Lecture 1 : What is an Object?

- **2 Rules of Smalltalk**
  - *Everything* is an object
  - Objects respond only to messages
  - Ex: Automobile object
    - Variables: velocity, weight, and color
    - Methods: accelerate and decelerate
- **What's special about an Object?**
  - Objects contain both *state* and *behavior* and communicate with one another via *messages*.
  - Automobile's velocity variable is changed by accelerate method
  - An application is a group of objects interacting in a coordinate fashion
  - Stop light application manages many Automobile objects
- **OO versus Procedural Approach to programming**
  - Aspects of Procedural Approach
    - Behavior is vested in the procedures
    - Procedures must know data structures
    - Procedures communicate only via data
  - Procedural approach places too much emphasis on data, rather than the behavior of the application.
  - Ex: Baker Procedural approach to baking cookies
    - Has 2 structures: Baker structure and cookie structure
    - Steps:
      - Make pointer to a cookie struct
      - Call bakeCookies(Baker, cookie), returns pointer to cookies
    - Review aspects of Procedural Approach
    - Aspects of OO Approach
      - An application is a set of objects communicating via messages
      - An Object's functionality is described by its methods
      - Data required to support an object's functionality is stored in private variables
    - Examples:
      - Baker Object
        - State
          - Weight
          - Height
          - Name
        - Method
          - `bakeCake()`
          - `bakeCookies()`
      - Kitchen Application
        - Objects
          - Baker
          - Chef
          - Dishwasher
          - Oven
        - Procedural Approach
          - Treat each object as a data structure. Each object must have its own data structure & variables.
          - Write a function `wash()`. Note that 4 different functions must be written
        - OO Approach
          - Treat each object as object. Objects can inherit variables form each other.
          - Write a method `wash()` that operates for all objects. (show in Smalltalk & C)

- Good Exercise for students: Use polymorphism for one object to do wash methods for Plates object and Cup object
- Good Exercise for students: write KitchenObject class.

```
KitchenObject subclass: #Baker
       instanceVariableNames:'name weight height'
       classVariableNames:''
       PoolDictionaries:''
       category:''

       name
              ^name

       name: aNewName
              name := aNewName

       bakeCake: ingredients
              | cake |
              cake := Cake new from: ingredients.
              ^cake

       wash: dirtyDishes
              dirtyDishes := dirtyDishes soak.
              dirtyDishes := dirtyDishes scrub.
              dirtyDishes := dirtyDishes dry.
              ^dirtyDishes

KitchenObject subclass: #Dishwasher
       instanceVariableNames:'name weight height'
       classVariableNames:''
       PoolDictionaries:''
       category:''

       name
              ^name

       name: aNewName
              name := aNewName

       wash: dirtyDishes
              ^ self runCycleOn: dirtyDishes

       runCycleOn: someDishes
              someDishes := someDishes rinse.
              someDishes := dry.
```

```
class CKitchenObject : public CBaker {
public:
       char* name;
       int weight;
       int weight;

public:
       CCake bakeCake(CIngredients ingredients);
       CDishes wash(CDishes dirtyDishes);
}

CCake CBaker::bakeCake(CIngredients ingredients)
{
       CCake cake = new CCake(ingredients);
       return cake;
}

CDishes CBaker::wash(CDishes dirtyDishes)
{
       dirtyDishes.soak();
       dirtyDishes.scrub();
       dirtyDishes.dry();
       return dirtyDishes;
}

class CKitchenObject : public CDishwasher {
public:
       char* name;
       int weight;
       int height;

public:
       CDishes wash(CDishes dirtyDishes);
}

CDishes CDishwasher::wash(CDishes dirtyDishes)
{
       this.runCycleOn(dirtyDishes);
}

CDishes CDishwasher::runCycleOn(CDishes dirtyDishes)
{
       dirtyDishes.rinse();
       dirtyDishes.dry();
}
```

```
typedef struct {
       char* name;
       int height;
       int weight;
} Worker;

Worker baker;
Worker dishwasher;

char* getName (Worker aWorker)
{
       return aWorker.name;
}

void nameWorker( Worker aWorker, char* newName)
{
       strcpy(aWorker.name, newName);
}

void bakeCake(Ingredient_struct* ingredients,
              Cake* newCake)
{
       newCake = doSomethingWith(ingredients);
}

void bakerWashDishes(Dishes* dirtyDishes)
{
       soak(dirtyDishes);
       scrub(dirtyDishes);
       dry(dirtyDishes);
}
```

```
void dishwasherWashDishes(Dishes* dirtyDishes)
{
       dirtyDishes = runCycleOn(dirtyDishes);
}

void runCycleOn(Dishes* dirtyDishes)
{
       dirtyDishes = rinse(dirtyDishes);
       dirtyDishes = dry(dirtyDishes);
}
```

