Creating classes (second part of OOP)

* Intro (don't make this too long)
* Review terminology for today:
  + Class = the data type. A template or blueprint for creating objects.
  + Object = a particular "value" of the class with its own identity, distinct from all the other objects of the class. E.g., There is only one Color class, but there can be many objects of the class.
    - Sometimes you will hear "instance" or "instance of a class" as a synonym for an object.
    - Analogy = class is like a recipe, and objects/instances are the foods you make from that recipe.
  + Reference (to an object): A data type that lets a variable "point to" an object in memory. All objects in Java must be accessed through a reference.
  + Instantiation = creating an object with the keyword "new".
  + Methods of a class (particularly "instance methods"): functions that can be called ("invoked") on objects of that class.
  + Constructor = a special method that is called when an object is instantiated, usually responsible for initializing the instance variables.
* Start designing Dog class. Remind them that dogs (as a class) will have STATE/BEHAVIOR; those are the pieces we care about. (nouns/verbs) Make a list:
  + Dog's name (string), age, weight (int), energy.
  + Eat, sleep, bark, etc.
  + Begin writing code for Dog class. First add 3 instance vars for String name, int age. Don't add anything else.
  + Write a DogDemo class. Explain that we usually separate the code for a class definition from any other programs that **use** the class (with main methods). Note that Dog has no main method, so it can't be run.
  + Create a dog object. Show how to set the variables for the dog and print them back out. Draw picture.
  + Create a second dog object. Show setting/printing the variables, and **how each dog object has its own copies of the variables**. Draw picture.
  + Create a third dog **REFERENCE**, but point it towards one of the existing dogs. Draw picture. Show how we can re-direct the REFERENCE and reference other dogs.
  + Write a method called speak. (don't use the dog's name yet). Just print woof woof. Call the function speak() in DogDemo for each dog.
    - Now add the dog's name to the speak method. Explain how you can **refer to instance variables in these methods** and they will always refer to the version of the variable that belongs to the object the method was called on.
  + Write and explain toString: one of the most useful methods. Show it working.
  + Ask them to add an instance variable weight to the class. Then change the speak() function so if the dog is 20 or fewer pounds, print the dog "yips". If it is 20-60, the dog barks, and above 60, it woofs.
  + Ask them to add energy field. The dog can only bark if it has a positive amount of energy, and every time they bark, the energy decreases by 1. They can recover energy by sleeping, which adds 1 unit of energy.
  + Add a function eat, which takes a parameter called how much. Recovers energy.
  + Add constructor.
  + Explain public/private.
  + chase(otherdog)

DAY 2. (from day 1: did not get to constructor or public/private, or chase). Barely got through speak.   
FOR F21 – Start with info re project.

* Start by reviewing speak. Go over how we can make speak() use the weight of the dog to say yip or bark or woof, depending on how big the dog is.
* Talk about "overloading" methods---multiple methods with the same name, that take different number or types or arguments.
* Talk about public/private (they saw this in the previous lab). Public = the var or method is only accessible within its own class. Private = the var or method is accessible anywhere (inside and outside its class).  
  + Why have public and private?
  + A few reasons. First one from lab day: Private is often used on instance variables to prevent someone using your class (you or someone else) from accidentally setting a variable to a value that makes no sense (e.g., negative #s for weight or age, in the self-driving car lab, a direction other than NESW, etc).
  + [Bring up driving lab, show private draw function for racetrack]. Contrast with startRace() which is public. Why is one private and one public? Because draw() which is private, is not intended to be used outside of the class. We make startRace public because we specifically want people who are using our racetrack class to call startRace
  + FOR LATER: Can use nim game as example. Can use 3 sep variables, or an int[] array for the sticks. NimBoard is the class. supports method for isGameOver() RemoveSticks().  
    Or 12hr vs 24hr clock.
  + Rule of thumb: make instance variables private by default unless you have a good reason. For methods – ask yourself if someone outside of the class should be able to call this method. If so, make public. Otherwise – normally this is a helper method – make private.
* Talk about constructors.  
  + Constructor: special method that is automatically called when an object is first instantiated (constructed). Triggered by the "new" keyword.
  + Constructors always have the same name as the class itself, and **have no return datatype** in their definition. This is how you recognize constructors in the class definition.
  + Most common use of a constructor is to initialize all the instance variables of a class and "get it ready for use."
  + Let's write a constructor for our dog class. (Default constructor).
  + Let's write a second constructor. Many times, classes will have a default constructor that takes no arguments, that initializes all the instance vars to default values. And then they will have other constructors that take parameters if users want to initialize the variables to other things.
* Exercises:

To your dog class, add the ability for the dog to have some amount of energy. The dog's energy can never go below zero. • Edit print() so it displays energy as well. • Add a getter and a setter called getEnergy() and setEnergy(int newEnergy). Test your code. • Add a method for the dog to playFetch(). Playing fetch tires the dog out, so it lowers the dog’s energy by 1. Test your code. • Add a method for the dog to sleep for a certain number of hours. The dog's energy should be raised proportionally to the number of hours it sleeps. Test your code.

