COMP 2150 – Spring 2014
Homework 3: Inheritance
(Posted Feb. 19 – Due Feb. 26 by 12:40 pm)

Remember that as stated on the syllabus, this assignment should be an individual effort (contact me or visit the Computer Science Learning Center, www.cs.memphis.edu/cslc, if you need help).

Submission: Please zip your source code into a single file (you can zip the entire project folder if you're using BlueJ) and upload it to the proper folder in the eCourseware dropbox at https://elearn.memphis.edu. The dropbox will cut off all submissions after the indicated deadline, so don't wait until the very last minute to submit your work!

Grading: This assignment will be graded by the TA Vincent Nkawu (venkawu@memphis.edu). If you have questions or concerns about your grade, please contact him first. I'll be happy to look over your assignment myself if he's not able to resolve the situation to your satisfaction. Also remember that as stated on the syllabus, all submissions MUST compile and run to receive credit. The TA does not have time to find and correct your syntax errors!

Coding Style: Be sure to follow the good coding practices that were discussed in COMP 1900:

- Use descriptive variable and method names. Avoid using single-character names unless it's very obvious what the variable's being used for (like a loop counter).
- Follow standard Java programming conventions for **variableAndMethodNames**, **ClassNames**, **CONSTANT_NAMES**.
- Consistently indent your code.
- Use comments judiciously. Every source code file should include a comment block at the top that lists your name and the assignment number. Every non-trivial method should have a comment preceding it that specifies what the method does, what its parameters are, and what it returns. For simple methods like accessors or mutators this isn't necessary, but it doesn't hurt to put in a simple comment like `// accessor methods`. I also expect to see comments throughout your code explaining what actions are being taken!

You use your code from Homework 2 to release your game, *Super Happy Fun Cute Creatures World Traveler*, to widespread acclaim. Unfortunately, within only a few months of release your player base is getting restless! They are easily bored and are demanding that you add more depth to the game.

In response, you decide to introduce a new breed of **CuteCreatures** into your game world. These creatures will automatically "evolve" once they reach level 20. (Not every creature can evolve – some will stay in their original form no matter how high they level.) Evolved creatures are similar to regular creatures, but they gain the following extras:

- When the creature evolves, it gains a bonus 15 maximum and current hit points and 5 attack damage. This is on top of the regular gains from leveling.
- Each evolved creature becomes "attuned" to a particular element. This is done based on the first letter of the creature's species:
  - A-G creatures attune to fire
  - H-M creatures attune to water
  - N-S creatures attune to air
  - T-Z creatures attune to earth
- Evolved creatures gain the ability to perform a special elemental attack in addition to their regular attack. This elemental attack is guaranteed never to miss, but it will never score a critical hit either. Additionally, there is no ±20% variance in the attack damage like there is with the regular attack. The damage done by an elemental attack depends on the element of the target creature:

- If the target creature is of the same element as the attacking creature, the elemental attack deals zero damage.
- If the target creature's element resists the attacking creature's element, the elemental attack deals 0.25x normal attack damage. See the table below.
- If the target creature's element is vulnerable to the attacking creature's element, the elemental attack deals 4x normal attack damage. See the table below.
- In all other cases, the elemental attack deals the attacking creature's normal attack damage. This includes situations in which an evolved creature performs an elemental attack vs. a creature with no element (which could be an **EvolvableCuteCreature** who hasn't evolved yet, or just a regular **CuteCreature**).

	Deals 0.25x damage vs.	**Deals 4x damage vs.**	**Deals normal damage vs.**
Fire	Water	Air	Earth
Water	Earth	Fire	Air
Air	Fire	Earth	Water
Earth	Air	Water	Fire

1. **(20 pts)** Write a subclass **EvolvableCuteCreature** that extends **CuteCreature** with the functionality described above. Here are a few guidelines for this subclass:

   - Remember that constructors are not inherited, so you'll need to provide one for the **EvolvableCuteCreature** class. Use **super** to save yourself some coding if possible!
   - Include an extra instance variable to indicate which element the creature is attuned to.
   - Add a new **elementalAttack(CuteCreature c)** method. If an **EvolvableCuteCreature** tries to call this method without having evolved yet, display an appropriate error message.
   - Override the **levelUp()** method from **CuteCreature** to handle the evolution process at level 20. (You should redeclare **levelUp()** as **protected** rather than **private**.)
   - Override the **toString()** method to include which element the creature is attuned to.

2. **(5 pts)** Write a client program that creates some **EvolvableCuteCreature** objects and tests your methods.

3. **(Honors only – point breakdown for honors students is #1: 17 pts, #2: 3 pts, #3: 5 pts)** Java's **Object** class provides a **clone()** method that returns a copy of the calling object. The **clone()** method performs what's known as a *shallow copy* of the calling object. To make a shallow copy of an object **bar** of class **Foo**, here's what happens:

   a. A new instance of **Foo** (call this **barCopy**) is created.
   b. For each of **barCopy**'s instance variables **iv**, an assignment statement is used to copy the corresponding value from **bar** to **barCopy**: `barCopy.iv = bar.iv;`
   c. The object **barCopy** is returned.

   Assume that your **EvolvableCuteCreature** class is cloneable (this means it implements the **Cloneable** interface, as specified in the API – don't worry about exactly what that means yet). Suppose you already have an **EvolvableCuteCreature** object named **e**, and you create a copy named **eCopy** by calling **e.clone()**. If you then make any changes (after the clone operation) to the instance variables of **e**, will those changes also be reflected in **eCopy**? Why or why not? Clearly explain your answer! Would your answer still be the same if you replaced **EvolvableCuteCreature** with the **Deck** or **Hand** classes that we wrote in lecture? Why or why not?