Lecture 2: Classes and Instances

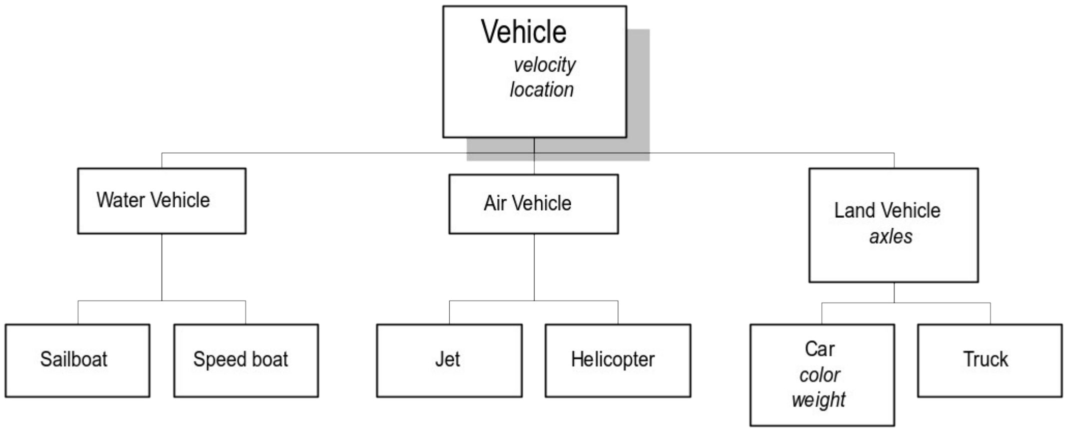
- **Class**
  - A template for objects that share common characteristics.
  - Includes an object's state variable and methods
    - Ex: Vehicle class

```
Vehicle
   Velocity
   Location
   Color
   Weight
Start( )
Stop( )
Accelerate( )
```

- **Instance**
  - A particular occurrence of an object is defined by a class
    - Classes are sometimes thought of as factories. If we had an automobile factory, the class would be the factory and the automobiles would be the instances of that factory.
  - Each instance has its own values for instance variables
    - Each automobile has its own engine, hood, doors, etc.
  - All instances of a class share the same method
    - Methods are the functions that are applicable to all instances of a class.
      - The method `accelerate` is applicable to all automobiles
    - Ex: A road contains many instances of vehicles, all different colors, going different speeds, starting, stopping, accelerating, etc
    - Ex: cars on a road. It is important to note that car1 and car 3 are not the same object, but are the both instances of the class Car. car1 and car3 are equivalent, but not equal. Equality implies they are the same object.

```
aRoad = Road new.
car1 = Car new withColor: red withSpeed 30.
car2 = Car new withColor: blue withSpeed 45.
car3 = Car new withColor: red withSpeed 30.
```

- **Class Hierarchy**
  - Allows sharing of state and behavior
    - Subclasses are able to use the methods and variables of the parent classes.
  - Each class refines / specializes its ancestors
  - Child can add new state information
    - A Land Vehicle adds the state information regarding the number of axles
  - Child can add, extend or override parent behavior
    - All Vehicles can be driven, but all types of vehicles require different sets of methods to drive
  - Superclass is the parent and subclass is a child
  - Abstract class holds common behavior & characteristics, concrete classes contain complete characterization of actual objects
    - In the Vehicle example, Vehicle is the Superclass, and Sailboat, Speed boar, Jet, Helicopter, Car and Truck are the concrete classes.



Ex: Vehicle hierarchy (leaves are concrete, all other are abstract)

Lecture 3: Messages, Methods, and Programming in Smalltalk

- **Messages**
  - A message specifies what behavior an object is to perform
  - Only way to communicate with an object
  - Implementation is left up to the receiver object
    - Ex: Ask the baker to bake a cake. We don't care how he does it.
    - Ex: `baker bakeCake.`
  - State Information can only be accessed via messages
    - Ex: I want to know how old you are (one of your state variables), so I ask you. I don't care how you compute your age, all I care about is the answer.
    - Ex: `baker age.`
  - The receiver object always returns a result (object).
    - A lot of the time a receiver is modified and it doesn't make sense to return something, so the argument is returned
    - Ex: `#(a b c) at: 3 put: #d` returns #d
- **Methods**
  - A method specifies how a receiver object performs a behavior.
  - Executed in response to a message
  - Must have access to data (must be passed, or contained in object)
    - If there is no access passed or contained in the object, what can be done?
  - Needs detailed knowledge of data
  - Can manipulate data directly
    - Can modify instance variables of the object receiving the message
    - Ex: `#(a b c) at: 3 put #d`. modifies the collection which is the instance variable
  - Returns an object as a result of its execution
    - Since a method is executed in response to a message, and we have already said all messages return an object, it should only make sense that the method returns an object as the result of its execution
  - Has same name as the message name
    - Ex: `#(a b c) size`. `size` is the message called by the receiver, and the `size` method is the method in class `Array` to be executed
  - Visual Works does no type checking on arguments, although the types should be type-compatible.
  - Method returns the receiver object by default, unless explicitly returned
    - Ex: Bob is asked to bake a cake. Bob's 'bake' method explicitly says to return a cake, rather than returning himself to the requester.
    - Ex: the `at:` method of class Interval
      - Explicitly returns a temp variable

```
at: anInteger
        "Answer the number at index position
         anInteger in the receiver interval."
    | answer |
    anInteger > 0
        ifTrue: [
            answer := beginning + (increment *(anInteger
                        - 1)).
            (increment < 0
                and: [answer between: end and:
                        beginning])
                    ifTrue: [^answer].
            (increment > 0
                and: [answer between: beginning and: end])
                    ifTrue: [^answer]].
    ^self errorInBounds: anInteger
```

