Project: Ball Players – Part 1

Deliverables

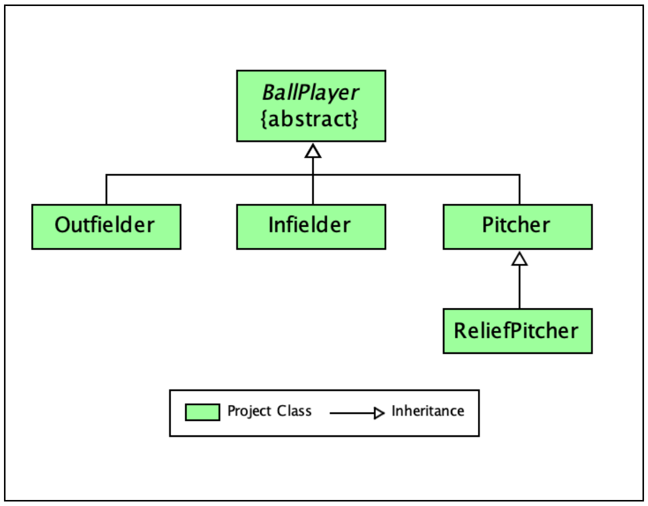
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. To avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The grades for the Completed Code submission will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

- BallPlayer.java, *no test file (abstract class)* - methods should be tested in OutfielderTest.java
- Outfielder.java, OutfielderTest.java
- Infielder.java, InfielderTest.java
- Pitcher.java, PitcherTest.java
- ReliefPitcher.java, ReliefPitcherTest.java

Specifications

Overview: This project is the first of three that will involve the pay analysis and reporting for ball players. You will develop Java classes that represent categories of ball players including outfielders, infielders, pitchers, and relief pitchers. Note that there is no requirement for a class with a main method in this project. You will need to create a JUnit test file for the indicated classes and write one or more test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project and add the class and test files as they are created. All your files should be in a single folder. The UML class diagram at right provides a visual overview of how the classes in the project relate to one another. As you add your classes to the jGRASP project, you should generate the UML class diagram by double-clicking <UML> for the project in the Open Projects window. Once generated, you can use the mouse to select/drag a class to arrange the diagram similar to the one shown.



You should read through the remainder of this assignment before you start coding.

- **BallPlayer.java**

  Requirements: Create an *abstract* BallPlayer class that stores ball player data and provides methods to access the data.

  Design: The BallPlayer class has fields, a constructor, and methods as outlined below.

  (1) **Fields**: *instance* variables for the player's number of type String, the player's name of type String, the player's position of type String, the player's base salary of type double, the player's bonus adjustment factor of type double, and the player's batting average of type double; *static* (or class) variable for the count of BallPlayer objects that have been created (set to zero when declared). These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of BallPlayer. <u>These are the only fields that this class should have</u>.

  (2) **Constructor**: The BallPlayer class must contain a constructor that accepts six parameters representing the values to be assigned to the *instance* fields above. Since this class is abstract, the constructor will be called from the subclasses of BallPlayer using *super* and the parameter list.

  (3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
    - `getNumber`: Accepts no parameters and returns a String representing the number.
    - `setNumber`: Accepts a String representing the number, sets the field, and returns nothing.
    - `getName`: Accepts no parameters and returns a String representing the name.
    - `setName`: Accepts a String representing the name, sets the field, and returns nothing.
    - `getPosition`: Accepts no parameters and returns a String representing the position.
    - `setPosition`: Accepts a String representing the position, sets the field, and returns nothing.
    - `getBaseSalary`: Accepts no parameters and returns a double representing the base salary.
    - `setBaseSalary`: Accepts a double representing the base salary, sets the field, and returns nothing.
    - `getBonusAdjustmentFactor`: Accepts no parameters and returns a double representing the bonus adjustment factor.
    - `setBonusAdjustmentFactor`: Accepts a double representing the bonus adjustment factor, sets the field, and returns nothing.
    - `getBattingAvg`: Accepts no parameters and returns a double representing batting average.

- o `setBattingAvg`: Accepts a double representing the batting average, sets the field, and returns nothing.
- o `getCount`: Accepts no parameters and returns an int representing the count. Since count is *static*, this method should be *static* as well.
- o `resetCount`: Accepts no parameters, resets count to zero, and returns nothing. Since count is *static*, this method should be *static* as well.
- o `stats`: accepts no parameters and returns a String representing the batting average using ".000" as the pattern for DecimalFormat (in the Outfielder example below the return value is `.375`).This should be called in the toString method below to get the batting average. See the Outfielder class below for an example of batting average in the toString result. Subclasses Pitcher and ReliefPitcher should override this method so that pitching statistics are returned instead of batting average.
- o `toString`: Returns a String describing the BallPlayer object. This method will be inherited by the subclasses and will be called implictly by instances of the subclasses unless it is overridden in the subclass. If it is overridden, then it may be called from the toString method in the subclasses of BallPlayer using super.toString(). For an example of the toString result, see the Outfielder class below. Note that you can get the class name for an instance c by calling c.getClass().

