* OOP is meant to reduce complexity of problems
* If we reduce the mental complexity of code we increase the complexity of the programs we can create
* OOP places related code in the same place to limit interdependencies, keeps code DRY, and makes code more intuitive
* What is OOP?
  + Opposite of procedural/functional code
  + Procedural code (antonym of OOP): functions operating on data, emphasizes step by step instructions
  + OOP: functions and data are combined into object, emphasizing interactions between objects
* Objects have data and methods in the form of variables/methods respectively
* API = set of public methods of how we can interact with a given object
* A Class is the blueprint for an object
* An object is an INSTANCE of a class
* In Ruby, objects are instantiated using ::new (class method), which calls #initialize(instance method)
* UML
  + “Unified Modeling Language”
  + Used to diagram classes

**THE 4 PRINCIPLES OF OOP**

Encapsulation:

* All the data and logic for an object work should be encapsulated in that object
* No one else should have access to these things
* If we want people to access data/method in our object, we create a public-facing *interface*
* Interface: signature of methods, arguments, argument types, and return types
  + Allows us to interact with the underlying data of the object
  + When designing interfaces, there’s a common pitfall: classes/objects that are too tightly “coupled”
    - Coupling = too niche an object where it’s sole purpose is to supplement another object
* Private methods have no access frmo outside, also prevents calling with **explicit receiver (other\_pup.name)** **(except writers)**
* **Protected = private but you can call with explicit receiver**
* **Protected and private only apply to instance methods**

Abstraction

* **Don’t need to know implementation to interact with interface**
  + Car example - can interface with car without understanding implementation
* Abstraction vs. Encapsulation
  + Complementary but not the same
    - Encaps = each class should be self-contained
    - Abstraction: I shouldn’t have to think about HOW a class works, to interface with it

Polymorphism

* If two objects have the same interface, then they are indistinguishable
* If it looks like a duck and quacks like a duck, then it’s a duck
* Apply polymorphism to make each class of animal sound out with same method name to simplify roll call method

Inheritance

* We copy pasted lots of puppy into kitten class, means we can DRY
* You re-use class implementations
* Overwrite or exend this implementation in the subclass.
* Children have same interface as Parent
* Can call ::ancestors on a class to see their ancestors (String.ancestors)
* Class SubClass < SuperClass, have to require\_relative file that contains SuperClass in SubClass file
* Child classes should add to but not change parent interface
* Using keyword **super** it invokes parent’s overridden definition of a method using **super**.
  + With arguments calls those arg values into super class’ version of the method
    - Pass the arguments needed by the parent class!
      * Think
  + Super without argmuents implicitly uses those passed into sub class’ version of the method
  + super() to invoke the parent method with no arguments

Modules

* In Ruby, a single class can only have a single pairing
* A way around this is using modules/mixins
* Mixins are modules (e.g. Enumerable, Comparable, Kernel)
* You can have multiple mixins in a single class
* You can define methods and use self and @instance\_variables jus like when defining a class
* ::include
  + Adds a module’s methods to a class as instance methods
* ::extend
  + This command adds a module’s methods to a class as *class* methods
* Mixins vs inheritance
  + Inheritance adds both class and instance methods
  + Mixins only add one or the other
  + Inheritance is used for objects that relate to each other by type
    - E.g. a Toyota is a type of car
  + Mixins are used to add functionality to different kinds of things
    - E.g. A dragon with wings and a flyingBroomStick are both FLyable
* Each and Enumerables
  + If i ::include Enumerables
    - Need to include an each function that takes in a block
* How to make your own module
  + Module Cuddleable
    - Def cuddle
      * Puts “#{name} is so cute!”
    - End
    - Even tho name doesnt mean anything in module
    - When you include Cuddleable in puppy class, it’ll mean something
    - If there is no name method you can define a name method in the module and raise an error saing to mplent a name method.
      * THE INSTANCE METHOD WILL TAKE PRECEDENCE
      * BUT module methods takes precedence over inherited parent methods, so have to define somethin in body of class to take precedence over module  
        (def name   
        super  
         end)

Important OOP Patterns for today

* Null Object Pattern
  + Stand in for a null value that maintains a required interface
  + Avoids NoMethodErrors
  + Maintains the interface but doesn’t do anything
  + In animal farm, create null animal class where all attributes are nil
* Singleton
  + It is a module
  + Need to require ‘singleton’ at the top
  + By including this module in a class, we disable the new class method
    - INSTEAD USE .instance
      * Use to create the singular 1 instance provided by Singleton
  + Makes it so there is only one instance of that class
  + Can create ONE instance of a null object that you populate n times

**OOP TIPS**

* Break things into parts
  + Nouns could be objects
  + Verbs could be methods
* Single Responsiblity Principle
  + A class does only one thing
  + Beware of the “God Object”
  + UML only defines interface
    - Not describing implementation