* Passing objects to internal methods
  + write chase(dog otherdog):
  + show how you can access the otherdog's variables.
  + lower the energy of both dogs.
  + explain "this" variable.
  + Note how we are still obeying the contract of public/private. We are "inside" the dog class, so we can access all of dog's private variables --- clearly this extends to other dog objects that we have access to.
* Arrays of objects.
  + Make an array of dogs.
  + Print them all out.
    - Explain the syntax array[0].whatever()
    - emphasize that the dot operator works ANYTIME the thing on the left is a reference to an object. So you can chain method calls. -> Use example from lab -> in Car.drive(), leftOfMe.getCar().getColor().getRed()
  + Get their average age/weight.
* Optional – debugger
* static
  + Static can be assigned to a variable or a method.
    - When assigned to a variable, the variable is no longer called an instance variable, it's called a CLASS VARIABLE. And unlike instance variables, each object does not have its own version of class variables. Instead, there is one class variable for the whole class, and all objects can access it.
  + create numberOfDogsCreated static variable. (make public).
  + Show how we can access this variable inside or outside the class (outside by prefacing with name of the class).
  + But now anyone can change this variable, which means we can't trust it to actually hold the number of dogs (because someone might change it).
  + So let's make it private.
  + But now how do we access it?
    - getNumberOfDogs(): first declare non-static. Illustrate using this.
    - But what if we wanted to use this before we even make any dog objects?
    - Static methods can be called on the name of the class itself (and this is the preferred way to use static methods).
    - Rule of thumb: always access static methods and static (class) variables with the name of the class, not a particular object. Reason: because if you use the name of the object, it's easy to get confused and think that the variable has something to do with the object.
  + Summary of static:
    - Non-static variable -> instance variable -> one copy of the variable per object created.
    - Static variable -> class variable -> there is just one variable for the class itself, and all objects can access it.
    - Non-static method -> instance method -> can access instance variables or class variables (static or non-static). Can call other instance methods or static methods.
    - Static methods -> cannot access instance variables or instance methods. Can only access static variables and other static methods.
  + How do you decide whether a variable or method should be static or non-static?
    - For vars: does it make sense for every object to have its own version of this variable (implies non-static) or is this variable a property of the class as a whole (static)?
    - For methods: does this method depend on accessing any instance variables? If so, it must be an instance method (non-static). If not, you can make it static.
  + Common uses for static:
    - A collection of static methods all centered around a common theme (e.g., Math).
    - Constants with a class (Math.PI) (Color.BLUE). (Things that act like const: System.in)
* Review entire lab, because now we know everything!
* Good class design.
  + During this discussion, we can start introducing the idea of how there can be more than one way to design a class.
  + Inspired by the question of public/private: why make instance variables private if all values are legal? The reason I gave them before is we don't want ppl mucking with our variables, but what if all values are legal?
  + Introduce the idea of INTERFACE vs IMPLEMENTATION. INTERFACE: the parts of the class that we let the user see. So this would automatically include anything public (vars/methods). IMPLEMENTATION=inner workings of the class, including anything private, plus the code for public methods (since usually the user doesn't see the code in the API, just the method "signatures")  
      
    When we design a class, we must think about what we want to make public and what we want to keep private.
  + *The less you make public, the easier it is to switch up the way the class works (IMPL) later*.
  + Car analogy: we have a gas pedal and a brake pedal. These are part of the car's **interface**. You can think of them like public methods. We know when you push the gas, the car goes faster, and when you push the brake, the car slows down.
  + Now the **implementation** of the gas and the brake pedal we don't know much about. In fact, the implementation is entirely different for a traditional gas-powered car, vs a hybrid car, vs a totally electric car, right? So those implementations are kept private, because it allows a car to evolve over time and the implementations can change behind the scenes, but the interface remains constant.
  + So here's a more concrete example in Java. Suppose you are designing a class to represent a Rectangle that you might draw on a simple canvas. What information about a rectangle do we need to know about a rectangle to draw it on a screen?
  + Two ways: (at least): top left corner x & y, plus width & height.
    - or top left corner x & y, lower right corner x & y.
    - or any other corner x & y, plus opposite corner x & y
    - or any corner, plus width and height.
  + Suppose we pick top left x & y, plus width and height.
    - 4 public instance variables (x, y, width, and height)
    - suppose we also make public methods to get lower right x and lower right y coordinates.
  + Now a friend writes a class and for their implementation, they pick top left x and y plus lower right x and y.
    - 4 public instance variables (topleftx, toplefty, bottomrightx, bottomrighty).
    - and we'll make public methods for getWidth() and getHeight().
  + So the problem is now what if I design my program to use my first class, but then I want to switch my code to use the 2nd class. Maybe I want to do this because the 2nd class included some extra methods I want to use or something like that.  
    - It would be really hard to switch now, because my class uses an entirely different INTERFACE than the 2nd class. Entirely different public variables and public methods.
    - So, better idea: **both** classes keep all the variables private. Each class will have 6 public methods, called getTopLeftX/Y, getBottomRightX/Y, getWidth, getHeight. This is our new interface, which is identical in both classes. The 6 methods will be written in different ways depending on which set of private variables I use, so the IMPLs will be different.
    - But the point is the INTERFACE is the same in both, so any code that uses either Rectangle class can switch to the other one very easily.

Design fraction class.

1. Create a new class called Fraction. Add appropriate private instance variables.
2. Write a constructor that takes a numerator and denominator.
3. Write a toString() method. Now write some tests that construct some sample fractions and print them out.
4. Write a method for multiplying "the current fraction" by another fraction. Write tests.
5. Write a method for multiplying "the current fraction" to another fraction. Write tests.
6. Write subtraction and division. Write tests.
7. Write negate. Write tests.