Due: Skeleton Code (ungraded – checks class names, method names, parameters, and return types)
Completed Code – **Thursday, November 10, 2016 by 11:59 p.m.**

Deliverables

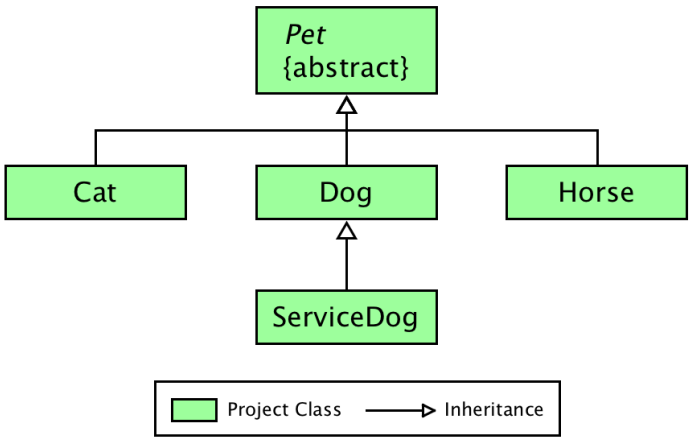
Your project files should be submitted to Web-CAT by the due date and time specified above (see the Lab Guidelines for information on submitting project files). You may submit your skeleton code files until the project due date but should try to do this by Friday (there is no late penalty since this 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:

- Pet.java
- Cat.java, CatTest.java
- Dog.java, DogTest.java
- ServiceDog.java, ServiceDogTest.java
- Horse.java, HorseTest.java
- (Optional) PetBoardingPart1.java, PetBoardingPart1Test.java

Specifications

Overview: This project is the first of three that will involve the boarding and reporting for pets. You will develop Java classes that represent categories of pets including cats, dogs, service dogs, and horses. 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.



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

- **Pet.java**

  Requirements: Create an *abstract* Pet class that stores pet data and provides methods to access the data.

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

  (1) **Fields**: *instance* variables for the pet's owner of type String, the pet's name of type String, the pet's breed of type String, the pet's weight of type double, and the pet's days to be boarded of type int; *static* (or class) variable of type int for the count of Pet objects that have been created (set to zero when declared and incremented in the constructor). These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of Pet. These are the only fields that this class should have.

  (2) **Constructor**: The Pet class must contain a constructor that accepts five parameters representing the values to be assigned to the *instance* fields: owner, name, breed, weight, and days to be boarded. Since this class is abstract, the constructor will be called from the subclasses of Pet using *super* and the parameter list. The count field should be incremented in this 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.
  - `getOwner`: Accepts no parameters and returns a String representing the owner.
  - `setOwner`: Accepts a String representing the owner, 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.
  - `getBreed`: Accepts no parameters and returns a String representing the breed.
  - `setBreed`: Accepts a String representing the breed, sets the field, and returns nothing.
  - `getWeight`: Accepts no parameters and returns a double representing weight.
  - `setWeight`: Accepts a double representing the weight, sets the field, and returns nothing.
  - `getDays`: Accepts no parameters and returns a int representing days.
  - `setDays`: Accepts an int representing the days, 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.

- `toString`: Returns a String describing the Pet object. This method will be inherited by the subclasses, and a subclass may override the inherited toString method as needed. Items on each line are separated by three spaces. The double value for the boarding cost should be formatted (`"$#,##0.00"`), but the other numeric values do not require formatting. For an example of the toString result, see the Cat class below.

  Note that you can get the class name for the instance by calling this.getClass().toString().substring(6). For the example Cat c, this.getClass().toString() returns "class Cat" and substring(6) extracts the class name "Cat" which begins at character 6. This approach allows the toString method in Pet to work for all subclasses that inherit the toString method.
- `boardingCost`: An *abstract* method that accepts no parameters and returns a double representing the boarding cost for a pet. Since this is an abstract method, it has no body in Pet; however, each non-abstract subclass must implement this method.

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

- **Cat.java**

  Requirements: Derive the class Cat from Pet.

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

  (1) **Fields**: an *instance* variable for number of lives left of type int, which is declared with the *private* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 10. These are the only two fields that should be declared in this class.

  (2) **Constructor**: The Cat class must contain a constructor that accepts six parameters representing the five instance fields in the Pet class (owner, name, breed, weight, and days) and the one instance field liveLeft declared in Cat. Since this class is a subclass of Pet, the super constructor should be called with field values for Pet. The instance variable livesLeft should be set with the last parameter. Below is an example of how the constructor could be used to create a Cat object:

  ```
  Cat c = new Cat("Barb Jones", "Callie", "Siamese", 9.0, 7, 9);
  ```

  (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.
  - `getLivesLeft`: Accepts no parameters and returns an int representing livesLeft.
  - `setLivesLeft`: Accepts an int for livesLeft, sets the field, and returns nothing.

