CS 2100: Data Structures & Algorithms 1

Inheritance

A Re-emphasis and Proper Introduction

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Basic Inheritance

Inheritance is an **object-oriented** concept that supports cohesion, code reuse and polymorphic behavior

Motivation

- Sometimes we want to create objects that **naturally share a lot of functionality**.
  - e.g., AVL trees and BST both store and use **binary** nodes
  - e.g., **find()** in AVL and BST works the same way

- **Goal 1:** *Reduce* the amount of code that needs to be *duplicated*
- **Goal 2:** Allow for ***polymorphism*** between types that have shared attributes

Concrete Motivation

- Suppose we are writing some code for a **car website** (e.g., carmax)
- We might have some objects (and attributes) like:
  - **CAR**: make, model, price, year
  - **MOTORCYCLE**: make, model, ...,
  - **TRUCK**: make, model, price, towing capacity

- Suppose we are writing some code for a **business**
- We might have some objects (and attributes) like:
  - **EMPLOYEE**: name, homeAddress, workAddress, employeeId, …
  - **MANAGER**: name, homeAddress, workAddress, employeeId, office, …

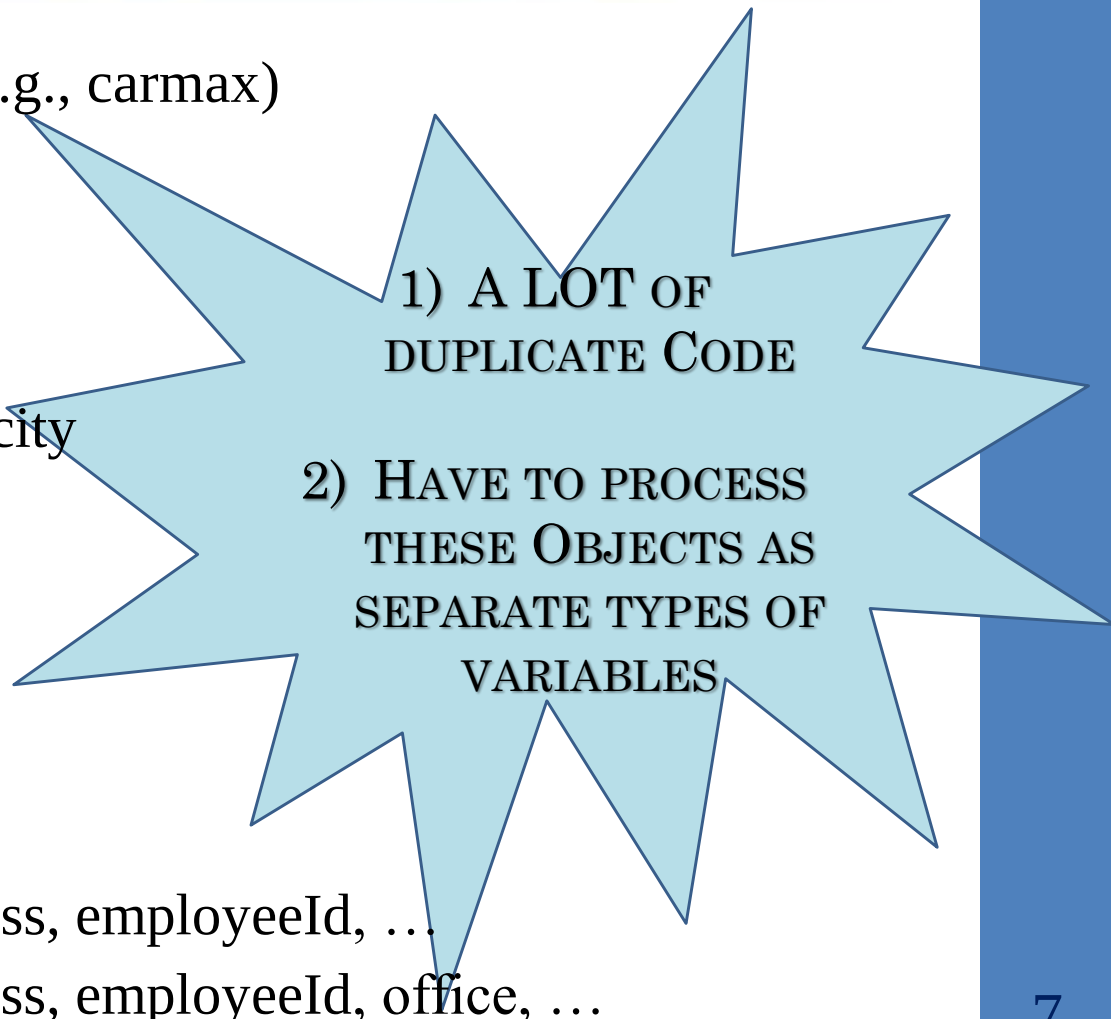
Concrete Motivation

- Suppose we are writing some code for a **car website** (e.g., carmax)
- We might have some objects (and attributes) like:
  - **CAR**: make, model, price, year
  - **MOTORCYCLE**: make, model, ...,
  - **TRUCK**: make, model, price, towing capacity

- Suppose we are writing some code for a **business**
- We might have some objects (and attributes) like:
  - **EMPLOYEE**: name, homeAddress, workAddress, employeeId, …
  - **MANAGER**: name, homeAddress, workAddress, employeeId, office, …

Concrete Motivation

- Suppose we are writing some code for a **car website** (e.g., carmax)
- We might have some objects (and attributes) like:
  - **CAR**: make, model, price, year
  - **MOTORCYCLE**: make, model, ...,
  - **TRUCK**: make, model, price, towing capacity



- Suppose we are writing some code for a **business**
- We might have some objects (and attributes) like:
  - **EMPLOYEE**: name, homeAddress, workAddress, employeeId, …
  - **MANAGER**: name, homeAddress, workAddress, employeeId, office, …

Inheritance

- Java provides **inheritance** as a mechanism for *organizing your classes* more succinctly.
- **Inheritance**: Is a property of a class in which it has a **parent** class. The **child** class **inherits** the fields and the methods of the parent class.

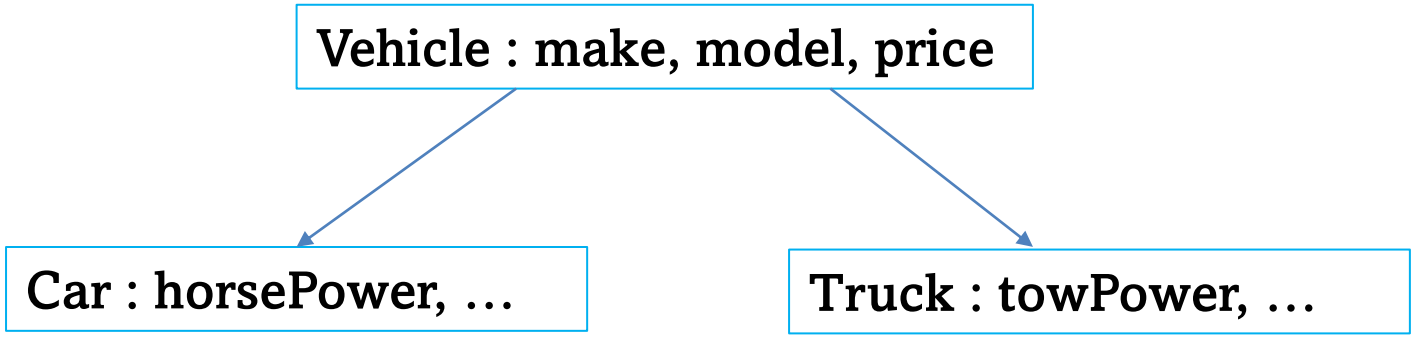
Inheritance Idea

- In the figure below for a car dealership: Many fields are duplicated in the **two classes**

