Deliverables

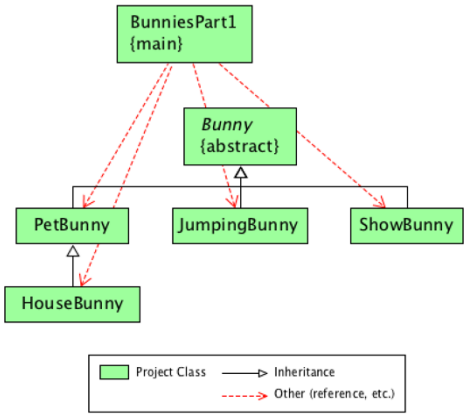
Your project files should be submitted for grading by the due date and time specified. Note that there is also an optional Skeleton Code assignment (ungraded) which will indicate level of coverage your tests have achieved. The files you submit to skeleton code assignment may be incomplete in the sense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

Files to submit to for grading:

- Bunny.java
- PetBunny.java, PetBunnyTest.java
- HouseBunny.java, HouseBunnyTest.java
- JumpingBunny.java, JumpingBunnyTest.java
- ShowBunny.java, ShowBunnyTest.java
- BunniesPart1.java, BunniesPart1Test.java

Specifications

Overview: This project is the first of three parts that will involve calculating the estimated monthly cost of owning a bunny where the amount is based on the type of bunny, its weight, and various additional costs. You will develop Java classes that represent categories of bunny: pet bunny, house bunny (a subclass of pet bunny), jumping bunny, and show bunny. These categories will be implemented as follows: an abstract Bunny class which has three subclasses PetBunny, JumpingBunny, and ShowBunny. The PetBunny class has a subclass HouseBunny. The driver class for this project, BunniesPart1.java, should contain a main method that creates one or more instances of each of the non-abstract classes in the Bunny hierarchy. As you develop each non-abstract class, you should add code in the main method to create and print one or more instances of the class. Thus, after you have created all the classes, your main method should create and print one or more objects (e.g., at least one for each of the types PetBunny, HouseBunny , JumpingBunny, and ShowBunny. You can use BunniesPart1 in conjunction with interactions by running the program in the canvas (or debugger with a breakpoint) and single stepping until the each of the instances is created. You can then enter interactions for the instances in the usual way. However, a more efficient way to test your methods would be to create the JUnit test file (required for this project) for each class and write an appropriate number of test methods to ensure the classes and methods meet the specifications. All of your files should be in a single folder. You should create a jGRASP project upfront and then add the source



and test files as they are created. You should generate (or regenerate) the UML class diagram each time you add a class to the project (see page 8).

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

- **Bunny.java**

  Requirements: Create an *abstract* Bunny class that stores Bunny data and provides methods to access the data. The Bunny class should implement the Comparable interface for Bunny objects.

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

  (1) **Fields**: Three *instance* variables for the Bunny's name of type String, breed of type String, and weight of type double. These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of Bunny. The last field should be a protected *static* variable, bunnyCount, of type int with *protected* access. This class variable is used to track the number of bunnies that are created from the classes in the Bunny hierarchy. These are the only fields that this class should have.

  (2) **Constructor**: The Bunny class has a constructor that accepts three parameters representing the values for the respective *instance* variables, assigns the values, and then increments the bunnyCount class variable. Since this class is abstract, the constructor cannot be called with the *new* operator to create an instance of Bunny. Instead, the constructor will be called from the constructors of the subclasses of Bunny 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.
  - `getName`: Accepts no parameters and returns a String representing the Bunny's owner.
  - `setName:` Accepts a String representing the name, sets the field, and returns nothing.
  - `getBreed`: Accepts no parameters and returns a String representing the breed field.
  - `setBreed:` Accepts a String representing the breed, sets the field, and returns nothing.
  - `getWeight`: Accepts no parameters and returns a double representing the weight.
  - `setWeight:` Accepts a double representing the weight, sets the field, and returns nothing.
  - `getBunnyCount`: Accepts no parameters and returns an int value of bunnyCount. Since bunnyCount is *static*, this method should be *static* as well.
  - `resetBunnyCount:` Accepts no parameters, resets bunnyCount to zero, and returns nothing. Since bunnyCount is *static*, this method should be *static* as well.
  - `estimatedMonthlyCost`: An *abstract* method that accepts no parameters and returns a double representing the estimated monthly cost for the Bunny. Note that this method has no body in Bunny; it will be overridden in its subclasses.

