Inheritance I

CMSC 202

Class Reuse

- We have seen how classes (and their code) can be reused with composition.
  - An object has another object as one (or more) of its instance variables.
- Composition models the “has a” relationship.
  - A Car has a String (vin, color, make, model)
  - A Car has an Engine
  - A Book has an array of Pages

Object Relationships

- An object can be a specialized version of another object.
  - A Car is a Vehicle
  - A Motorcycle is a Vehicle
  - A Boat is a Vehicle
  - An Aircraft is a Vehicle
  - This kind of relationship is know as the “is a” relationship.
- In Object Oriented Programming, this relationship is modeled with a technique known as ***Inheritance***.
- Inheritance creates new classes by “adding” code to a preexisting class, without actually modifying that class' definition.

Inheritance

- ***Inheritance*** is one of the most important techniques used in OOP
- Using ***inheritance***
  - A very general class is first defined,
    - Vehicle, Fruit, Shape
  - Then more specialized versions of the class are defined such as Car, Boat, Aircraft (more specific versions of a Vehicle)
    - Adding instance variables and/or
    - Adding methods.
      - Car's have wheels, Boats have props, Aircraft have wings...
  - The specialized classes are said to ***inherit*** the methods and instance variables of the general class.

Derived Classes

- There is often a natural hierarchy when designing certain classes.
- Example:
  - In a record-keeping program for the vehicles on a military base, there are automobiles and aircraft.
  - Automobiles can be divided into Cars and Motorcycles.
  - Aircraft can be divided into Planes and Helicopters.

Derived Classes

- All vehicles have certain characteristics in common:
  - Vin number, color, number of operators, speed, number of passengers
  - the methods for setting and changing the vin, color, speed, number of passengers, and number of operators
- Some vehicles have specialized characteristics:
  - Move
    - Aircraft move on the ground and can move in the air
    - Automobiles move on the ground
  - Calculating move functions for these two different groups would be different.

