**Animal House**

You're an animal lover and very fond of keeping track of things. Write a program that will keep track of all the animals you have in your house.

All instance variables should be private. Add public getter / setter methods when required. It is considered best practice to only release information to the "public" (clients of your class) as necessary, and "encapsulate" the rest.

1. Begin with a class called Animal. Animal will have the following:
   1. Instance variables String name and int birthYear
   2. An Animal may have arbitrarily many Toys (more info to follow), and should have a method to add a new Toy to its collection.
   3. An Animal may also have a friend of type Animal. The friend variable should initially start null, and a method should be created to set the friend reference.
   4. A static integer variable representing the currentYear.

/\* This variable can reasonably be static, as its value won't change per Animal instance – this characteristic belongs to all Animals \*/

* 1. A public method int getAge() that returns this Animal's age in years, given the currentYear and the Animal object's birthYear (don't worry about the months).

/\* static variables should be accessed through the class. You shouldn't access the current year using *this*, because *this* refers to *this object* \*/

* 1. A public method String toString() that returns a printable String containing this Animal's info, like the following:

Hello, I am <name>. I am <age> years old.

If this Animal has a friend, add the following line:

I have a friend named <friend's name>.

Otherwise, add the line "I have no friends" :( Finally, add the line "I have the following toys: " concatenated with this Animal's toys.

How can you possibly concatenate an ArrayList, you ask?! The ArrayList class has its own (overridden) toString() method. The toString() method is inherited from the Object class and is the method called by Java when a reference type (an object) is printed.

Further, ArrayList uses a call to Toy's (the type parameter) toString() method when displaying the objects it contains. Woah!

[**This**](https://youtu.be/9POYunsdhv0) **video may be useful as an introduction to the Object class and its methods.**

All classes should have a well-written toString() method that allows its objects' state to be printed in a useful manner (rather than something like House@677327b6, which is the format inherited from Object (more info on the Object class in the PPTs!) – type/@ symbol/hex representation of the object's hash code). Very useful when debugging!

Note that, for a variable Animal a, the line System.out.println(a) is no different than the line System.out.println(a.toString()), as the compiler adds the toString() call for you if you omit it (when printing an object).

1. Write two classes Dog and Cat that extend Animal.
   1. Dog should have a boolean goodBoy instance variable and appropriate constructor.
   2. Cat should have an int numLives instance variable. Cat should have two constructors:
      1. Three parameters: String name, int age, and int numLives. Should contain a super() call.
      2. Two parameters: String name and int age.

The two-parameter constructor should utilize the three-parameter constructor (rather than repeating all the code in each) with a call to this(name, age, 9) – in other words, if a Cat object is created without specifying the number of lives it has, it will be given 9 lives by default.

Without a period, this() refers to *this* object's constructor. This concept is referred to as "constructor chaining" – chaining constructors together, rather than repeating all the code in each. Note that this() and super() are mutually exclusive (more info in the PPTs).

* 1. Override the toString() method, adding the information specific to Dogs and Cats. Use super to re-use what is already done in Animal.

1. Write a class called Toy that has the following:
   1. A String name instance variable and applicable constructor.
   2. An overridden toString() method in the form of "A <name>".
   3. An overridden equals() method (also inherited from Object), that returns true if this Toy has the same name as the parameter. Check the powerpoints for info on overriding the equals() method.

/\* Recall that when comparing Strings, you can't check for equivalency with the comparison (==) operator. Instead, you must use the equals() method. When you do this, you are calling the String class' overriden equals() method! \*/

Add the @Override annotation above the method signature as a sanity check – the compiler will warn you if you are not actually overriding a method (examples in the powerpoints).

1. (Riddle) You are going to make a word. To make this word, you need to put a letter in between two e's to make a word that starts with e and ends with e. What is the word?
2. Write a class called House that has the following:
   1. A House instance can have arbitrarily many animals.
   2. A printAnimals() method that prints the info on all the Animals in the house. There should only be one print statement inside the for-each loop.
   3. Add a method to the House class called cleanHouse(). This method will remove any duplicate toys that a *particular Animal* possesses (two *different* Animals can have a similar toy).
      1. Use the ArrayList class' indexOf() and lastIndexOf() methods to check for (then remove) duplicate Toys. These methods expect a Toy parameter and will return the first and last index of where they exist in the list, respectively.

But how does Java know two Toys objects are the same? The == operator only works when two objects *refer to the same object in memory* (not very useful for detecting duplicate toys). Example:

ArrayList<Toy> toys = new ArrayList<>();

toys.add(new Toy("Ball"), new Toy("Bone"), new Toy("Rope"));

out.println(toys.indexOf(new Toy("Rope"))); //won't find, -1

The example above will always print -1. Two different Toy objects are not the same object, even though they may be *equivalent.* So what can you do? The answer is to override the equals() method (inherited from Object)! The equals() method is used by Java to determine equivalence (similar to how the toString() method is used by Java for printing reference types).

Use the following loop as a starting point:

for (Animal a : animals)

{

ArrayList<Toy> toys = a.getToys(); //one Animal's toys each iteration

for (int i = 0; i < toys.size(); i++)

/\* Add your code here \*/

}

If your ArrayList is properly parameterized\* (e.g. ArrayList**<Toy>** toys), the indexOf() and lastIndexOf() methods will use a call to Toy's overridden equals() method to determine if a particular object exists in the list. In other words, the indexOf() method returns the index where a call to the overridden equals() method for the parameter object returns true (an equivalent object). Gaaaaaah!

*\*If your ArrayList WASN'T parameterized (e.g. ArrayList toys), list elements would be type Object, and Object's equals() method would be called (which simply uses the == operator to check if they are the same object), rather than Toy's overridden method.*

1. A Runner class with a main() method has been provided. Done properly, your output should look like the output below. Fix any mistakes you may have.

Same toy? false

Equal toys? true

Hello, I am Sparky. I am 12 years old.

I have a friend named Fluffy

I have the following toys: [A Busy bee, A Bone, A Ball]

I am a good boy.

Hello, I am Fluffy. I am 9 years old.

I have a friend named Sparky

I have the following toys: [A Mouse]

I have 9 lives.