- `toString`: Returns a String describing the Bunny. This method will be inherited by the subclasses although it may be overridden in the subclass as appropriate. If it is overridden, then it should be called from the toString method in the subclasses of Bunny using super.toString(). For more details and an example of the toString result, see the PetBunny class below. The first line and all or part of the second line in each toString result in the subclasses should be produced by this toString method. Note that you can get the class name within the class by calling: `this.getClass().getName()`
- `equals`: Accepts a parameter of type Object and returns false if the Object is a not a Bunny; otherwise, when cast to a Bunny, if it has the same field values (ignoring case in Strings) as the Bunny upon which the method was called, it returns true; otherwise, it returns false. Note that this equals method with parameter type Object will be called by the JUnit Assert.assertEquals method when two Bunny objects are checked for equality.

  Below is a version you are free to use.

```
public boolean equals(Object obj) {

   if (!(obj instanceof Bunny)) {
      return false;
   }
   else {
      Bunny b = (Bunny) obj;
      return (name.equalsIgnoreCase(b.getName())
            && breed.equalsIgnoreCase(b.getBreed())
            && Math.abs(weight - b.getWeight()) < .000001);
   }
}
```

- `hashCode()`: Accepts no parameters and returns zero of type int. This method is required by Checkstyle if the `equals` method above is implemented.
- `compareTo`: Takes a Bunny object as a parameter and returns an int indicating the results of comparing Bunny objects based on their respective names where the natural order is alphabetical (ignoring case). This method is required since the Bunny class implements the Comparable interface for Bunny.

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

- **PetBunny.java**

Requirements: Derive the PetBunny class from the Bunny class.

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

(1) **Fields**: One public constant of type double: `BASE_COST = 1.85`
(Note that a constant must be declared as both *static* and *final*.)
This is the only field that should be declared in this class.

(2) **Constructor**: The PetBunny class must contain a constructor that accepts three parameters representing the four instance fields in the Bunny class. Since this class is a subclass of Bunny, the super constructor should be called with field values for Bunny. Below are examples of how the constructor could be used to create a PetBunny object:

```
PetBunny pb1 = new PetBunny("Floppy", "Holland Lop", 3.5);
```

(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.

- `estimatedMonthlyCost`: Accepts no parameters and returns a double calculated as: `BASE_COST * weight`.
- `toString`: None – this class will use the `toString` method from Bunny. Below is an example based on `pb1` above. Note that Breed and Weight are each preceded by three spaces (not a tab).

```
Floppy (PetBunny)   Breed: Holland Lop   Weight: 3.5
Estimated Monthly Cost: $6.48
```

Code and Test: As you implement the PetBunny 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 Bunny (parent class of PetBunny). The test methods in PetBunnyTest should be used to test the methods in both Bunny and PetBunny. Remember, a PetBunny *is-a* Bunny which means PetBunny inherited the instance methods defined in Bunny. Therefore, you can create instances of PetBunny in order to test methods of the Bunny class. You may also consider developing BunniesPart1 (page 7) in parallel with this class to aid in testing.

- **HouseBunny.java**

Requirements: Derive the HouseBunny class from the PetBunny class.

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

(1) **Field**: an instance variable for wearAndTear of type double, which should be declared with the *private* access modifier; one public constant of type double: `BASE_COST = 2.25`
These are the only fields that should be declared in this class.

Constructor: The HouseBunny class must contain a constructor that accepts four parameters representing the three instance fields in the Bunny class (inherited by PetBunny) and one instance field from HouseBunny. Since this class is a subclass of PetBunny, the super constructor should be called with the values for the inherited fields. The instance variable for

wearAndTear should be set with the last parameter of the HouseBunny constructor. Below is an example of how the constructor could be used to create a HouseBunny object.

```
HouseBunny hb1 = new HouseBunny("Spot", "Really Mixed", 5.8, 0.15);
```

(2) **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.