Car : make, model, price, horsePower, …	Truck : make, model, price, towPower, …

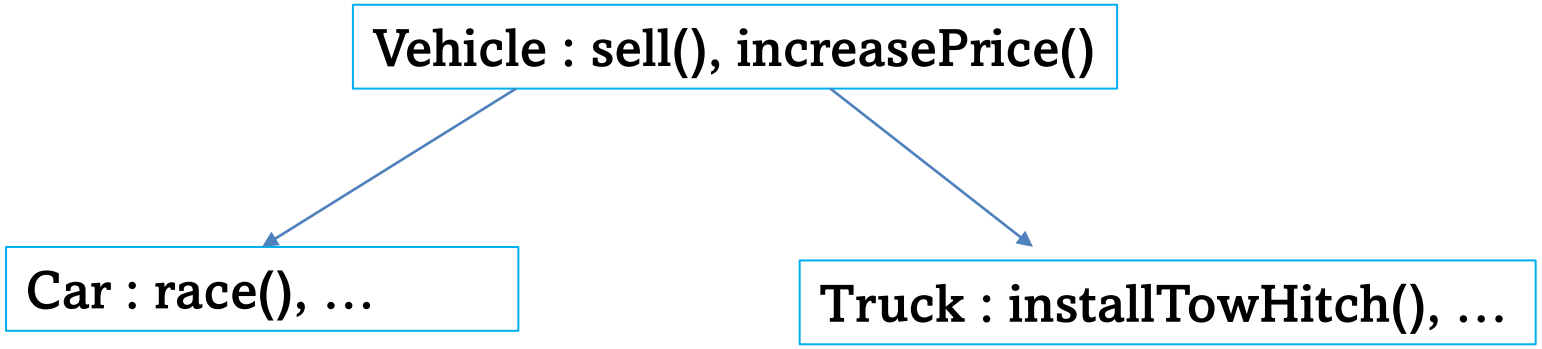
Inheritance Idea

- Using inheritance, all **vehicles** has some ***shared properties***, and cars/trucks have ***some unique ones*** too

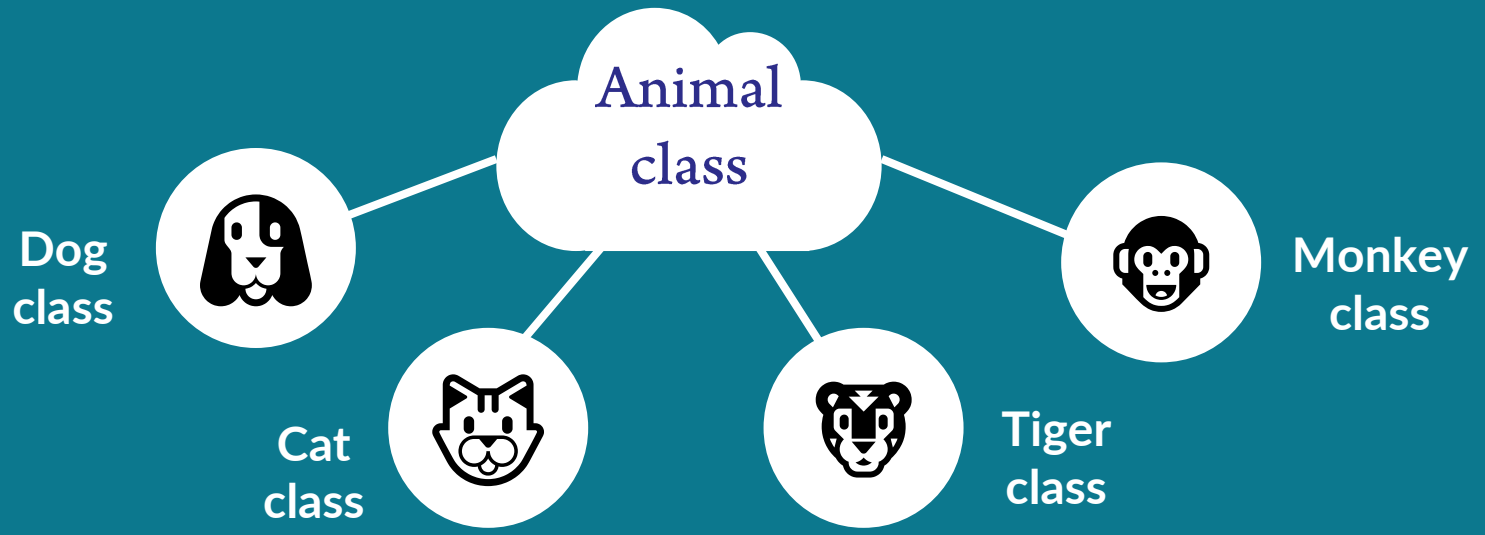


Inheritance Idea

- ***Behavior*** can be duplicated as well



Inheritance: is-a relationship



- A *subclass* extends a *superclass* (*abstracting common states and behaviors*)
- Use the *is-a test* to verify that your inheritance hierarchy is valid;
  if X extends Y then X is-a Y must make sense
- The *is-a relationship* works only in one direction; a lion *is-a* animal but not all animals are lions

