level the chef has earned for preparing this food. The certification level is based solely on how many times the chef has prepared the food. If the chef has prepared this food at least 100 times in any of the meals the chef has made, then the chef earns an “A” certification level. Otherwise, if the chef has prepared the food at least 50 times then he/she earns a “B” certification level. Otherwise, he/she earns a “C” certification level. You can use any other methods from the **Food**, **Meal**, or **Chef** classes while coding this method.

As an example:

If Chef Sarah has prepared a meal of chicken and green beans 70 times, a meal of lamb and carrots 150 times, and a meal of steak and green beans 40 times, then
calling **foodCertification("meat", "chicken")** will return **"B"**
calling **foodCertification("vegetable", "carrots")** will return **"A"**
calling **foodCertification("vegetable", "green beans")** will return **"A"** (since 70 + 40 = 110)
calling **foodCertification("meat", "steak")** will return **"C"**

Complete the **foodCertification** method below:

```
/**
 * This method returns the certification level (either A, B, or C)
 * that the chef has obtained for preparing this type of food.
 * If the chef has prepared this food at least 100 times, then they
 * would receive a certification level of "A".  If they have prepared
 * the food from 50-99 times, they would receive a certification level
 * of "B".  Otherwise they would receive a certification level of "C".
 * @param foodType is the category of the food
 * @param foodDesc is the description of the food
*/
public String foodCertification(String foodType, String foodDesc)
{


}
```

PART B)

The **mealMasterChef** method in the **Chef** class accepts a **Meal** as its only parameter and determines if the chef is a master chef for a given combination of foods. This method will return true only if the chef has earned an "A" certification level for every food in the Meal parameter. Otherwise it will return false. You can use any other methods from the **Food**, **Meal**, or **Chef** classes while coding this method.

As an example:

If Chef Sarah has prepared a meal of chicken and green beans 70 times, a meal of lamb and carrots 150 times, and a meal of steak and green beans 40 times, then

if **mealA** is the "Chef's Special" composed of chicken and green beans calling **sarah.mealMasterChef(mealA)** will return **false** (since Sarah has a "B" certification level for chicken and an "A" certification level for green beans)

if **mealB** is the "Lamb Delite" composed of lamb and green beans calling **sarah.mealMasterChef(mealB)** will return **true** (since Sarah has an "A" certification level for both lamb and green beans)

Complete the **mealMasterChef** method below:

Complete the **mealMasterChef** method below:

```
/**
 * This method returns true if the chef is considered a master chef
 * for a certain meal combination.  Otherwise it returns false. A chef
 * is considered a master chef for a meal only if they have an "A"
 * certification for every food in the meal.
 * @param mealCombo is a Meal or combination of foods
 */
public boolean mealMasterChef(Meal mealCombo)
{


}
```