Interfaces and Abstract Classes

CS 240 – Advanced Programming Concepts

Polymorphism

- Poly = “many”
- Morph = “change”
- Polymorphism = “many forms” (we don’t say many changes)
- Objects can take on many forms in an object-oriented program
  - The form of the class in which they are declared
  - The form of any parent class in the class’ inheritance hierarchy
  - ‘Object’ is a parent class whether declared or not (always at the top of the inheritance hierarchy)
- The form is represented by the type of the reference that refers to the object
  - Reference and object types may differ (but must be compatible according to inheritance—’is a’)
- **Simple definition:** A reference of one type referring to an object of a different type

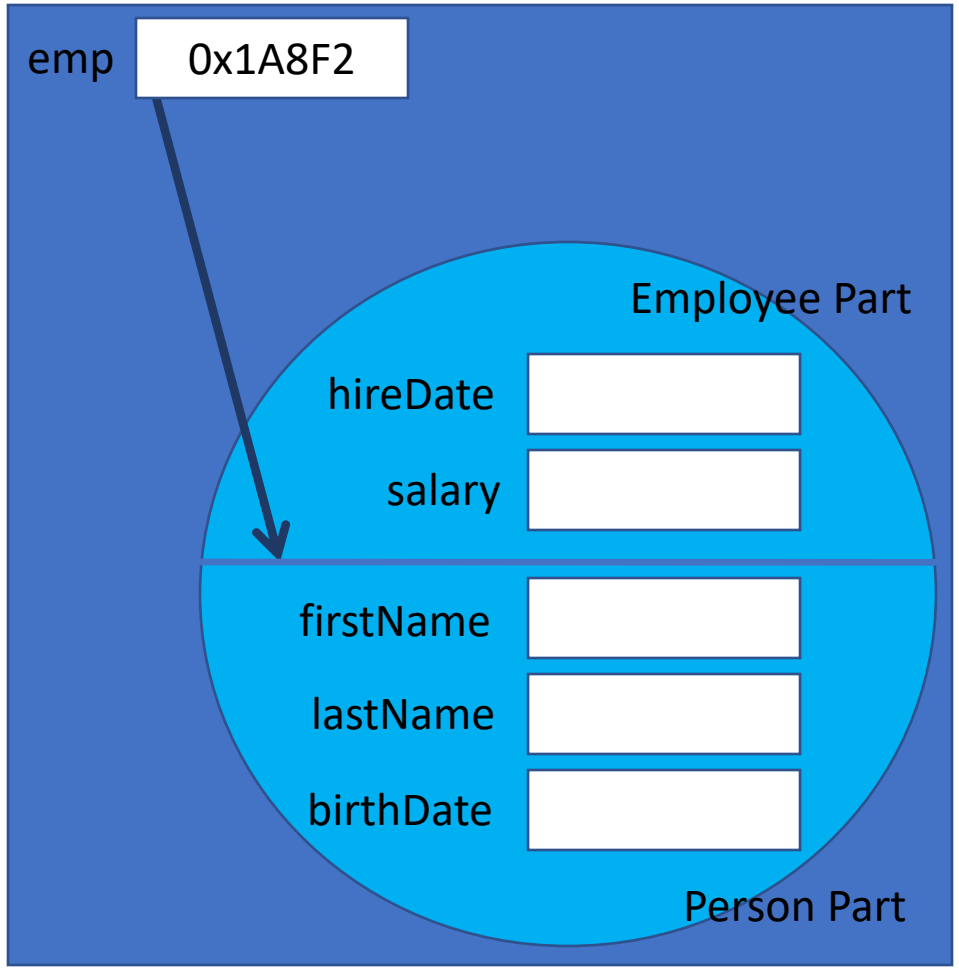
Simple Polymorphism Examples

- Employee emp = new Employee(); **OK**
- Person emp = new Employee(); **OK**
- Object emp = new Employee(); **OK**
- ~~Dog emp = new Employee();~~ **NOT OK**

Result in Memory

```
Person emp = new
  Employee();
```

Although they still exist in memory, 'hireDate' and 'salary' cannot be accessed from the 'emp' reference



