Project 9: Softball Players – Part 1

Due: Skeleton Code **(**ungraded – checks class names, method names, parameters, and return types)
Completed Code – **Thursday, April 14, 2016 by 11:59 p.m.**

Deliverables

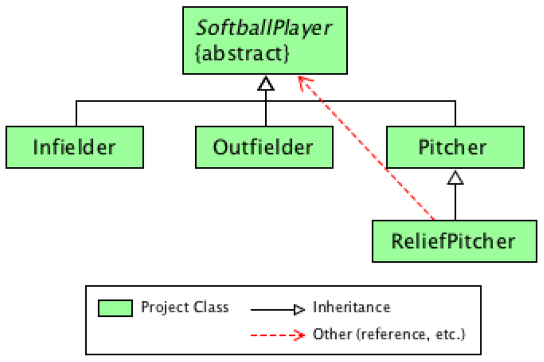
Your project files should be submitted to Web-CAT by the due date and time specified above (see the Lab Guidelines for information on submitting project files). You may submit your skeleton code files until the project due date but should try to do this by Friday (there is no late penalty since this is ungraded). You must submit your completed code files to Web-CAT before 11:59 PM on the due date for the completed code to avoid a late penalty for the project. You may submit your completed code up to 24 hours after the due date, but there is a late penalty of 15 points. No projects will be accepted after the one-day late period. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your lab instructor before the deadline. The Completed Code will be tested against your test methods in your JUnit test files and against the usual correctness tests. The grade will be determined, in part, by the tests that you pass or fail and the level of coverage attained in your Java source files by your test methods.

Files to submit to Web-CAT:

- SoftballPlayer.java
- Outfielder.java, OutfielderTest.java
- Infielder.java, InfielderTest.java
- Pitcher.java, PitcherTest.java
- ReliefPitcher.java, ReliefPitcherTest.java
- (Optional) SoftballPlayersPart1.java, SoftballPlayersPart1Test.java

Specifications

Overview: This project is the first of three that will involve the rating and reporting for softball players. You will develop Java classes that represent categories of softball players including outfielders, infielders, pitchers and relief pitchers. You may also want to develop an optional driver class with a main method. As you develop each class, you should create the associated JUnit test file with the required test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project upfront and then add the source and test files as they are created. All of your files should be in a single folder. Below is the UML class diagram for the required classes which shows the inheritance relationships. The "other" dependency (red dashed line) is for ReliefPitcher using the constant BASE_RATING in SoftballPlayer. This is not shown for the other subclasses because they already have a dependency arrow drawn directly to SoftballPlayer.



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

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

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

  Design: The SoftballPlayer 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 specialization factor of type double, and the player's batting average of type double; *static* (or class) variable for the count of SoftballPlayer objects that have been created (set to zero when declared and incremented in the constructor); a constant (static final) BASE_RATING of type int with value 10. These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of SoftballPlayer. <u>These are the only fields that this class should have</u>.

  (2) **Constructor**: The SoftballPlayer class must contain a constructor that accepts five parameters representing the values to be assigned to the *instance* fields: number, name, position, specialization factor, and batting average. Since this class is abstract, the constructor will be called from the subclasses of SoftballPlayer using *super* and the parameter list. The count field should be incremented in the constructor.

  (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.
  - `getBattingAvg`: Accepts no parameters and returns a double representing batting average.
  - `setBattingAvg`: Accepts a double representing the batting average, sets the field, and returns nothing.
  - `getSpecializationFactor`: Accepts no parameters and returns a double representing the specialization factor.
  - `setSpecializationFactor`: Accepts a double representing the specialization factor, sets the field, and returns nothing.

- `getCount`: Accepts no parameters and returns an int representing the count. Since count is *static*, this method should be *static* as well.
- `resetCount`: Accepts no parameters, resets count to zero, and returns nothing. Since count is *static*, this method should be *static* as well.
- `stats`: accepts no parameters and returns a String representing the batting average. 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.
- `toString`: Returns a String describing the SoftballPlayer object. This method will be inherited by the subclasses. This is the only toString method in this project; it should not be overridden. This methed will call the stats method described above. 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().
- `rating`: An *abstract* method that accepts no parameters and returns a double representing the rating of a softball player. Since this is abstract, each non-abstract subclass must implement this method.

Code and Test: Since the SoftballPlayer class is abstract you cannot create instances of SoftballPlayer upon which to call the methods. However, these methods will be inherited by the subclasses of SoftballPlayer. You should consider first writing skeleton code for the methods in order to compile SoftballPlayer so that you can create the first subclass described below. At this point you can begin completing the methods in SoftballPlayer and writing the JUnit test methods for your subclass that tests the methods in SoftballPlayer.