    - Ex: the `asString` method of class String

        - Returns the receiver (`self`)

```
asString
   "Answer the string representing the
     receiver (the receiver itself)."
   ^self
```

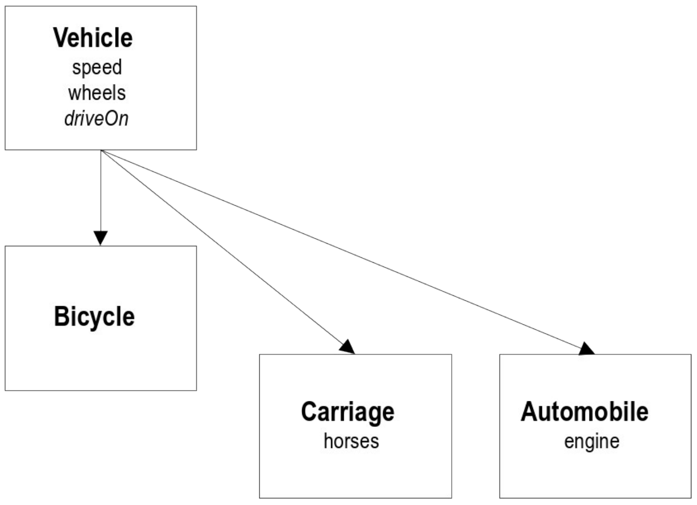
- **Programming in Smalltalk**
  - Code is written and tested in small pieces
    - Usually each method is tested after completion
  - Smalltalk is interpreted
    - Code is compiled into bytecode incrementally during development
      - Once the code has been written, it is “accepted” and compiled into bytecode, then tested.
    - Bytecode is interpreted by the Virtual Machine.
    - The advantage to a Virtual Machine is that different machines can have their own VM to interpret the bytecode. Thus, compiled code should be platform independent.
    - Rather than compile all classes for each program, Smalltalk compiles all of the classes and methods into an “image”

Lecture 4: OO Classification Techniques

- The Vehicle Class Description

```
Object subclass: #Vehicle
       instanceVariableNames: 'speed wheels'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := numberOfWheels.
              aNewVehicle speed := aSpeed.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self speed < aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1.
                            ^self]
                     ifFalse:
                       [self speed := (self speed) - 1.
                            ^self].
```

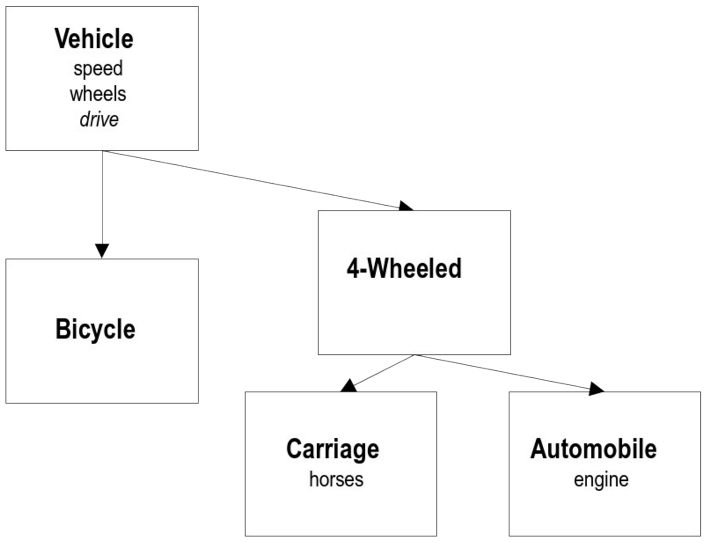
- **Specialization**
  - The act of creating a subclass of class. The new class inherits, overrides, and extends the behavior of the superclass.
  - How?
    - Add instance variables as needed
    - Add, extend, or override methods as needed
  - “is-a” relationship. An automobile “is-a” vehicle.
  - Benefit- code reuse
  - Ex: Class Vehicle exists before Class Automobile is invented. Class Automobile is invented, but based on the methods and variables of Class Vehicle.



