ask if they want dog or cat class

topics for both days

* classes
* designing - good practices
* inheritance, subclasses and interfaces
* public, private, protected, (and blank)
* putting classes in arraylist
* polymorphism
* -static methods
* abstract classes & interfaces
* -generics
* final
* constructors
* -equals
* instanceof
* toString

Review class terminology:

* class vs object

A **class** is data type that may contain multiple pieces of data, along with functions that act on that data. The pieces of data are all usually related to each other and represent a single concept in your program.

An object is an **instance** of a class. In other words, a class is a template for creating objects.

Using a real-world analogy, (we all met your pets yesterday) the concept of a Dog or a Cat is a class. Individual dogs and cats in the world are your objects.

* Members of a class

The variables and functions that are associated with a class are designed to group together the STATE and BEHAVIOR of its associated objects.

State = the inherent properties of an object. For a dog, this might be the dogs name or its age. [Ask class for others]

Behavior = these are things that a dog can inherently do. For example, a dog can bark. [What are other behaviors]

* Mention the STATE of an object is stored in variables — these are referred to as INSTANCE VARIABLES or sometimes FIELDS.
* Mention the BEHAVIORS of an object are stored in functions - in OOP, the functions that are members of a class that operate on the instance variables are called METHODS or INSTANCE METHODS.
* Start designing class in IntelliJ.
  + Make Dog class.
  + Add at least two fields, say name, age. (make public for now)
  + Add speak() and and eat().
  + Make DogDemo class, with two dogs. Set each dog's name and age, then call speak and eat.
  + Make toString().
* Introduce public/private/(protected)
  + Public - visible to everyone (for instance variables, anyone access/change it; for methods, anyone can call it).
  + Private - visible only to objects of this exact class. (for instance variables, only code that lives inside this object can see/change the variable; for methods, only code that lives inside can call it).
  + Protected - see later.
* Good practice for public/private.
  + Instance variables - your default choice for these should be private. The reason is that most of the time, all the possible values for these variables are not valid for the class. Or there's some additional logic that has to happen when these variables change. Consider dog age. You don't want someone setting this to be negative.
    - Exception - tiny classes that serve really only to group related variables together. Example would be a Point class that stores a point in 2-d space. You'd have x and y, and they could be public if you want. Rationale - every
  + Methods - default for these is most of the time public. The only exception is if you need a helper function for something in your class that you don't want anyone outside the class to know about.
* So now set name and age to private.
* But now the program doesn't run!
* Introduce **getters** and **setters**.
  + Do the getter and setter for age.
  + Do getter for name.
  + Does setName() really make sense? Usually a dog's name doesn't change over their lifetime. It would be nice if we had a way to set a dog's name ONCE, when we first create the object, THEN, say it never changes.
  + For this, we're going to review constructors.
* Introduce constructors & final & concept of immutability.
  + **Constructor** = special method that runs when an OBJECT is first created.
  + Typically it is used to initialize the object into a usable/ready-to-go state.
  + Write a default constructor (no-arg).
  + Write a constructor that takes name.
  + Mark name as **final**. Why bother? True that the lack of a setName() function effectively makes it so we can't change the name....or does it? We can still change it from inside the class. So setting it as final signals this really is a permanent value about this object that doesn't change over its lifetime.
  + Make second constructor with age.
* Java References.
  + So when you create an object (sometimes called instantiating an object), you get a reference to this new object. For those who have taken C++, Java references are a lot like C++ pointers, only safer, in that Java won't let you do dangerous things with references that you can do with C++ pointers.
  + Basically, in Java, when we use the = sign with PRIMITIVE types (int, double, char, boolean, anything starting with a lowercase letter), the data type is copied.
    - x = 5
    - y = 10
    - x = y [copies 10 into x]
    - x++
    - System.out.println(x, y) // prints 11, 10 → clearly x and y are still separate integers
    - d1 = new Dog("Fido", 3)
    - d2 = new Dog("Rufus, 4)
    - print both
    - d1 = d2
    - print both
    - d2.setAge(5)
    - print both.
  + DRAW DIAGRAM. Emphasize that we are sharing one object from two different references. And this is totally fine and very common, you just have to remember if you change one, then you change the other.
  + This is why immutability is so important. Imagine if both name and age were final. Then the entire object is immutable. Once you create a dog, it never changes. (If only all our dogs could stay puppies forever, right?) But then you don't run into this problem where changing one object seems to change another object (but actually the same object) in a different place in your program.
    - There are other benefits too, such as when you write concurrent programs, programs where you have two separate parts that are running simultaneously
  + Many objects in Java (and other languages) are immutable by design. Consider Strings in Java (and Python). When you want to change a letter in a string, you have to make a new string. If you change a string to uppercase/lowercase, or remove some letters from a string, you must make a new string.
  + By default, when making a new class, consider if the class can be immutable. It makes thinking about objects a lot easier if you can, because things can't accidentally change.