Reasons for Polymorphism

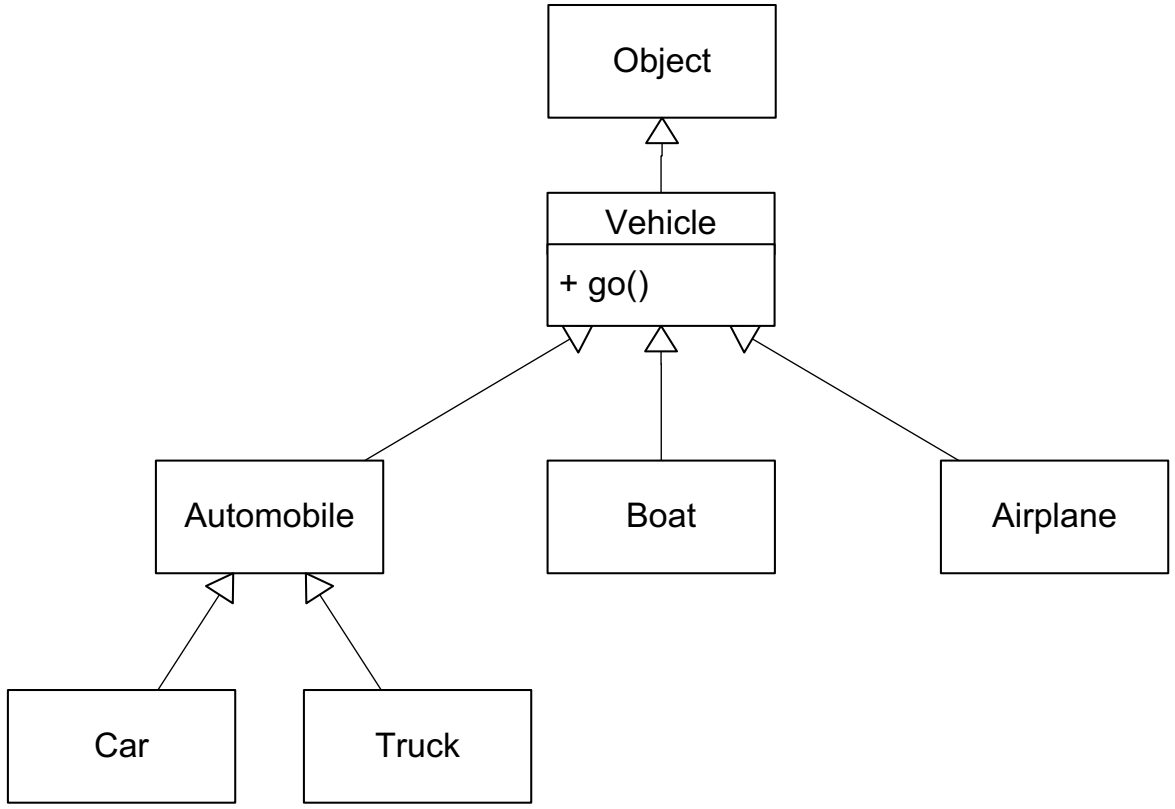
- Would probably never create the simple version of polymorphism
  - `Person emp = new Employee();`
- Get the same result (reference of one type referring to an object of another type) when you use either of the following:
  - Heterogeneous Collections
    - Collections (such as arrays or ArrayLists) of a parent type that contain children of different types
  - Polymorphic parameters
    - Parameters in a method call that expect a parent reference or object but receive a child of the expected type

City Simulation Example

- Create a simulation of a city, using an inheritance hierarchy of vehicles
- Will have different kinds of vehicles in the city that can all ‘go’
- Will start the simulation by placing vehicles in an array and calling a ‘go()’ method

- What type of polymorphism is this?