  `totalEarnings`: An *abstract* method that accepts no parameters and returns a double representing the total earning of a ball player. Since this is abstract, each non-abstract subclass must implement this method.

Code and Test: Since BallPlayer is *abstract*, you cannot create instances to test. You will need to use instances of a subclass, e.g., Outfielder, which specified below. Thus, it is common to test the methods in an abstract class in the test file for the first non-abstract subclass (i.e., Outfielder).

- **Outfielder.java**

Requirements: Derive the class Outfielder from BallPlayer.

Design: The Outfielder class has fields, a constructor, and methods as outlined below.

(1) **Field**: *instance* variable for outfielderFieldingAvg of type double. This variable should be declared with the *private* access modifier. This is the only field that should be declared in this class.

(2) **Constructor**: The Outfielder class must contain a constructor that accepts seven parameters representing the six instance fields in the BallPlayer class and the one instance field declared in Outfielder. Since this class is a subclass of BallPlayer, the super constructor should be called with field values for BallPlayer. The instance variable outfielderFieldingAvg should be set with the last parameter. Below is an example of how the constructor could be used to create an Outfielder object:

```
Outfielder p1 = new Outfielder("32", "Pat Jones", "RF", 150000,
                               1.25, .375, .950);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- o `getOutfielderFieldingAvg`: Accepts no parameters and returns a double representing outfielderFieldingAvg.
- o `setOutfielderFieldingAvg`: Accepts a double representing the outfielderFieldingAvg, sets the field, and returns nothing.
- o `totalEarnings`: Accepts no parameters and returns a double representing the total earnings for the player calculated by multiplying the base salary by the bonus adjustment factor, batting average, and outfielderFieldingAvg, and then adding the result to the base salary to get the total earnings.
- o `toString`: There is no toString method in the Outfielder class. When toString is invoked on an instance of Outfielder, the toString method inherited from BallPlayer is called. Below is an example of the toString result for Outfielder p1 as it is declared above.

```
32 Pat Jones (RF) .375
Base Salary: $150,000.00 Bonus Adjustment Factor: 1.25
Total Earnings: $216,796.88 (class Outfielder)
```

Code and Test: As you implement the Outfielder class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Outfielder in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. <u>After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding OutfielderTest.java file. In addition, you need to write test methods for the methods inherited from BallPlayer. Instances of BallPlayer, cannot be created, but an Outfielder is-a BallPlayer</u>.

- **<u>Infielder.java</u>**

Requirements: Derive the class Infielder from BallPlayer.

Design: The Infielder class has a field, a constructor, and methods as outlined below.

(1) **Field**: instance variable for infielderFieldingAvg of type double. This variable should be declared with the *private* access modifier. <u>This is the only field that should be declared in this class</u>.

(2) **Constructor**: The Infielder class must contain a constructor that accepts seven parameters representing the six instance fields in the BallPlayer class and the one instance field declared in Infielder. Since this class is a subclass of BallPlayer, the super constructor should be called with field values for BallPlayer. The instance variable infielderFieldingAvg should be set with the last parameter. Below is an example of how the constructor could be used to create an Infielder object:

```
Infielder p2 = new Infielder("23", "Jackie Smith", "3B", 150000,
                              2.50, .275, .850);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
- `getInfielderFieldingAvg`: Accepts no parameters and returns a double representing infielderFieldingAvg.
- `setInfielderFieldingAvg`: Accepts a double representing the infielderFieldingAvg, sets the field, and returns nothing.
- `totalEarnings`: Accepts no parameters and returns a double representing the total earnings for the player calculated by multiplying the base salary by the bonus adjustment factor, batting average, and infielderFieldingAvg, and then adding the result to the base salary to get the total earnings.
- `toString`: There is no toString method in the Infielder class. When toString is invoked on an instance of Infielder, the toString method inherited from BallPlayer is called. Below is an example of the toString result for Infielder p2 as it is declared above.

```
23 Jackie Smith (3B) .275
Base Salary: $150,000.00 Bonus Adjustment Factor: 2.5
Total Earnings: $237,656.25 (class Infielder)
```

Code and Test: As you implement the Infielder class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Infielder in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding InfielderTest.java file.

- **Pitcher.java**

Requirements: Derive the class Pitcher from BallPlayer.

Design: The Pitcher class has a field, a constructor, and methods as outlined below.

(1) **Field**: *instance* variables for wins of type int, losses of type int, and era (a.k.a., earned run average) of type double. These fields should be declared with the *protected* access modifier. These are the only fields that should be declared in this class.