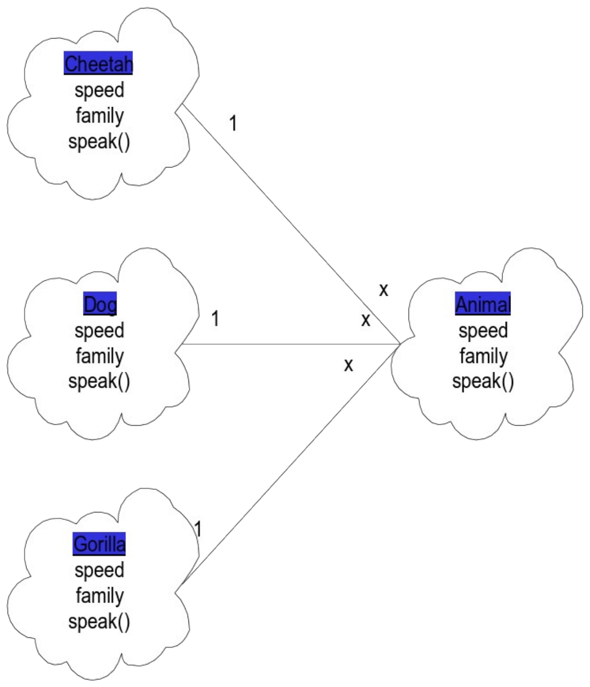
```
Vehicle subclass: #Automobile
       instanceVariableNames: 'speed wheels engine'
       classVariableNames: ''
       PoolDictionaries: ''
```

```
category: ''.

withWheels: numberOfWheels goingSpeed: aSpeed withEngine: anEngine
       "Create a new Automobile Object"
       | aNewAuto |
       aNewAuto := self new.
       aNewAuto := Vehicle withWheels: numberOfWheels
                     goingSpeed: aSpeed.
       aNewAuto engine := anEngine.
       ^aNewAuto.

driveOn: aRoad
       "Returns the receiver, does the driving"
       self speed < aRoad speedLimit
              ifTrue: [self engine accelerate. ^self]
              ifFalse: [self engine decelerate. ^self].
```

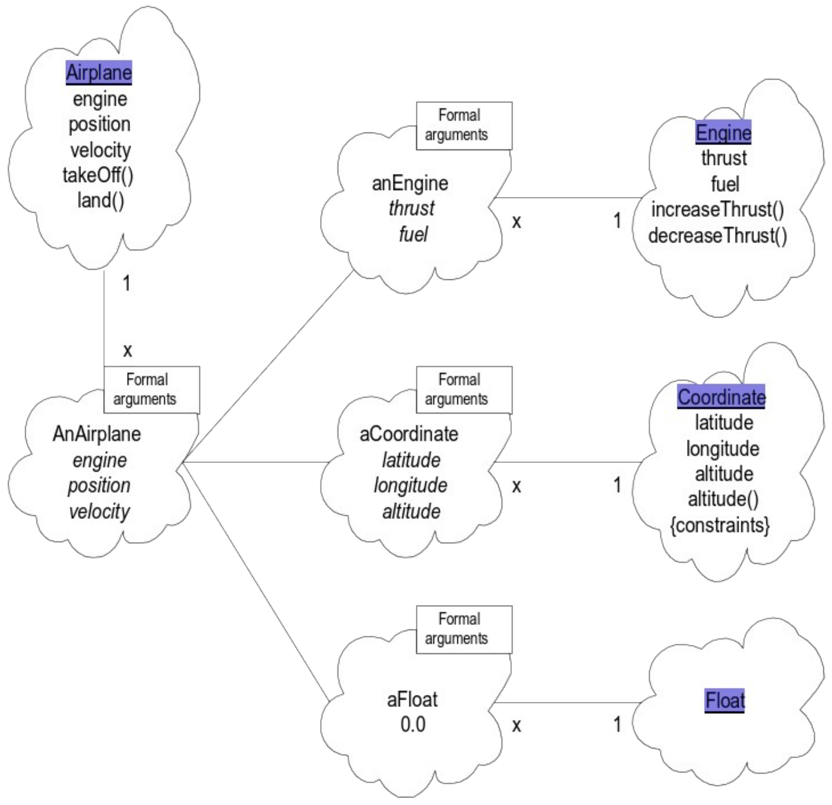
- **Abstraction**
  - The act of creating a superclass for several classes.
  - How?
    - Identify the shared state and /or behavior across the classes
    - Move shared properties to the new abstract superclass
    - Interpose the new abstract superclass in the class hierarchy
  - Benefit: code reuse, simplify maintenance, better understanding
  - Example



  - Example:



- **Composition**
  - The act of creating a class that is composed of instances of other classes (via instance variables). The new class does not inherit form the other classes, but can access their state and behavior via messages.
  - How?
    - Create a new class that is composed of other classes
      - Attributes of the new class are instances of other classes
      - The new class obtains the behavior of composition classes by sending messages to the referenced instances (“delegation”).
  - Benefit: provides protection from changes in referenced classes.
  - Behavior is not inherited
  - “has-a” relationship (also known as “is-part-of”).
  - Ex: An instance of class Airplane might be composed of instances of the class variables Engine, Position, and Velocity.
    - Engine points to an instance of class Engine (user defined)
    - Position points to an instance of class Position (user defined)
    - Velocity points to an instance of Visual Works system class, Float.
    - An Airplane “has-a” Engine, Position, and Velocity.