- `getWearAndTear`: Accepts no parameters and returns a double representing the wearAndTear field.
- `setWearAndTear:` Accepts a double representing the wearAndTear, sets the field, and returns nothing.
- `estimatedMonthlyCost`: Accepts no parameters and returns a double calculated as: `BASE_COST * weight * (1 + wearAndTear).`
- `toString`: The toString method required in the HouseBunny class should invoke the inherited toString method and then append wear and tear info to the result. Below is an example of the toString result for `hb1` as declared above.

```
Spot (HouseBunny)   Breed: Really Mixed   Weight: 5.8
Estimated Monthly Cost: $15.01 (includes 15.0% for wear and tear)
```

Code and Test: As you implement the HouseBunny 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 HouseBunny in a JUnit test method in the HouseBunnyTest 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 BunniesPart1 (page 7) in parallel with this class to aid in testing.

- **JumpingBunny.java**

Requirements: Derive the JumpingBunny class from the Bunny class.

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

(1) **Field**: an instance variable for trainingCost of type double, which should be declared with the *private* access modifier; one public constant of type double: `BASE_COST = 2.50`.

These are the only fields that should be declared in this class.

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

```
JumpingBunny jb1 = new JumpingBunny("Speedy", "English", 6.3, 25.0);
```

(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 method.

- `getTrainingCost`: Accepts no parameters and returns a double representing the trainingCost field.
- `setTrainingCost:` Accepts a double representing the trainingCost, sets the field, and returns nothing.
- `estimatedMonthlyCost`: Accepts no parameters and returns a double calculated as: `BASE_COST * weight + trainingCost`.
- `toString`: The toString method required in the JumpingBunny class should invoke the inherited toString method and then append training cost info to the result. Below is an example of the toString result for `jb1` as declared above.

```
Speedy (JumpingBunny)   Breed: English   Weight: 6.3
Estimated Monthly Cost: $40.75 (includes $25.00 for training)
```

Code and Test: As you implement the JumpingBunny 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 in JumpingBunnyTest. You may also consider developing BunniesPart1 below in parallel with this class to aid in testing. For more details, see **Code and Test** above for the PetBunny and HouseBunny classes.

- **ShowBunny.java**

Requirements: Derive the ShowBunny class from the Bunny class.

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

(1) **Field**: an instance variable for groomingCost of type double, which should be declared with the *private* access modifier; one public constant of type double: `BASE_COST = 2.75`
These are the only fields that should be declared in this class.

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

```
ShowBunny sb1 = new ShowBunny("Bigun", "Flemish Giant", 14.6, 22.0);
```

(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.

- `getGroomingCost`: Accepts no parameters and returns a double representing the groomingCost.
- `setGroomingCost`: Accepts a double representing the groomingCost, sets the field, and returns nothing.
- `estimatedMonthlyCost`: Accepts no parameters and returns a double calculated as: `BASE_COST * weight + groomingCost`.
- `toString`: The toString method required in the ShowBunny class should invoke the inherited toString method and then append grooming cost info to the result. Below is an example of the toString result for `sb1` as declared above.

```
Bigun (ShowBunny)   Breed: Flemish Giant   Weight: 14.6
Estimated Monthly Cost: $62.15 (includes $22.00 for grooming)
```

Code and Test: As you implement the ShowBunny 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 in ShowBunnyTest. You may also consider developing BunniesPart1 below in parallel with this class to aid in testing.

- **BunniesPart1.java**

Requirements: Driver class with main method.

Design: The BunniesPart1 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 Bunny and PetBunny, you can add statements to main that create and print an instance of PetBunny. [Since Bunny is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print one or more instances of PetBunny, HouseBunny, JumpingBunny, and ShowBunny. Since printing the objects does not show the actual fields, you should also run BunniesPart1 in the canvas (or debugger with a breakpoint) to examine the objects while the program is running. Between steps you can use interactions to invoke methods on the objects in the usual way. For example, if you create PetBunny pb1, as described above and your main method is stopped between steps after PetBunny pb1 has been created, you can enter the following in interactions to calculate the estimated monthly cost for the PetBunny object.