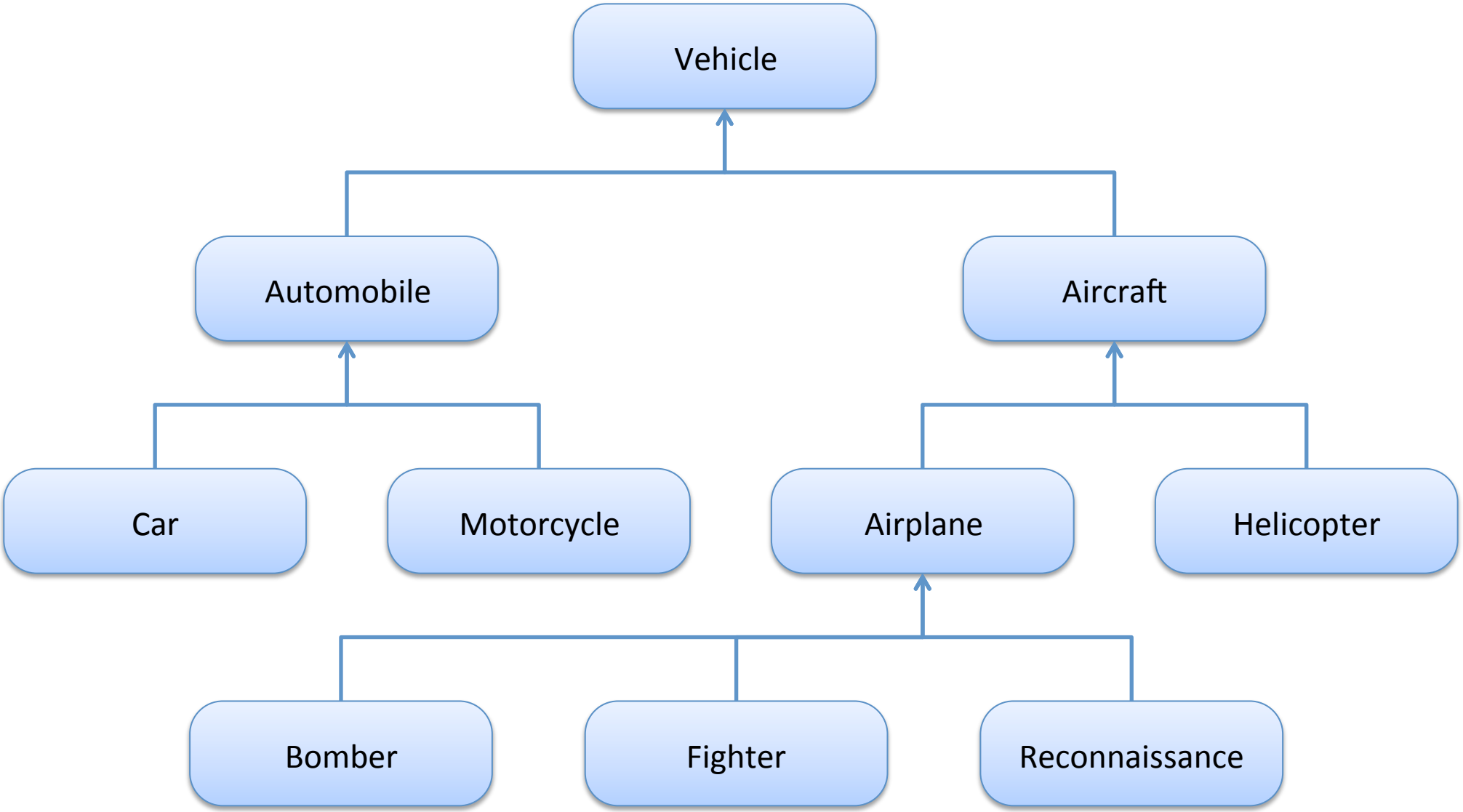
Inheritance and OOP

- Inheritance is an abstraction for
  - sharing similarities among classes (vin, color, speed,...), and
  - preserving their differences (how they move).
- Inheritance allows us to group classes into families of related types (Vehicles), allowing for the sharing of common operations and data.

General Classes

- A class called `Vehicle` can be defined that includes all Vehicles.
- This class can then be used to define classes for Automobiles and Aircraft
- The `Automobiles` class can be used to define a `Car` class, and so forth.

A Class Hierarchy



The Vehicle Class

```
public class Vehicle {
   private int vin;
   private Color color;
   private int numOperators;
   private int numPassengers;
   private int speed;

   private static int serialNumber = 111111;

   public Vehicle(){ /* code here */}
   public Vehicle(Vehicle v){ /* code here */ }
   public Vehicle(Color cc, int numOperators) {/*code here */}

   // some accessors and mutators
   public void changeColor(Color c){/* code here */}
   public void setNumPassangers(int p){/* code here */}
   public int getNumOperators(){ /* code here */   }
   public int getVinNumber() { /* code here */ }
   public String toString() { /* code here */ }
   public boolean equals(Vehicle other) { /* code here */}
   public void accelerate() { /* code here */ }
   public void decelerate() { /* code here */ }
   public int getSpeed() { /* code here */ }
}
```

Derived Classes

- Since an Automobile "***is a***" Vehicle, it is defined as a ***derived*** class of the class `Vehicle`.
  - A derived class is defined by adding instance variables and/or methods to an existing class.
  - The class that the derived class is built upon is called the ***base class***.
  - The phrase `extends BaseClass` must be added to the derived class definition:

```
public class Automobile extends Vehicle
```

Automobile Class

```
public class Automobile extends Vehicle {

   // instance variables local to the derived class
   private String make;
   private String model;
   private boolean locked;

   public Automobile() {/* code here */}
   public Automobile(String make, String model) {/* code here */}

   // methods that are local to the derived class
   public void isLocked() {/* code here */}
   public void lock() {/* code here */}
   public void unlock() {/* code here */}

   public String toString() {/* code here */}
   public boolean equals(Automobile other) {/* code here */}
}
```

Derived Class (Subclass)

- A derived class is also called a ***subclass***.
- The class derived from is called a ***base class*** or ***superclass***.
- The derived class inherits all of the following from the base class
  - public methods,
  - public and private instance variables, and
  - public and private static variables
- The derived class can add more instance variables, static variables, and/or methods.
  - The public interface of a derived class is typically larger than its base class because of the inherited members.