(2) **Constructor**: The Pitcher class must contain a constructor that accepts nine parameters representing the six values for the instance fields in the BallPlayer class and three for the instance fields declared in Pitcher. Since this class is a subclass of BallPlayer, the super constructor should be called with values for BallPlayer. The instance variables wins, losses, and era should be set with the last three parameters. Below is an example of how the constructor could be used to create a Pitcher object:

```
Pitcher p3 = new Pitcher("43", "Jo Williams", "RHP", 150000,
                          3.50, .125, 22, 4, 2.85);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- o `getWins`: Accepts no parameters and returns an int representing wins.
- o `setWins:` Accepts an int representing the wins, sets the field, and returns nothing.
- o `getLosses`: Accepts no parameters and returns an int representing losses.
- o `setLosses:` Accepts an int representing the losses, sets the field, and returns nothing.
- o `getEra`: Accepts no parameters and returns a double representing era.
- o `setEra:` Accepts a double representing era, sets the field, and returns nothing.
- o `totalEarnings`: Accepts no parameters and returns a double representing the total earnings for the player calculated by multiplying the base salary by the bonus adjustment factor, (1 / (1 + era)), and (wins - losses) / 25.0, and then adding the result to the base salary to get the total earnings.
- o `stats`: accepts no parameters and returns a String representing the wins, losses, and era (in the example below the return value is: `22 wins, 4 losses, 2.85 ERA`). This should be called in the toString method below to get the pitcher stats that follow the name and position. This method overrides the method declared in BallPlayer so that pitching statistics are returned instead of batting average. Note that no decimal format object is needed for ERA.
- o `toString`: There is no toString method in the Pitcher class. When toString is invoked on an instance of Pitcher, the toString method inherited from BallPlayer is called. Below is an example of the toString result for Infielder p2 as it is declared above.

```
43 Jo Williams (RHP) 22 wins, 4 losses, 2.85 ERA
Base Salary: $150,000.00 Bonus Adjustment Factor: 3.5
Total Earnings: $248,181.82 (class Pitcher)
```

Code and Test: As you implement the Pitcher class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Pitcher in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding PitcherTest.java file.

- **ReliefPitcher.java**

Requirements: Derive the class ReliefPitcher from class Pitcher.

Design: The ReliefPitcher class has a field, a constructor, and methods as outlined below.

(1) **Field**: *instance* variable for saves of type int. This field should be declared with the *private* access modifier. This is the only field that should be declared in this class.

(2) **Constructor**: The ReliefPitcher class must contain a constructor that accepts ten parameters representing the six values for the instance fields in the BallPlayer class, three for the instance fields declared in Picher, and one for the instance variable in ReliefPitcher. Since this class is a subclass of Pitcher, the super constructor should be called with nine values for the Pitcher constructor. The instance variable saves should be set with the last parameter. Below is an example of how the constructor could be used to create a ReliefPitcher object:

```
ReliefPitcher p4 = new ReliefPitcher("34", "Sammi James", "LHP", 150000,
                                3.50, .125, 5, 4, 3.85, 17);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getSaves`: Accepts no parameters and returns an int representing saves.
- `setSaves`: Accepts an int representing saves, sets the field, and returns nothing.
- `totalEarnings`: Accepts no parameters and returns a double representing the total earnings for the player calculated by multiplying the base salary by the bonus adjustment factor, (1 / (1 + era)), and (wins - losses + saves) / 30.0, and then adding the result to the base salary to get the total earnings.
- `stats`: accepts no parameters and returns a String representing the wins, losses, saves and era (in the example below: `5 wins, 4 losses, 17 saves, 3.85 ERA`). This should be called in the toString method below to get the pitcher stats that follow the name and position. This method overrides the method declared in BallPlayer so that pitching statistics are returned instead of batting average.
- `toString`: There is no toString method in the ReliefPitcher class. When toString is invoked on an instance of ReliefPitcher, the toString method inherited from Pitcher (which was inherited from BallPlayer) is called. Below is an example of the toString result for ReliefPitcher p4 as it is declared above.

```
34 Sammi James (LHP) 5 wins, 4 losses, 17 saves, 3.85 ERA
Base Salary: $150,000.00 Bonus Adjustment Factor: 3.5
Total Earnings: $214,948.45 (class ReliefPitcher)
```

Code and Test: As you implement the ReliefPitcher class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of ReliefPitcher in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding ReliefPitcherTest.java file.

UML Class Diagram: If you have not already done so, add your classes to the jGRASP project, then generate the UML class diagram by double-clicking <UML> for the project in the Open Projects window. Once generated, you can use the mouse to select/drag a class to arrange the diagram like the one on page 1.