Stopped roughly here on end of day 1. Didn't really finish immutability. Start day 2 with warmup on references.

* Day 2

Day 2

* Warmup on references
* Let's write a method that allows one dog to play with another dog.
  + Writing play(Dog otherdog)
  + One dog plays with another dog.
  + Notice how we're allowed to access both dog's private variables, or you can use getName().

INHERITANCE

* Make another class, say SledDog, with an int energy and a method pullSled().
* pullSled decreases energy. and prints name + pulls sled. notice how we can't access name, we must use getName.
* Constructor for SledDog, call superclass constructor.
  + Constructors are not inherited.
* Notice how we can't access name or age directly from parent class.

ABSTRACT CLASS/INTERFACE

* Make pet class, dog/cat inherit from it.
  + Pet should have getname, setname, getage. But what about speak()? Each type of Pet makes different sounds, so to write this method, we'd have to know what all the different pets are.
  + Instead, mark it as abstract.
* So now we have Dog class, Cat class. Both inherit from Pet. We also have SledDog.
* Draw hierarchy of these.
* Make a Shelter program.
* Make 5 pets – 2 dogs, 1 cat, 1 sleddog, then another dog.
* Put them all in arraylist.
* Print the arraylist using foreach iterator.  
  + Write toSTring method.
  + Call speak inside each one.
  + Point out how the CORRECT speak is called, even though Pet.speak() is abstract.
* Show how IntelliJ can generate it for you!
  + Add in getClass().getName()
* Have dogs play with the one next to it.
  + Illustrate casting and instanceof.
* Mention this is polymorphism. Here, polymorphism means the program works correctly even if a subtype is passed in.

INTERFACES

* Make a Toy interface.
  + only has getName()
* Give pet "playswith(Toy t)" prints name plays with toy getName.
  + Note we can write all of this even before any toys exist!
* Make a toy ball class.
  + private string color.
  + Constructor with color.
  + Put a ball in the shelter. Everyone plays with it.

STATIC

* Say we want to always know how many dogs we have in our program.
* In Dog class, add "private static int totalDogs."
* Make method to retrieve this.
  + Put it after we've created the dogs. Call it on a dog instance. (yes, that way)
  + But what if we want to call this BEFORE we've created any dogs.
  + Final output should show ZERO dogs, then four dogs.
* So what static says is **this method or field is associated with the class, not one particular object of the class**.
* The other place you see static is of course public static void main. Because again, that main method is not associated with any object of the class. We never even instantiated that class.
* So another common place you see static functions

EQUALS

* Create two dogs with same name and age.
* == says they are not equal.
* equals() can.
* Pattern for writing equals.
  + Must take Object.
  + 3 sections
  + if obj == this return true
  + if not instanceof return false
  + cast other object to this class and compare fields.
* Mention this for comparing strings.

MORE TIME – JAVA API

* look up arraylist, string