- **Factorization**
  - The act of breaking a class into smaller classes.
  - How?
    - Factor the class into smaller classes
      - Create a new class for each distinct type of state / behavior
      - Recombine the new classes via inheritance and /or composition to achieve original functionality.
  - Example: break the class Animal into different Species Classes
  - Benefit: Potential reusable, smaller classes

```
Object subclass: #Vehicle
       instanceVariableNames: 'speed wheels'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := numberOfWheels.
              aNewVehicle speed := aSpeed.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self speed > aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1. ^self]
                     ifFalse:
                       [self speed := (self speed) - 1. ^self].
```

```
#Vehicle subclass: #TwoWheel
       instanceVariableNames: 'speed wheels balance'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := 2.
              aNewVehicle speed := aSpeed.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self balnce = nil
                     ifTrue: [self speed := 0. ^self].
              self speed < aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1. ^self]
                     ifFalse:
                       [self speed := (self speed) - 1. ^self].

#Vehicle subclass: #FourWheel
       instanceVariableNames: 'speed wheels fourWheelDrive'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
                         isFourWheelDrive: anAnswer
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := 4.
              aNewVehicle speed := aSpeed.
              aNewVehicle fourWheedDrive := anAnswer.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self speed < aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1. ^self]
                     ifFalse:
                       [self speed := (self speed) - 1. ^self].
```

Lecture 5: Encapsulation & Polymorphism

- **Encapsulation**
  - Objects encapsulates *State* as a collection of variables
    - Common practice is to provide a set of private methods for manipulating variables.
    - Example: Baker has work state (ie rolling dough, baking, resting)
      - `baker state.` Returns the baker's state
      - `baker state: 'baking'.` Sets the baker's state
    - Example: The class Engine
      - In the previous lecture we looked the the Automobile class. When we created an instance of the class Automobile, we assumed the instance creation was called with an instance of Engine as an argument
      - An engine must have many private methods. When you turn the ignition, you don't have to start each component of the engine individually. Lets look at a simple engine class

```
Object subclass: #Engine
       instanceVariableNames: 'state pistons battery'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       start
              "Starts up the engine"
              self startEachComponent.
              ^status.

private
       startEachComponent
              "Checks to see if the battery is charged, and
                     tries to start the pistons"
              status := true.
              pistons := Pistons new.
              battery := Battery new.
              battery status
                     ifFalse: [status := false].
              pistons start
                     ifFalse: [status := false].
```

  - Objects encapsulates *Behavior* as methods invoked by messages
    - Set of methods encompasses everything the object knows how to do
    - Ex: Baker has `setState` method to set `stateVariable`, and `queryState` to get `stateVariable`'s value:

```
setState: aValue
       stateVariable=aValue.

queryState
       ^StateVariable.

Baker Bob do: 'resting'.
Bob queryState.
```

  - Encapsulation protects the state information of an object
    - Legal Example: Baker object can access thoughts (read and write)
    - Illegal Example: Someone else cannot read the baker's thoughts.
  - Encapsulation hides implementation details
    - Don't care how baker bakes cake.
  - Encapsulation provides a uniform interface for communicating with an object.

- We can ask the baker to bake a cake, or we can ask the chef to bake a cake. They will do it differently, but we can ask them the same way.
- Facilitates modularity, code reuse and maintenance.
  - Side note: C++ faq claims encapsulation does not facilitate code-reuse, this is an important difference in the language C++ programmers should consider.
- **Polymorphism**
  - Variety of objects in an application that exhibit the same generic behavior, but implement it differently
    - Ex: Ask a dog to speak, it barks. Ask a cat to speak, it meows. Each animal can be asked to speak, and each will do it differently.
    - Ex: The + operator for class Float and class Integer
      - Float:

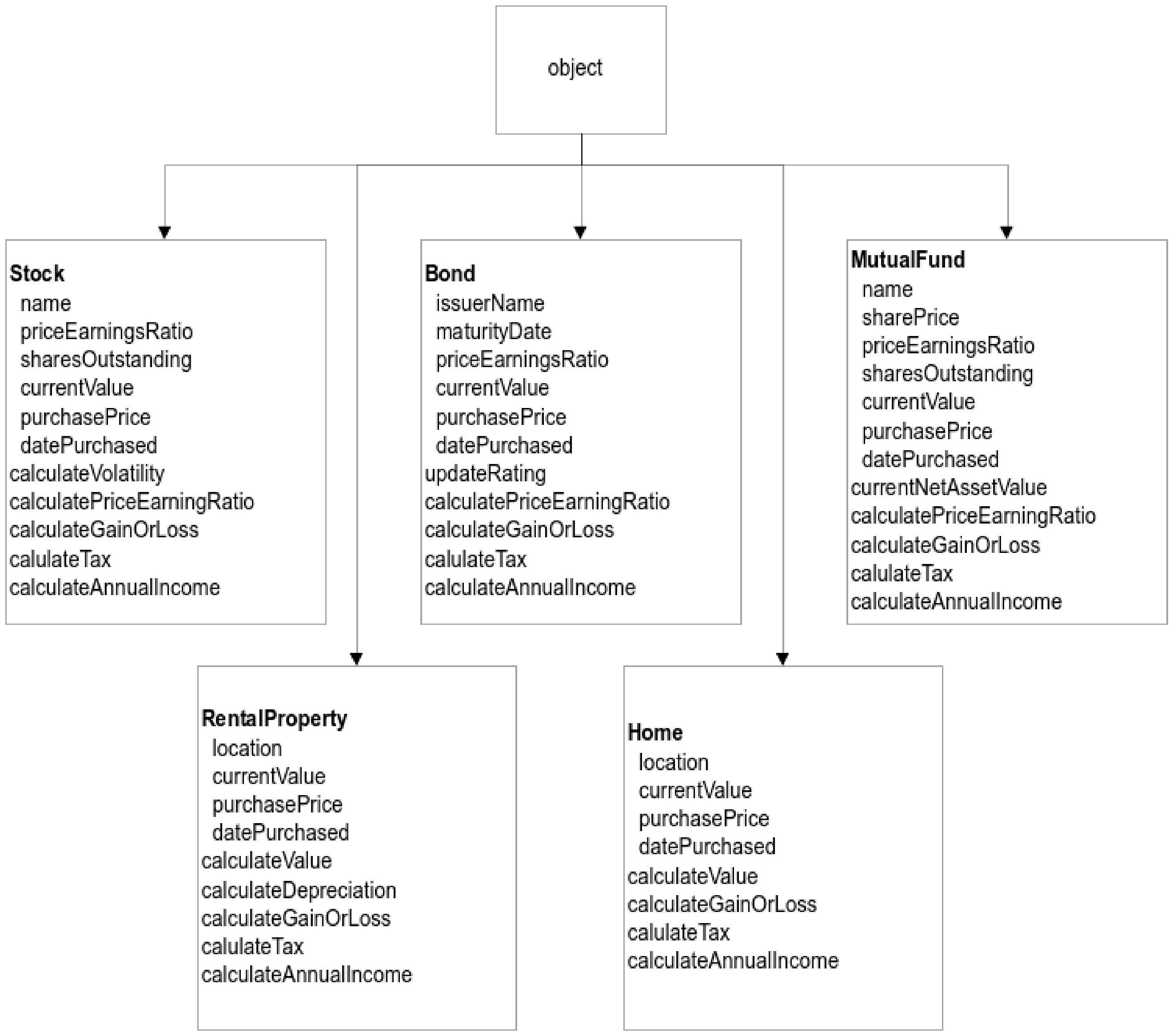
```
+ aNumber
        "Answer sum of the receiver and aNumber."
    | result |
    <primitive: 41>
    aNumber isFloat
        ifTrue: [
              result := self class basicNew: 8.
              FloatLibrary add: self to: aNumber result:
                     result.
             ^result]
        ifFalse: [^self + aNumber asFloat]
```

      - Integer:

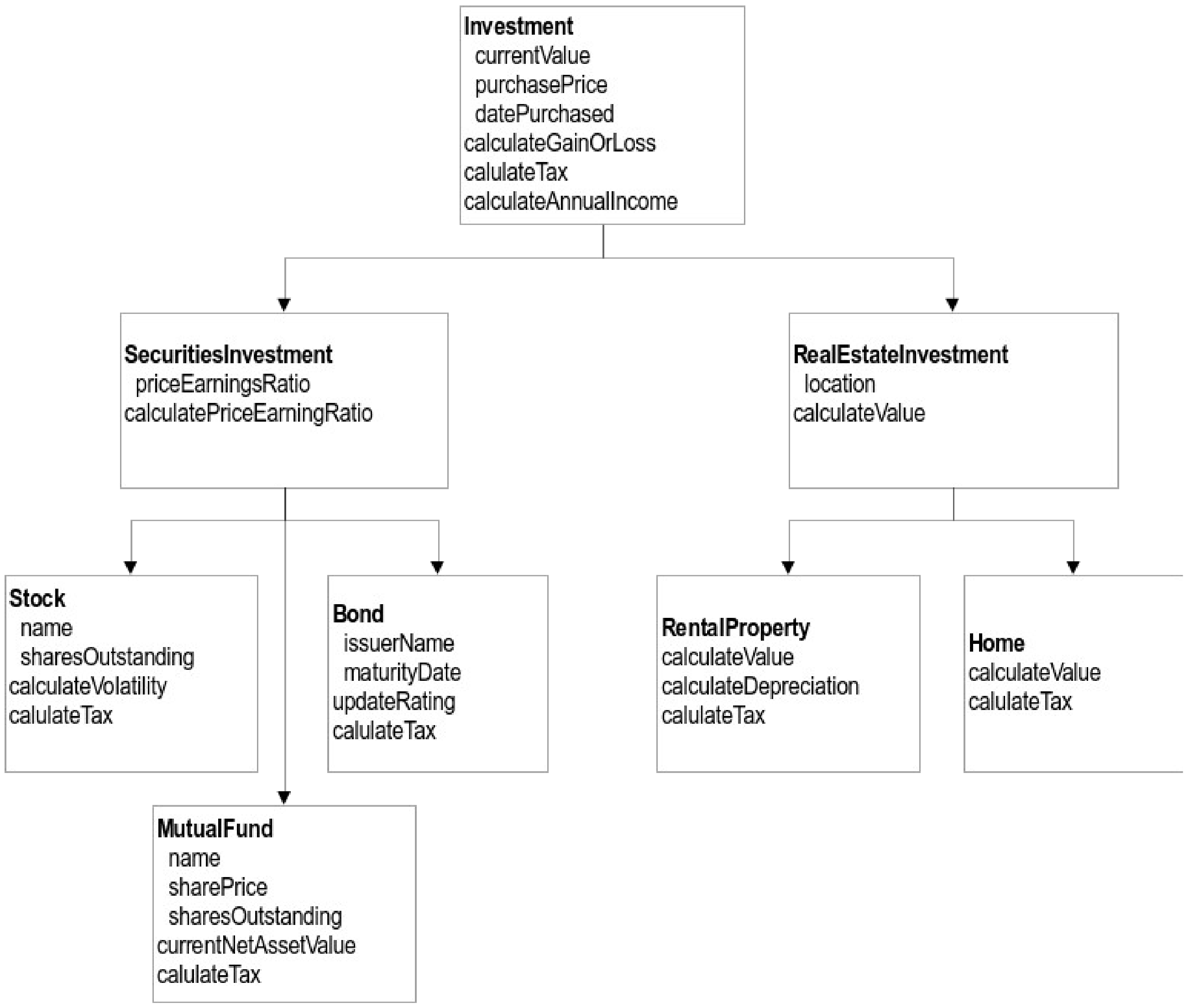
```
+ aNumber
        "Answer the sum of the receiver and aNumber."
    <primitive: 21>
    ^aNumber + self
```