Inheritance Vocabulary

- When a new class is defined from an existing class
  - The new class is called the **subclass** (**derived class** or *child* class)
  - The existing class is called the **superclass** (**base class** or *parent* class)

- We would say the following:
  - The subclass inherits from the superclass *(methods and attributes)*
  - The subclass **extends** the superclass.

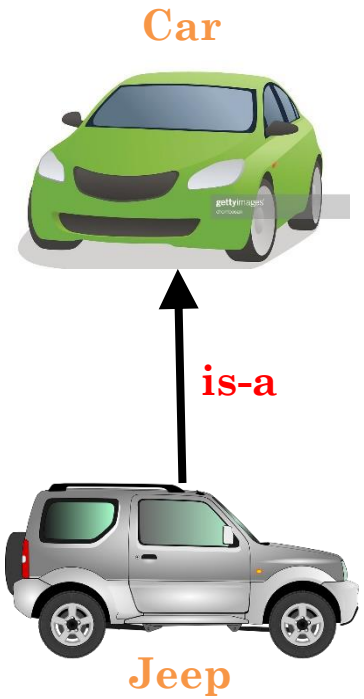
- A note on access modifier: **protected**
  - A subclass **cannot** access **private** fields or methods of the superclass
  - **Superclass** can allow subclass access by declaring fields/methods as protected (*visibility*: class itself, all subclasses, within same package)

Substitutability Principle

- We say: any subclass object (e.g., Jeep) **is-a** instance of a superclass object (e.g., Car), and **inherits** its states and behaviors

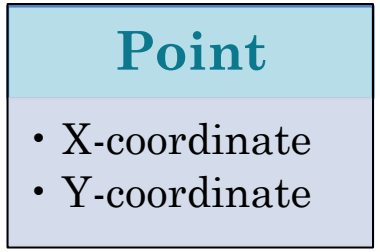
- Wherever we see a reference to a Car (**superclass**) object in our code, we can legally replace that with a reference to Jeep (any **subclass** object)

- Implies that we can **substitute** the subclass object in any way that's legal for the superclass

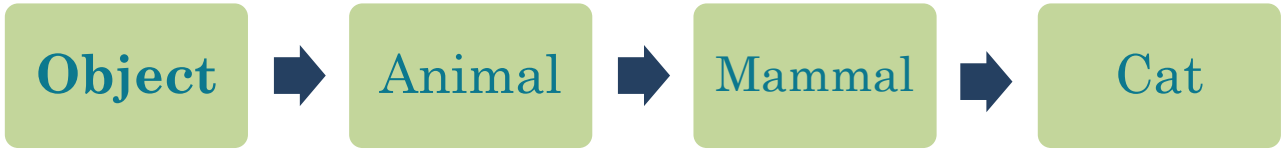


Composition vs Inheritance

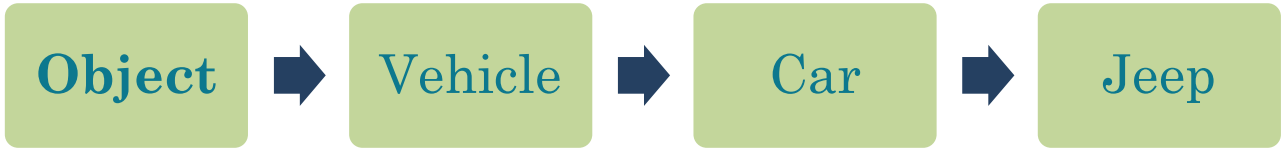
- Composition: **has-a** relationship
  - Point class, has a x- and y-coordinate
  - Living room, has a sofa, recliner, coffee table, tv



- Inheritance: **is-a** relationship
  - `public class Mammal extends Animal { }`



  - `public class Jeep extends Car { }`



Don't Repeat Yourself...!