- **Outfielder.java**

Requirements: Derive the class Outfielder from SoftballPlayer.

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 six parameters representing the five instance fields in the SoftballPlayer class (number, name, position, specialization factor, and batting average) and the one instance field outfielderFieldingAvg declared in Outfielder. Since this class is a subclass of SoftballPlayer, the super constructor should be called with field values for SoftballPlayer. 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", 1.0, .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.

- `getOutfielderFieldingAvg`: Accepts no parameters and returns a double representing outfielderFieldingAvg.
- `setOutfielderFieldingAvg`: Accepts a double representing the outfielderFieldingAvg, sets the field, and returns nothing.
- `rating`: Accepts no parameters and returns a double representing the rating for the player calculated by multiplying BASE_RATING by the specialization factor, the batting average, and outfielder fielding averge.
- `toString`: NONE. When toString is invoked on an instance of Outfielder, the toString method inherited from SoftballPlayer is called. Below is an example of the toString result for Outfielder p1 as it is declared above.

```
32 Pat Jones (RF) .375
Specialization Factor: 1.0 (class Outfielder) Rating: 3.562
```

Code and Test: As you implement the Outfielder class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. You can now continue developing the methods in SoftballPlayer (parent class of Outfielder). The test methods in OutfielderTest should be used to test the methods in both SoftballPlayer and Outfielder. Remember, Outfielder *is-a* SoftballPlayer which means Outfielder inherited the instance method defined in SoftballPlayer. Therefore, you can create instances of Outfielder in order to test methods of the SoftballPlayer class. You may also consider developing SoftballPlayersPart1 (page 7) in parallel with this class to aid in testing.

- **Infielder.java**

Requirements: Derive the class Infielder from SoftballPlayer.

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. This is the only field that should be declared in this class.

(2) **Constructor**: The Infielder class must contain a constructor that accepts six parameters representing the five instance fields in the SoftballPlayer class (number, name, position, specialization factor, and batting average) and the one instance field infielderFieldingAvg declared in Infielder. Since this class is a subclass of SoftballPlayer, the super constructor should be called with field values for SoftballPlayer. 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",
                             1.25, .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.
- `rating`: Accepts no parameters and returns a double representing the rating for the player calculated by multiplying BASE_RATING by the specialization factor, batting average, and infielder fielding average.
- `toString`: NONE. When toString is invoked on an instance of Infielder, the toString method inherited from SoftballPlayer is called. Below is an example of the toString result for Infielder p2 as it is declared above.

```
23 Jackie Smith (3B) .275
Specialization Factor: 1.25 (class Infielder) Rating: 2.922
```

Code and Test: As you implement the Infielder class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Infielder in a JUnit test method in the InfielderTest class and then run the test file. If you want to view your objects in the Canvas, set a breakpoint in your test method the run *Debug* on the test file. When it stops at the breakpoint, step until the object is created. Then open a canvas window using the canvas button at the top of the Debug tab. After you drag the instance onto the canvas, you can examine it for correctness. If you change the viewer to "toString" view, you can see the formatted toString value. You can also enter the object variable name in interactions and press ENTER to see the toStrng value. *Hint: If you use the same variable names for objects in the test methods, you can use the menu button on the viewer in the canvas to set "Scope Test" to "None". This will allow you to use the same canvas with multiple test methods*. You may also consider developing SoftballPlayersPart1 (page 7) in parallel with this class to aid in testing.

- **Pitcher.java**

Requirements: Derive the class Pitcher from SoftballPlayer.

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 eight parameters representing the five values for the instance fields in the SoftballPlayer class (number, name, position, specialization factor, and batting average) and three for the instance fields declared in Pitcher. Since this class is a subclass of SoftballPlayer, the super constructor should be called with values for SoftballPlayer. 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", 2.0, .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.