Vehicle Inheritance Hierarchy



Simulation Example 1

```
public class CitySimulation {
    public void run() {
        Vehicle [] vehicles = new Vehicle[6];
        vehicles[0] = new Car();
        vehicles[1] = new Car();
        vehicles[2] = new Truck();
        vehicles[3] = new Truck();
        vehicles[4] = new Boat();
        vehicles[5] = new Airplane();

        for(int i = 0; i < vehicles.length; i++) {
            vehicles[i].go()
        }
    }
}
```

Problems

- No common code to inherit for ‘go()’ method (so what should we write?)
- Even if there were common code, how would creators of subclasses know they need to override the ‘go() method?
- **Solution:** Abstract Method (which requires an Abstract Class)

Abstract Vehicle Class

```
public abstract class Vehicle {
 public abstract void go();
}
```

Abstract Methods

- Require containing class to be abstract
- Must be overridden in child classes unless the child is abstract
- A way to say a class has a behavior that will be defined in the subclasses
- Allows polymorphic method invocations of methods declared but not defined by the reference type

Polymorphic Method Invocation

- AKA: Polymorphic Operation

```
Vehicle v = new Car();
v.go();
```

- Allowable because…
  - Every class that has one or more abstract methods must be abstract
  - You can't create an instance of an abstract class

  So,
  - Vehicle references can only refer to vehicle's non-abstract child classes, so anything a vehicle reference can refer to, will have a non-abstract `go()` method

~~`Vehicle v = new Vehicle();`~~ **Illegal!**

Abstract Classes

- Cannot be instantiated
- Can be used as reference types (polymorphism)
- May have non-abstract methods
- Don’t have to have abstract methods
- Provide a guarantee:
  - If you have a non-null reference of an abstract type, it refers to an object that is not abstract and therefore has non-abstract implementations for all methods

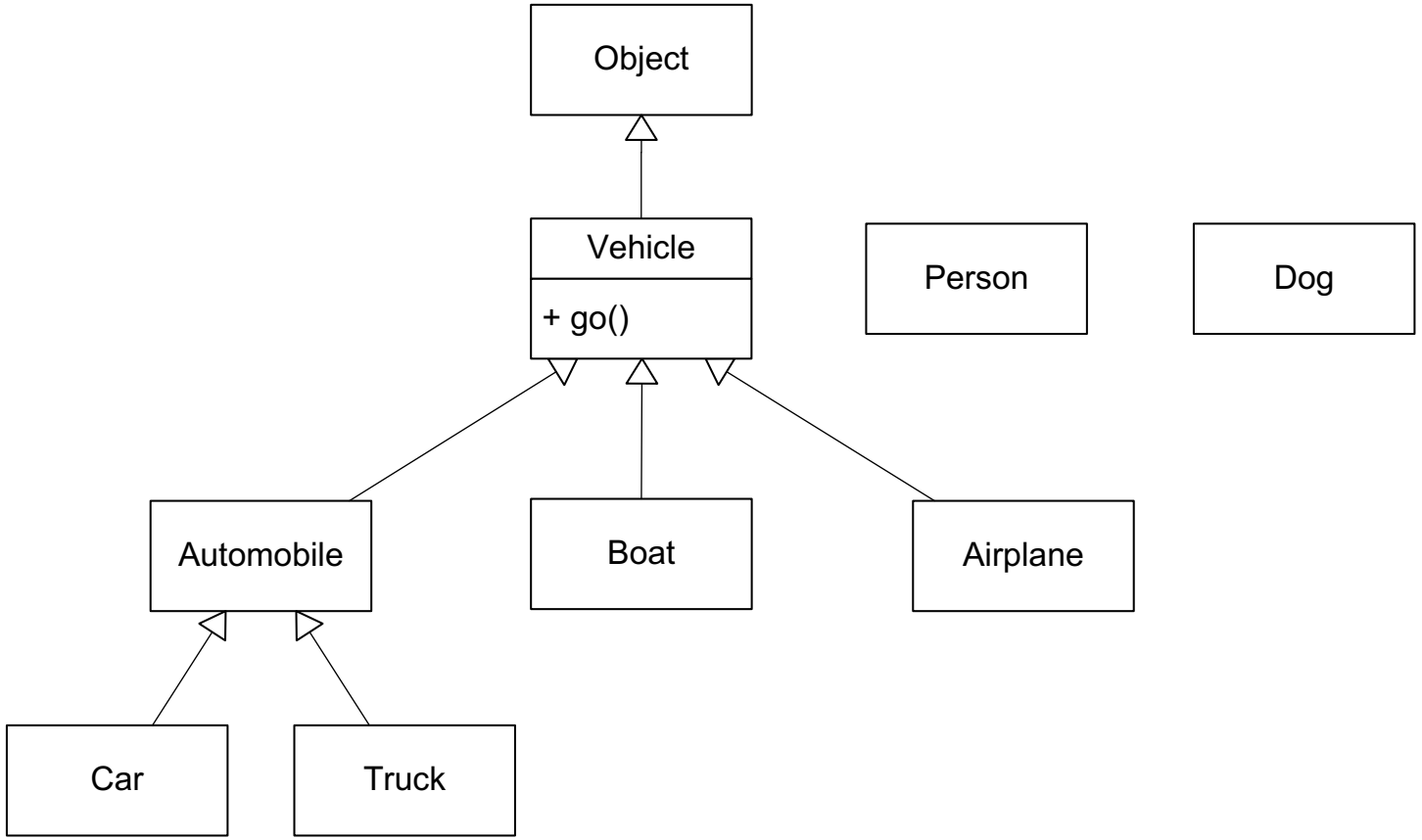
Simulation Example with Abstract Vehicle Class

```
public class CitySimulation {
    public void run() {
        Vehicle [] vehicles = new Vehicle[6];
        vehicles[0] = new Vehicle();  // Illegal if abstract
        vehicles[1] = new Car();
        vehicles[2] = new Truck();
        vehicles[3] = new Truck();
        vehicles[4] = new Boat();
        vehicles[5] = new Airplane();

        for(int i = 0; i < vehicles.length; i++) {
            // Guaranteed to invoke a real (non-abstract) method
            vehicles[i].go()
        }
    }
}
```