Lecture 6: OO 4-Pass Process – an Investment Manager

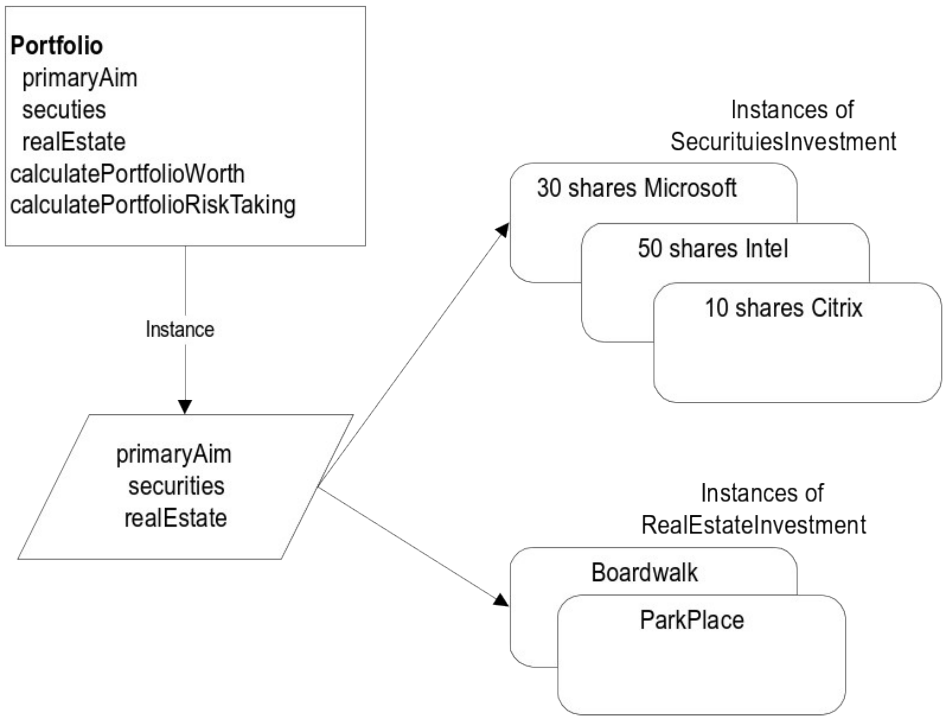
- **Pass 1: Abstraction**
  - Abstrction to share state/ behavior common to all investemnts
- **Pass 2: Abstraction**
  - Abstraction to share state / behavior for securities objects vr. Real estate investment objects
- **Pass 3: Composition**
  - Composition to create a portfolio of investments with a primary investment plan
- **Pass 4: Factorization**
  - Factorization to make explicit an anaysis of economic conditions related to investments
- Problem Statement: Design an Investment manager to handle stocks, bonds, mutual funds, houses and rental property
- Initial Design
  - What functionality do all investments share?
    - They all have currentValue, purchasePrice and datePurchased instance variables and calculateGainOrLoss, calculateTax and calculateAnnualIncome methods.
    - These variables and methods can be considered as the basis of creating a new, abstract superclass for the investments.