- `getWins`: Accepts no parameters and returns an int representing wins.
- `setWins`: Accepts an int representing the wins, sets the field, and returns nothing.
- `getLosses`: Accepts no parameters and returns an int representing losses.
- `setLosses`: Accepts an int representing the losses, sets the field, and returns nothing.
- `getEra`: Accepts no parameters and returns a double representing era.
- `setEra`: Accepts a double representing era, sets the field, and returns nothing.
- `rating`: Accepts no parameters and returns a double representing the rating for the player calculated by multiplying BASE_RATING by the specialization factor, (1 / (1 + era)), and ((wins - losses) / 25.0).
- `stats`: accepts no parameters and returns a String representing the wins, losses, and 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 SoftballPlayer so that pitching statistics are returned instead of batting average.
- `toString`: NONE. When toString is invoked on an instance of Pitcher, the toString method inherited from SoftballPlayer 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
Specialization Factor: 2.0 (class Pitcher) Rating: 3.740
```

Code and Test: As you implement the Infielder class, you should compile and test it as methods are created. For details, see **Code and Test** above for the Outfielder and Infielder classes. You may also consider developing SoftballPlayersPart1 (page 7) in parallel with this class to aid in testing.

- **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 nine parameters representing the five values for the instance fields in the SoftballPlayer class (number, name,

position, specialization factor, and batting average), three for the instance fields declared in Picher (wins, losses, and era), and one for the instance variable saves in ReliefPitcher. Since this class is a subclass of Pitcher, the super constructor should be called with eight 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",
                              2.0, .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.
- `rating`: Accepts no parameters and returns a double representing the rating for the player calculated by multiplying BASE_RATING by the specialization factor, (1 / (1 + era)), and ((wins - losses + saves) / 30.0).
- `stats`: accepts no parameters and returns a String representing the wins, losses, saves and 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 SoftballPlayer 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 SoftballPlayer) 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
Specialization Factor: 2.0 (class ReliefPitcher) Rating: 2.474
```

Code and Test: As you implement the ReliefPitcher class, you should compile and test it as methods are created. For details, see **Code and Test** above for the Outfielder and Infielder classes. You may also consider developing SoftballPlayersPart1 (page 7) in parallel with this class to aid in testing.

- **SoftballPlayersPart1.java (Optional)**

Requirements: Driver class with main method is optional but you may find it helpful.

Design: The SoftballPlayersPart1 class only has a main method as described below.

The main method should be developed incrementally along with the classes above. For example, when you have compiled SoftballPlayer and Outfielder, you can add statements to main that create and print an instance of Outfielder. [Since SoftballPlayer is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print instances of Outfielder, Infielder, Pitcher, and ReliefPitcher. Since printing the objects will not show all of the details of the fields, you should also run SoftballPlayersPart1 in the canvas (or debugger with a breakpoint) to examine the objects. Between steps you can use interactions to invoke methods on the objects in the usual way. For example, if you create p1, p2, p3, and p4 as described in the sections above and your main method is stopped between steps after p4 has been created, you can enter the following in interactions to get the rating for the ReliefPitcher object.

```
▶  p4.rating()
   2.4742268041237114
```

The output from main assuming you create print the four objects p1, p2, p3, and p4 as described in the sections above is shown as below. Note that new lines were added by main to achieve the spacing between objects.