Updated Requirements

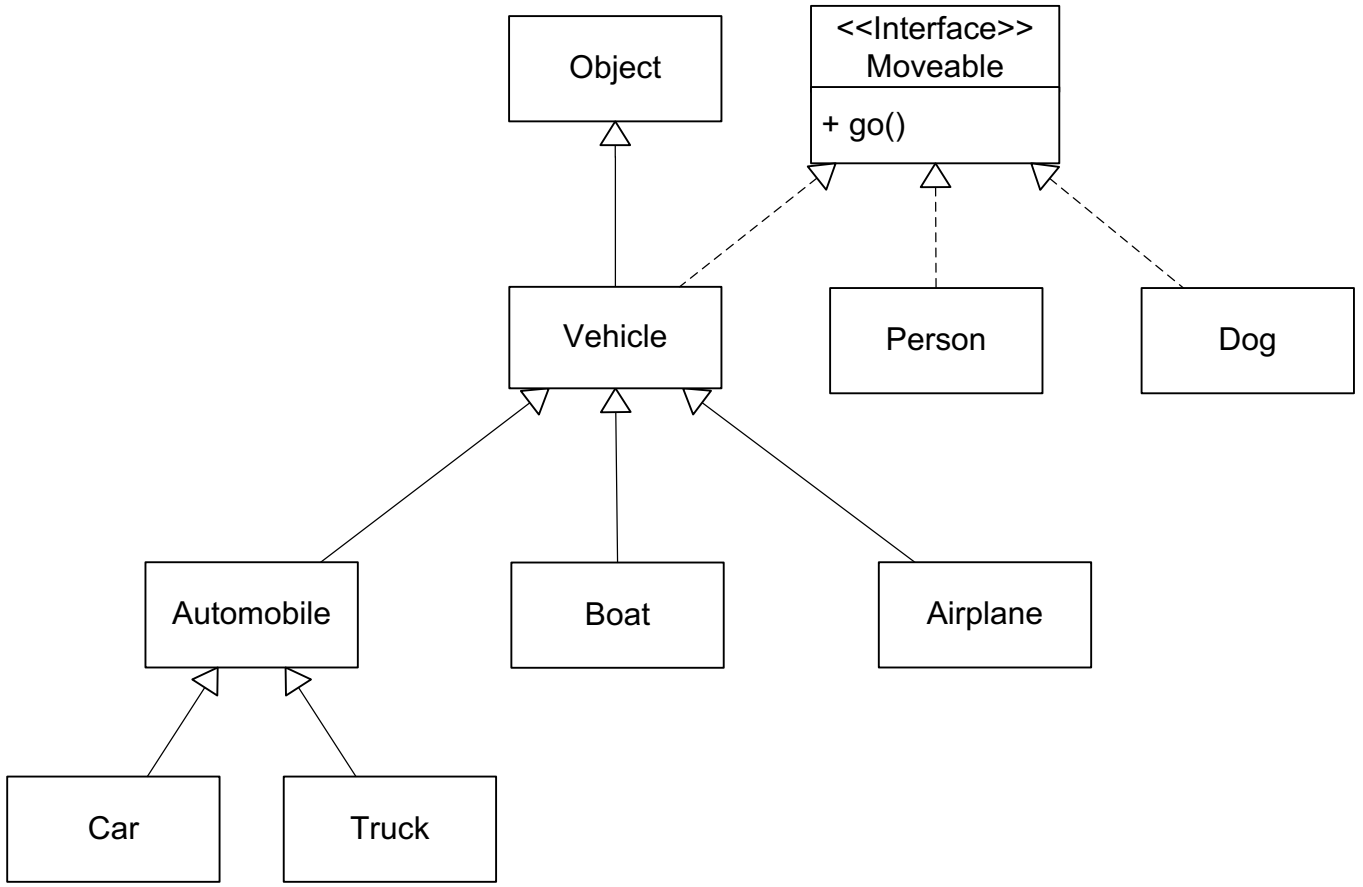
- Simulation must also contain people and dogs