- `boardingCost`: Accepts no parameters and returns a double representing the boarding cost for the cat calculated as follows (note, that boarding cost for a cat increases significantly as livesLeft decreases below 9; e.g., the cost triples if livesLeft is only 3):
  (BASE_RATE + weight * 0.10) * days * (9.0 / livesLeft)
- `toString`: Calls the toString method in the parent, super.toString(), and then appends the livesLeft info to it. Below is an example of the toString result for Cat c as it is declared above. Note that "Cat" at the beginning of second line should be determined by calling this.getClass() in the toString method in Pet.

```
Owner: Barb Jones   Pet: Callie   Days: 7   Boarding Cost: $76.30
Cat: Siamese   Weight: 9.0 lbs   Lives Left: 9
```

Code and Test: As you implement the Cat 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 Pet (parent class of Cat). The test methods in CatTest should be used to test the methods in both Pet and Cat. Remember, Cat *is-a* Pet which means Cat inherited the instance methods defined in Pet. Therefore, you can create instances of Cat in order to test methods of the Pet class. You may also consider developing PetBoardingPart1 (page 7) in parallel with this class to aid in testing.

- **Dog.java**

Requirements: Derive the class Dog from Pet.

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

(1) **Field**: a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 12. This is the only field that should be declared in this class.

(2) **Constructor**: The Dog class must contain a constructor that accepts five parameters representing the five instance fields in the Pet class (owner, name, breed, weight, and days). Since this class is a subclass of Pet, the super constructor should be called with field values for Pet. Below is an example of how the constructor could be used to create a Dog object:

```
Dog d = new Dog("Jake Smith", "Honey", "Great Dane", 60, 7);
```

(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.

- `boardingCost`: Accepts no parameters and returns a double representing the boarding cost for a dog calculated as follows:
  (BASE_RATE + weight * 0.05) * days
- `toString`: NONE. When toString is invoked on an instance of Dog, the toString method inherited from Pet is called. Below is an example of the toString result for Dog d as it is declared above.

```
Owner: Jake Smith   Pet: Honey   Days: 7   Boarding Cost: $105.00
Dog: Great Dane   Weight: 60.0 lbs
```

Code and Test: As you implement the Dog 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 as you have implemented and successfully compiled the constructor, you should create an instance of Dog in a JUnit test method in the DogTest class and then run the test file. If you want to view your objects in the Canvas, set a breakpoint in your test method and then 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 toString 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 PetBoardingPart1 (page 7) in parallel with this class to aid in testing.

- **ServiceDog.java**

Requirements: Derive the class ServiceDog from Dog.

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

(1) **Fields**: *instance* variables for service of type String and commands of type String [], which should be declared with the *private* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 13. These are the only fields that should be declared in this class.

(2) **Constructor**: The ServiceDog class must contain a constructor that accepts seven parameters representing the five values for the instance fields in the Pet class (owner, name, breed, weight, and days) and two for the instance fields declared in ServiceDog. Since this class is a subclass of Dog, the super constructor should be called with five values for Dog. The instance variables for service and commands should be set with the last two parameters. Below is an example of how the constructor could be used to create a ServiceDog object. Note that the arguments "sit", "down", "stay", "come", "around", "forward", "right", "left" are matched with commands, which is a variable length parameter in the constructor header (i.e., `String ... commandsIn`) and then simply a String[] in the constructor body.

```
ServiceDog d2 = new ServiceDog("Jen Baker", "Pepper", "Sheppard", 60, 7,
                               "guide dog",
                               "sit", "down", "stay", "come", "around",
                               "forward", "right", "left");
```

(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.

- `getService`: Accepts no parameters and returns a String representing service.
- `setService`: Accepts a String representing the service, sets the field, and returns nothing.
- `getCommands`: Accepts no parameters and returns a String[] representing commands.
- `setCommands`: Accepts a variable length parameter of type String (i.e., `String ... commandsIn`) representing commands, sets the field, and returns nothing.
- `boardingCost`: Accepts no parameters and returns a double representing the boarding cost for a service dog calculated as follows:
  (BASE_RATE + weight * 0.05 + commands.length) * days
- `toString`: Calls the toString method in the parent, super.toString(), and then appends the service info and commands info (if commands.length is greater than 0) to it. Below is an example of the toString result for ServiceDog d2 as it is declared above. If d2 had been created with no commands, the last line would be omitted.

```
Owner: Jen Baker   Pet: Pepper   Days: 7   Boarding Cost: $168.00
ServiceDog: Sheppard   Weight: 60.0 lbs   Service: guide dog
Commands: sit down stay come around forward right left
```

Code and Test: As you implement the ServiceDog class, you should compile and test it as methods are created. For details, see **Code and Test** above for the Cat and Dog classes. You may also consider developing PetBoardingPart1 (page 7) in parallel with this class to aid in testing.