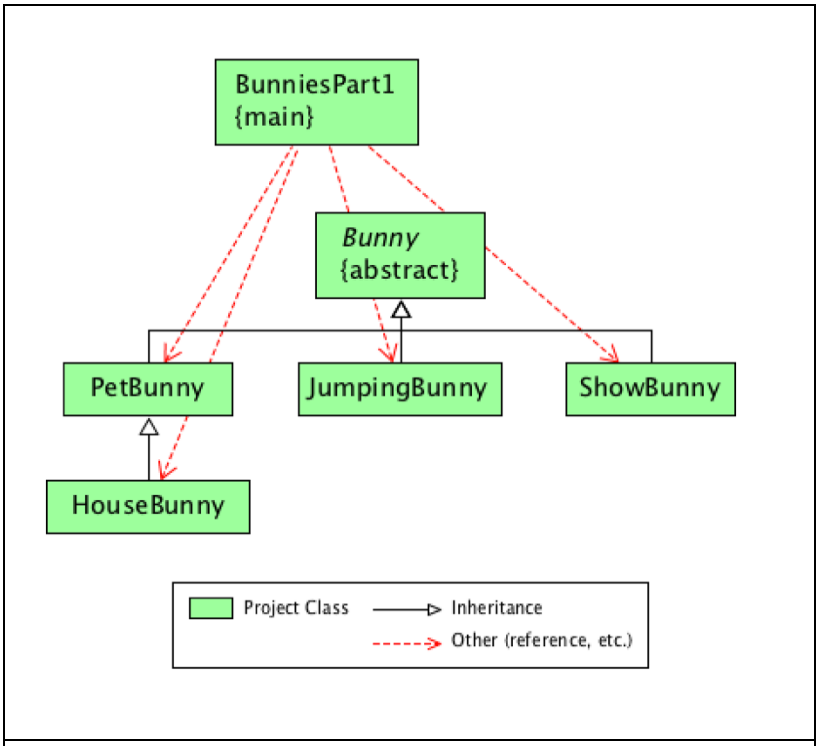
```
pb1.estimatedMonthlyCost()
6.4750000000000005
```

Code and Test: After you have implemented the BunniesPart1 class, you should create the test file BunniesPart1Test.java in the usual way. The only test method you need is one that creates an instance of BunniesPart1 (to cover its default constructor), and then resets the class variable Bunny*Count* that was declared in Bunny and shared by each subclass. In the test method, you should declare and create an instance of BunniesPart1, reset Bunny*Count*, call your main method, then assert that Bunny*Count* is four (assuming that your main creates four objects from the Bunny hierarchy). The following statements accomplish the test.

```
BunniesPart1 vp1 = new BunniesPart1();
Bunny.resetBunnyCount();
BunniesPart1.main(null);
Assert.assertEquals("Bunny.BunnyCount should be 4. ",
                    4, Bunny.getBunnyCount());
```

UML Class Diagram

As you add your classes to the jGRASP project, you should generate the UML class diagram for the project. To layout the UML class diagram, right-click in the UML window and then click **Layout** > **Tree Down**. Click in the background to unselect the classes. You can then select the BunniesPart1 class and move it around as appropriate. Below is an example. Note that the dependencies represented by the red dashed arrows indicate that BunniesPart1 references each of the subclasses of Bunny (i.e., the main method in BunniesPart1creates instances of each subclass and prints then out).



UML Class Diagram for Bunnies Project

Canvas for BunniesPart1

Below is an example of a jGRASP viewer canvas for BunniesPart1 that contains two viewers for each of the variables pb1, hb1, jb1, and sb1. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. A viewer for the class variable Bunny.BunnyCount is near the bottom of the canvas. 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 *BunnyCount*. To display types with the labels, click **View** on the canvas top menu bar then turn on **Show RuntimeTypes in Viewer Labels** with the check box. Since you are printing the objects in main, you can see the toString results in the Run I/O window. So a single canvas with all objects in the Basic viewer may be more useful than the toString viewer.