Inherited Members

- A derived class ***automatically*** has all of the following from the base class
  - instance variables,
  - static variables, and
  - public methods
- Definitions for the inherited variables and methods ***do not*** appear in the derived class.
  - The code is reused without having to explicitly copy it, unless the creator of the derived class redefines one or more of the base class methods.

Using Automobile & Inheritance

```
public static void main(String[] args){
    Automobile hummer = new Automobile("GMC","Hummer");

    // get the vin number of the hummer (method of Vehicle)
    System.out.println("Hummer vin: " + hummer.getVinNumber());

    // change the color of the hummer (method of Vehicle)
    hummer.changeColor(Color.DARK_GRAY);

    // lock the hummer (method of Automobile)
    hummer.lock();

    // lock the hummer (method of Vehicle)
    hummer.accelerate();

    // lock the hummer (method of ?)
    System.out.println(hummer);
}
```

Overriding a Method Definition

- A derived class can change or ***override*** an inherited method.
- In order to override an inherited method, a new method definition is placed in the derived class definition.
- For example, Automobiles decelerate and accelerate at a rate of 5 mph per function. It would make sense to override Vehicle's accelerate and decelerate methods by defining its own.

Overriding Example

```
public class Vehicle {
   // other class code ...
   public void accelerate() { ++speed; }
   public void decelerate() { --speed; }
}

public class Automobile extends Vehicle {
   // other class code ...
   public void accelerate() { speed += 5; }
   public void decelerate() { speed += 5; }
}
```

Now, this code

```
Automobile hummer = new Automobile( );
hummer.accelerate( );
```

invokes the ***overridden*** `accelerate()` method in the ***Automobile*** class rather than the `accelerate()` method in the ***Vehicle*** class

To override a method in the derived class, the overriding method must have the ***same method signature*** as the base class method.

Overriding Versus Overloading

- Do not confuse ***overriding*** a method in a derived class with ***overloading*** a method name.
  - When a method in a derived class has the same signature as the method in the base class, that is ***overriding***.
  - When a method in a derived class or the same class has a different signature from the method in the base class or the same class, that is ***overloading***.
  - Note that when the derived class ***overrides or overloads*** the original method, it still inherits the original method from the base class as well (we'll see this later).

The final Modifier

- If the modifier `final` is placed before the definition of a ***method***, then that method ***may not*** be overridden in a derived class.
- It the modifier `final` is placed before the definition of a ***class***, then that class ***may not*** be used as a base class to derive other classes.

Pitfall: Use of Private Instance Variables from a Base Class

- An instance variable that is `private` in a base class is not accessible by name in a method definition of a derived class.
  - An object of the `Automobile` class cannot access the private instance variable `speed` by name, even though it is inherited from the `Vehicle` base class.
- Instead, a private instance variable of the base class can only be accessed by the public* accessor and mutator methods defined in that class.
  - An object of the `Automobile` class can use the `getSpeed` or `accelerate`/`decelerate` methods to access `speed`.

* At least public is the only way we've discussed thus far

Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from a Base Class

- If private instance variables of a class were accessible in method definitions of a derived class, …
  - then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access the variables in a method of that class.
- This would allow private instance variables to be changed by mistake or in inappropriate ways.

Pitfall: Private Methods Are Effectively Not Inherited

- The private methods of the base class are like private variables in terms of not being directly available.
- A private method is completely unavailable, unless invoked indirectly.
  - This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private method.
- This should not be a problem because private methods should be used only as helper methods.
  - If a method is not just a helper method, then it should be public.