- **Horse.java**

Requirements: Derive the class Horse from class Pet.

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

(1) **Field**: *instance* variable for exerciseFee of type double, which should be declared with the *private* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public, static,* and *final* modifiers and initialized to 15. These are the only fields that should be declared in this class.

(2) **Constructor**: The Horse class must contain a constructor that accepts six parameters representing the five values for the instance fields in the Pet class (owner, name, breed, weight, and days) and one for the instance variable exerciseFee in Horse. Since this class is a subclass of Pet, the super constructor should be called with five values for the Pet constructor. The instance variable exerciseFee should be set with the last parameter. Below is an example of how the constructor could be used to create a Horse object:

```
Horse h = new Horse("Jessie Rider", "King", "Quarter Horse", 1000, 7, 10.0);
```

(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.

- `getExerciseFee`: Accepts no parameters and returns a double representing exerciseFee.
- `setExerciseFee`: Accepts a double representing exerciseFee, sets the field, and returns nothing.
- `boardingCost`: Accepts no parameters and returns a double representing the boarding cost for the horse calculated as follows:
  (BASE_RATE + weight * 0.01 + exerciseFee) * days
- `toString`: Calls the toString method in the parent, super.toString(), and then appends the exerciseFee info to it. Below is an example of the toString result for Horse h as it is declared above. Note that "Horse" at the beginning of second line should be determined by calling this.getClass() in the toString method in Pet.

```
Owner: Jessie Rider   Pet: King   Days: 7   Boarding Cost: $245.00
Horse: Quarter Horse   Weight: 1000.0 lbs   Exercise Fee: $10.00
```

Code and Test: As you implement the Horse class, you should compile and test it as methods are created. For details, see **Code and Test** above for the Cat and Dog classes. You may also consider developing PetBoardingPart1 (below) in parallel with this class to aid in testing.

- **PetBoardingPart1.java (Optional, but strongly recommended)**
  Requirements: This driver class with a main method is optional but you may find it helpful.

  Design: The PetBoardingPart1 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 Pet and Cat, you can add statements to the main method that create and print an instance of Cat. Remember, since Pet is abstract you cannot create an instance of it. When main is completed, it should contain statements that create and print instances of Cat, Dog, ServiceDog, and Horse. Since printing the objects will not show all of the details of the fields, you should also run in canvas (or debug 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 c, d, d2, and h as described in the sections above and your main method is stopped between steps after h has been created, you can enter the following in interactions to get the rating for the Horse object.

```
► h.boardingCost()
  245.0
```

The output from main assuming you create print the four objects c, d, d2, and h as described in the sections above is shown as below. Note that new lines were added by main to achieve the spacing between objects.

```
Owner: Barb Jones   Pet: Callie   Days: 7   Boarding Cost: $76.30
Cat: Siamese   Weight: 9.0 lbs   Lives Left: 9

Owner: Jake Smith   Pet: Honey   Days: 7   Boarding Cost: $105.00
Dog: Great Dane   Weight: 60.0 lbs
```

```
Owner: Jen Baker   Pet: Pepper   Days: 7   Boarding Cost: $168.00
ServiceDog: Sheppard   Weight: 60.0 lbs   Service: guide dog
Commands: sit down stay come around forward right left

Owner: Jessie Rider   Pet: King   Days: 7   Boarding Cost: $245.00
Horse: Quarter Horse   Weight: 1000.0 lbs   Exercise Fee: $10.00
```

Code and Test: After you have implemented the PetBoardingPart1 class, you should create the test file PetBoardingPart1Test.java in the usual way. The only test method you need is one that checks the class variable *count* that was declared in Pet 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 Pet hierarchy). The following statements accomplish the test.

```
// covers the default constructor (Web-CAT will check for this)
PetBoardingPart1 pbp1 = new PetBoardingPart1();

// checks class variable count
Pet.resetCount();
PetBoardingPart1.main(null);
Assert.assertEquals("Pet.count should be 4. ",
                    4, Pet.getCount());
```

Canvas for PetBoardingPart1

Below is an example of a jGRASP viewer canvas for PetBoardingPart1 that contains a viewer for the class variable Pet.count and two viewers for each of c, d, d2, and h. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. Notice that runtime types are shown in the object viewer labels. To turn on this feature, click View on the top menu, then select "Show Runtime Types in Viewer Labels" (you should see a check mark in the associated check box when this is on). 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*.