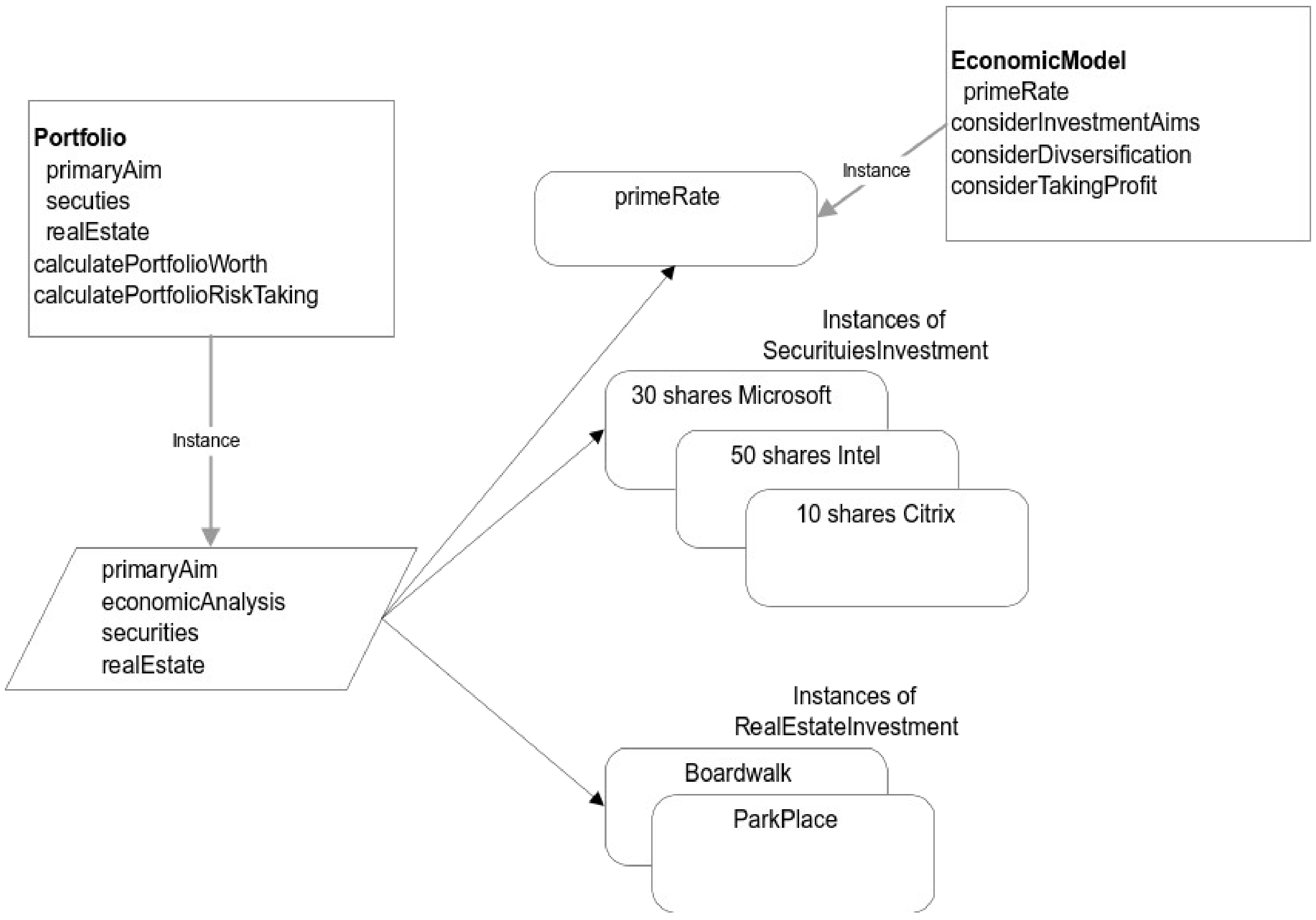
- Design Pass 1 (abstraction)
  - We can use abstraction to produce a new class, Investment. This is an abstact class that serves as the superclass for the concrete investment classes. It holds state variables and methods common to all investments



- Design Pass 2 (abstraction)
  - We now produce two new abstract classes:
    - SecuritiesInvestment to hold commonalties between Stock, Bond, and MutualFund.
    - RealEstateInvestemnt to hold commonalties between Home and RentalProperty.



- Design Pass 3 (composition)
  - Now we create a Portfolio class to hold all of the primary investment aim (risk level) and the collection of investments.
  - We'll create two state variables which hold the two collections of objects made up from the two classed defined in Pass 2.



- Design Pass 4 (factorizarion)
  - In the final pass, we factor out “economic model” state and behavior as apotentially reusable part of Portfolio, and create the new class EconomicModel. This class lives outside the hierachy, and becomes part of the Portfolio via composition.
  - Remember factorization has two components
    - Break up large, complex classes into separate, more reusable components
    - Recover the original functionality through composition or inheritance.
  - How did we know to use composition instead of inheritance?
    - Which makes more sense:
      - “is-a” EconomicModel a Portfolio? (Inheritance)
      - Is an EconomicModel “part-of” a portfolio? (Composition)