How Do We Start the Simulation Now?

- Would like the people and dogs to start moving at the same time as the vehicles
- Can we put ‘Person’ and ‘Dog’ objects in our Vehicle array?
- Can we change the type of the array to ‘Object’ and then put them in?
- Need polymorphism (heterogeneous collection) without inheritance

Inheritance Hierarchy with Interfaces



Interfaces

- Cannot be instantiated
- Can be used as reference types (polymorphism)
- Can be used as collection (array) types (polymorphism)
- May **NOT** have non-abstract methods
  - All methods are abstract (with three exceptions in Java version 8 and later)
- All methods are public (whether you declare them as public or not)
  - With one exception in recent Java version 8 and later
- Provide same guarantee as abstract classes:
  - If you have a non-null reference of an interface type, it refers to an object that implements the interface and is not abstract and therefore has non-abstract implementations for all behaviors (methods)

Interfaces (cont.)

- Can implement any number of interfaces and still subclass some other class
- Breaks inheritance barrier of polymorphism
  - Provides a way to use polymorphism where an inheritance relationship does not exist
  - Examples:
    - `Moveable m = new Car();`
    - `Moveable m = new Person();`
    - `Moveable m = new Dog();`
    - ~~`Moveable m = new Moveable();`~~ **Illegal!**
    - ~~`Moveable m = new Vehicle();`~~ **Illegal!**

Interfaces (cont.)

- Can have constant variables
  - All variables are public, static, and final
- In Java 8 and Later:
  - Can have instance methods with bodies (must be declared as default)
    - `default void myDefaultMethod() {…}`
  - Can have static methods (with bodies)
  - Can have private methods (not inherited so only useful as helper methods to default methods)

Simulation Example with Interface

```
public class CitySimulation {
    public void run() {
        Moveable[] moveables = new Moveable[6];
        moveables[0] = new Car();
        moveables[1] = new Car();
        moveables[2] = new Truck();
        moveables[3] = new Truck();
        moveables[4] = new Person();
        moveables[5] = new Dog();

        for(int i = 0; i < moveables.length; i++) {
            moveables[i].go()
        }
    }
}
```

Creating an Interface

```
public interface Moveable {
    public void go();
}
```

Implementing an Interface

```
public class Person implements Moveable {

    public void go() {
      // Code to make person go
    }
}
```

Implementing an Interface with An Abstract Class in Java

```
public abstract class Vehicle implements
 Moveable {


}
```

Extending a Class and Implementing Multiple Interfaces

```
public class Employee extends Person implements
  Moveable, Comparable{

    public void go() {
      // Code to make person go
    }

    public int compareTo(Object obj) {
      // Code to compare two employees
    }
}
```