- Many times we need a class that is only slightly different from an existing class
  - Don't repeat yourself (DRY)! ~ Write once!
  - Sometimes we just need to add something to the state or add/change the behavior of a method

    →Use inheritance!

- Note:
  - Every subclass `extends` its superclass
  - Exception: We inherit `Object` without typing `extends Object`

Motivations for Inheritance

- **Benefits**: Inheritance can help with the following:

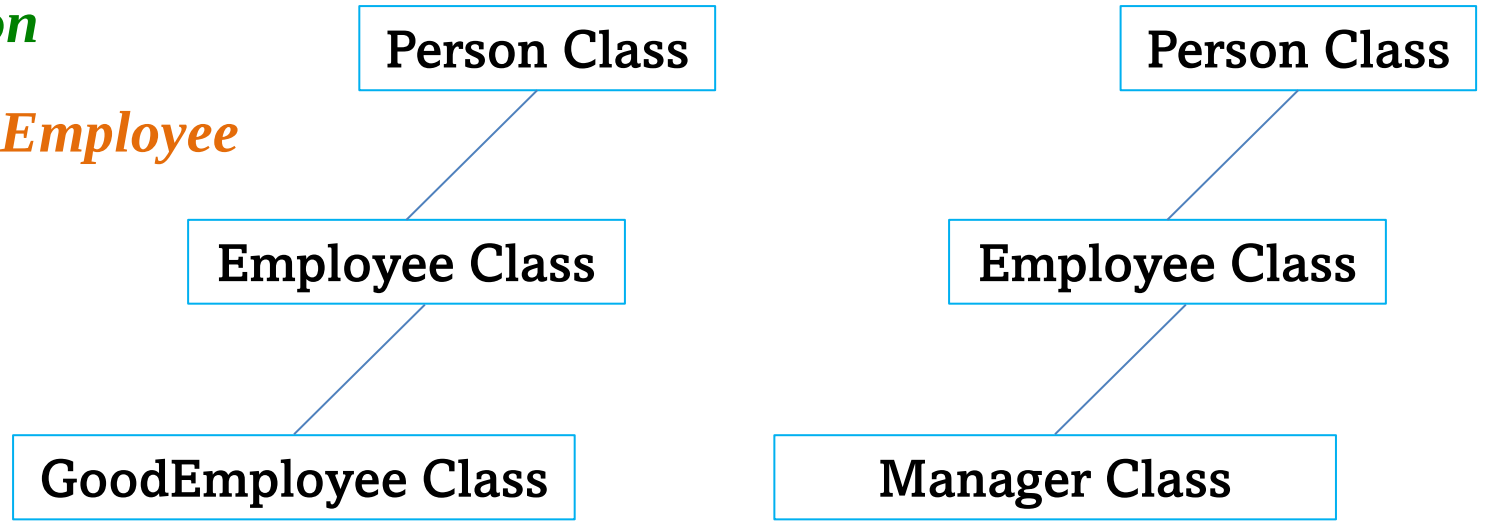
1. **Code reuse**
   - Our new (subclass/child) class "**extends**" the existing (subclass/parent) class and allows us to **re-use code** that they have in common
2. **SW that better matches the real world problem**
3. **Flexible Design**
   - Gives us **flexibility** at run-time in calling operations on objects that might have different types (→run-time **polymorphism**)

Another Inheritance Example

- Using inheritance, all ***Employees*** are a ***Person***, and all ***Good Employees*** are ***Employees***

- ***Employee*** **extends** ***Person***
- ***GoodEmployee*** **extends** ***Employee***

- Java allows you to use inheritance with the **EXTENDS** keyword



```java
public class Person {

    // Notice, use of *protected*
    protected String name;
    protected String homeAddress;

    /* Constructor */
    public Person(String n, String ha) {
        this.name = n;
        this.homeAddress = ha;
    }
```

"**extends**" means the class *automatically* gets all public fields and methods of its parent

```java
public class Employee extends Person{

    //fields
    protected String workAddress;
    protected int employeeId;

    /* Constructor */
    public Employee(String n, String ha, String wa, int id){
        super(n,ha); // calling Person's constructor method
        this.workAddress = wa;
        this.employeeId = id;
    }
```