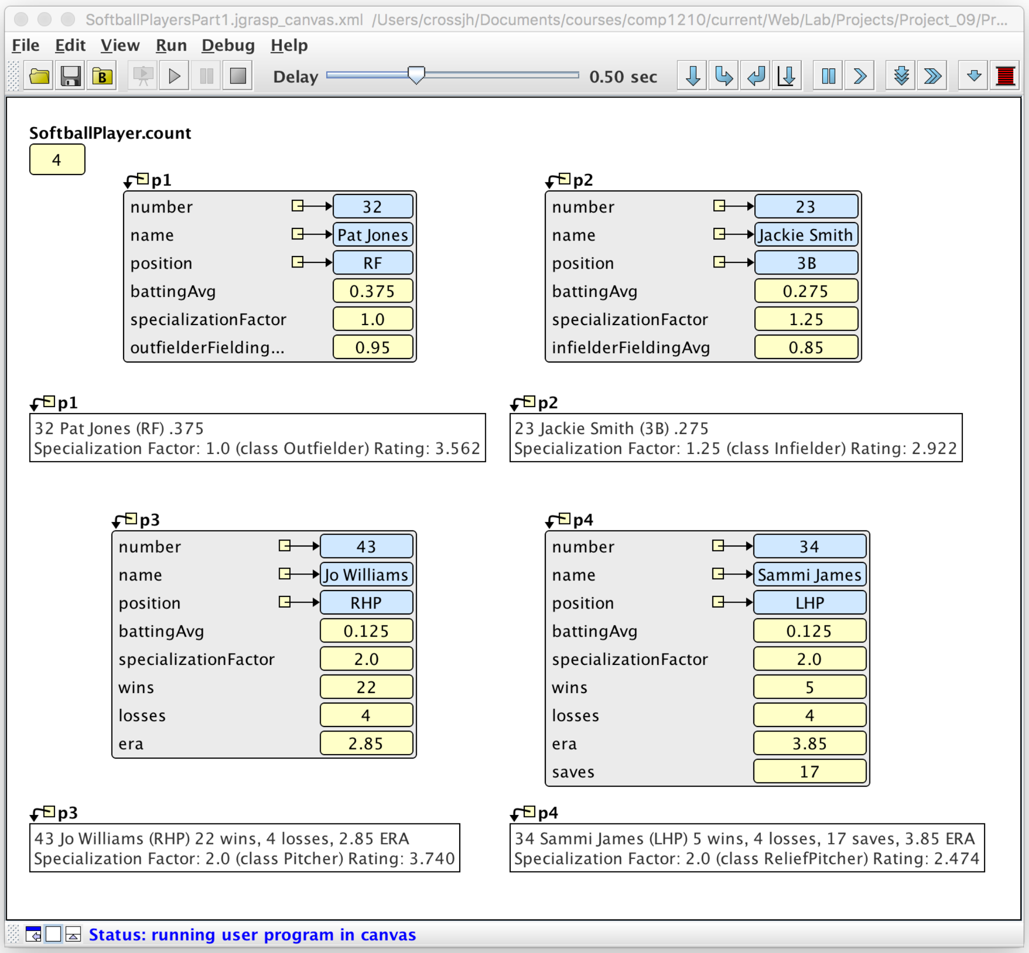
```
32 Pat Jones (RF) .375
Specialization Factor: 1.0 (class Outfielder) Rating: 3.562

23 Jackie Smith (3B) .275
Specialization Factor: 1.25 (class Infielder) Rating: 2.922

43 Jo Williams (RHP) 22 wins, 4 losses, 2.85 ERA
Specialization Factor: 2.0 (class Pitcher) Rating: 3.740

34 Sammi James (LHP) 5 wins, 4 losses, 17 saves, 3.85 ERA
Specialization Factor: 2.0 (class ReliefPitcher) Rating: 2.474
```

Code and Test: After you have implemented the SoftballPlayersPart1 class, you should create the test file SoftballPlayersPart1Test.java in the usual way. The only test method you need is one that checks the class variable *count* that was declared in SoftballPlayer and inherited by each subclass. In the test method, you should reset *count*, call your main method, then assert that *count* is four (assuming that your main creates four objects from the SoftballPlayer hierarchy). The following statements accomplish the test.

```
      SoftballPlayer.resetCount();
      SoftballPlayers1.main(null);
      Assert.assertEquals("Player.count should be 4. ",
                          4, SoftballPlayer.getCount());
```

Canvas for SoftballPlayerPart1

Below is an example of a jGRASP viewer canvas for SoftballPlayerPart1 that contains a viewer for the class variable SoftballPlayer.count and two viewers for each of p1, p2, p3, and p4. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. The canvas was created dragging instances from the debug tab into a new canvas window and setting the appropriate viewer. Note that you will need to unfold one of the instances in the debug tab to find the static variable *count*.


SoftballPlayersPart1.jgrasp_canvas.xml /Users/crossjh/Documents/courses/comp1210/current/Web/Lab/Projects/Project_09/Pr...
File Edit View Run Debug Help
Delay 0.50 sec

SoftballPlayer.count

4

p1

number	32
name	Pat Jones
position	RF
battingAvg	0.375
specializationFactor	1.0
outfielderFielding...	0.95

p2

number	23
name	Jackie Smith
position	3B
battingAvg	0.275
specializationFactor	1.25
infielderFieldingAvg	0.85

p1

```
32 Pat Jones (RF) .375
Specialization Factor: 1.0 (class Outfielder) Rating: 3.562
```

p2

```
23 Jackie Smith (3B) .275
Specialization Factor: 1.25 (class Infielder) Rating: 2.922
```

p3

number	43
name	Jo Williams
position	RHP
battingAvg	0.125
specializationFactor	2.0
wins	22
losses	4
era	2.85

p4

number	34
name	Sammi James
position	LHP
battingAvg	0.125
specializationFactor	2.0
wins	5
losses	4
era	3.85
saves	17

p3

```
43 Jo Williams (RHP) 22 wins, 4 losses, 2.85 ERA
Specialization Factor: 2.0 (class Pitcher) Rating: 3.740
```

p4

```
34 Sammi James (LHP) 5 wins, 4 losses, 17 saves, 3.85 ERA
Specialization Factor: 2.0 (class ReliefPitcher) Rating: 2.474
```

Status: running user program in canvas