"**super**" is used to access fields and methods in the parent.
super() will also call the constructor of the parent class

Inheritance: super

How to access a superclass's states and methods

- The subclass object inherits state and behavior from the superclass object, but can **override** these properties
- A subclass object may choose to access the superclass's implementation of its overridden method by using the keyword **super**

```
class Animal {                          // Animal: superclass
     public String getName() {
          return this.name;
}    }
class Cat extends Animal {     // Cat: subclass
     public String getName() {
          return "Meow " + super.getName();
}    }
```

calling

Inheritance: super()

*How to call the superclass's **constructor** method(s)*

- Unless specified otherwise, the subclass constructor calls the superclass constructor **with no arguments** e.g.
  `super();`

- To call a superclass constructor, use `super()` reserved word as a method. Has to be the first statement of the **subclass constructor** *(can also pass arguments)*

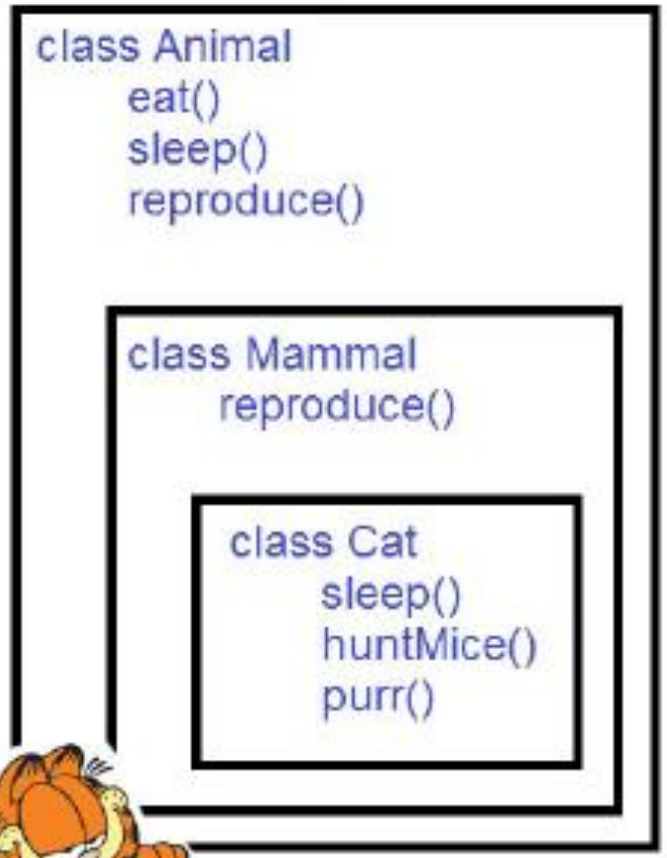
```
class Animal {     // superclass
   String name;
   public Animal(String name) {
      this.name = name;
   }
}
```

```
class Cat extends Animal { // subclass
   int hoursOfSleep;
   public Cat(String name, int hrs) {
      super(name);
      this.hoursOfSleep = hrs;
   }
}
```

```
Implicit super constructor Animal() is undefined. Must explicitly call another constructor
```

Inheritance

Inheritance: *Animal Example*



Cat garfield = new Cat
(20hrs, NoWay, Always);

Cat garfield has:
eat() -- Animal
reproduce() -- Mammal
sleep() -- Cat
huntMice() -- Cat
purr() -- Cat

Cat objects **inherit** all characteristics of Mammal objects and, in turn, Animal objects.

- sleep() in Cat *overrides* sleep() in Animal (to include long hours and naps)
- reproduce() in Mammal *overrides* reproduce() in Animal (mammals give live birth)

- We can **define** one class in terms of another
- Subclass gets (**inherits**) the state (*fields*) and behavior (*methods*) of the superclass
- We can **add** additional information (fields or methods) to the subclass
- We have the ability to **override** methods in the subclass *to better suit the required functionality* of that class

Inheritance and Run-time Polymorphism Example:

```
public class Animal {
    public void move() {
        S.O.P("Animals can move!");
    }
}

public class Cat extends Animal {
    public void move() {
        S.O.P("Cats can walk & run");
    }
}
```

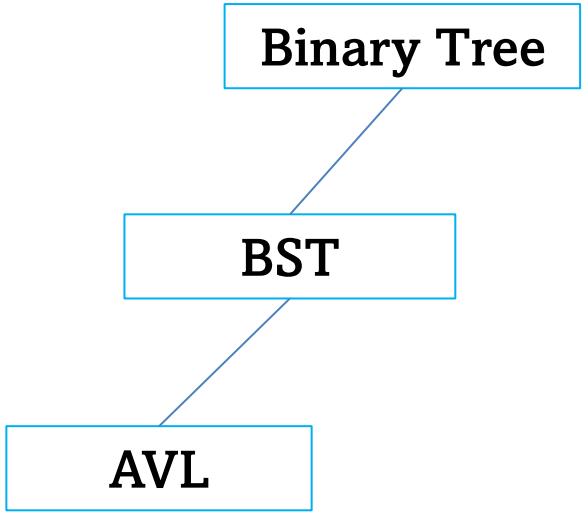
```
public class TestCat {
    public static void main(String
                        args[]) {
        //Animal reference & object
        Animal a = new Animal();
        //Animal reference, Cat obj.
        Animal b = new Cat();

        a.move();// method in Animal
        b.move();// method in Cat
```

```
OUTPUT: Animals can move!
        Cats can walk & run
```

Practical Example: Trees

- There are some things that **ALL trees have/do**:
  - Store tree nodes
  - All tree nodes have left and right child
  - All nodes have height
  - You can insert into any tree (though different mechanism)
- Inheritance is perfect for this



Practical Example: Trees

Binary Tree

- Stores BinaryTreeNode root
- pre-order, post-order, in-order traversal methods

BST

- Does everything Binary Tree does
- Inserts in sorted order, removes nodes
- Find()

AVL

- Does everything a BST does.
- Adds tree rotation methods
- Inserts and removes same way then rotates

Practical Example: Trees

- **Binary Tree**: things ALL binary trees have/do

```
public class BinaryTree<T> {
    protected TreeNode< T > root = null;
    /* IMPLEMENT THESE METHODS FOR HW */
    public void printInOrder();
    public void printPreOrder();
    public void printPostOrder();
}

public class TreeNode<T> {
    protected T data = null;
    protected TreeNode<T> left = null;
    protected TreeNode< T > right = null;
    protected int height = 0;
}
```

Practical Example: BSTs

- **Binary Search Tree**: things only BSTs do
- What is this "**`T extends Comparable <T>`**" thing?!?

```
public class BinarySearchTree< T extends Comparable< T > >
                extends BinaryTree< T > implements Tree< T >{

  public void insert(T data){...}

  public boolean find(T data){...}

  public void remove(T data){...}

  public TreeNode< T > findMax(TreeNode< T > curNode){...}
}
```

Practical Example: AVL Trees

- **AVLTree**: things only AVLs do
- Notice that we have **insert()** method again??

```
public class AVLTree< T extends Comparable< T >>
                extends BinarySearchTree< T >{

  @Override
  public void insert(T data){...}

  @Override
  public void remove(T data){...}

  private TreeNode< T > balance(TreeNode< T > curNode){...}
  private TreeNode< T > rotateRight(TreeNode< T > curNode){...}
  private TreeNode< T > rotateLeft(TreeNode< T > curNode){...}

  private int balanceFactor(TreeNode< T > node){...}
}
```

Practical Example: AVL Trees

- Notice that **AVL** Tree and **BST** both had an **insert()** method with the same parameters.
- This is called **overriding** a method.
- The parent class implemented the method already, but the child class wants to **override** that implementation, and *reimplement it slightly differently*.
  - Sometimes child will use `super.methodHere()` to **call the parent version** and then add more functionality on top
  - Sometimes child class will totally rewrite the method.

Practical Example: AVL Trees

- How does java know which **insert()** method to *actually* execute?
- Jave uses **Dynamic Dispatch**, meaning the run-time type of the object is examined, and the method in that class is automatically invoked.

```
BinarySearchTree< Integer > myTree = new AVLTree< Integer >();
/* ... */
myTree.insert(5); //AVLTree.insert() is called.
```