Polymorphism I

CMSC 202

Topics

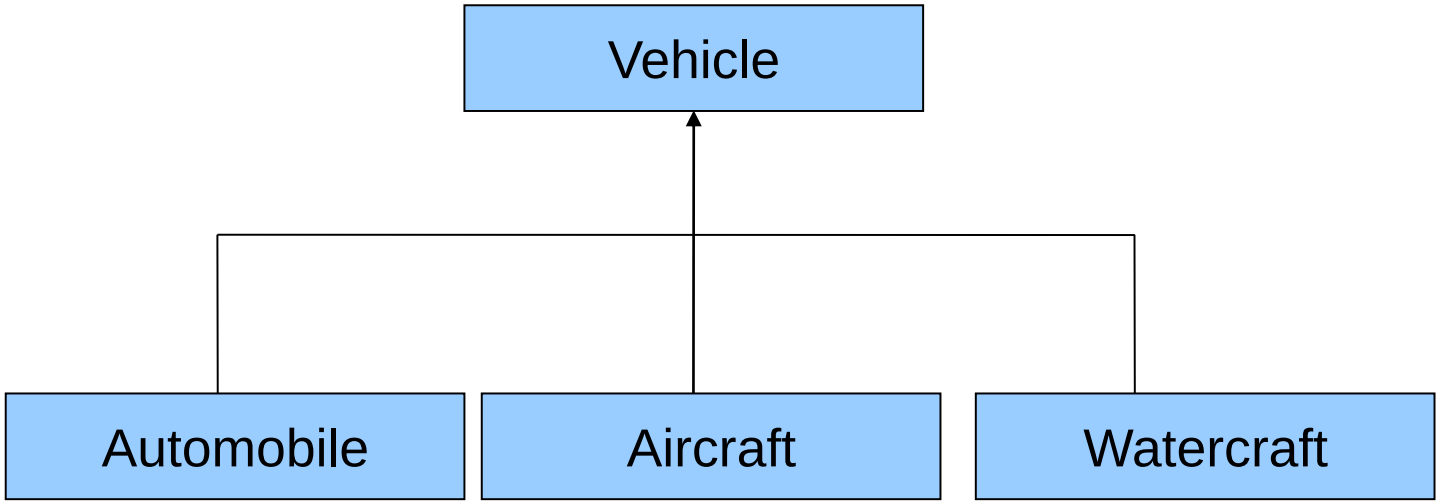
- Binding (early and late)
- Upcasting and downcasting
- Extensibility
- The final modifier with
  - methods
  - classes

Introduction to Polymorphism

- Object-oriented programming mechanisms
  - Encapsulation - data and methods together
  - Inheritance - extending a class for specialization
  - Polymorphism

- Polymorphism
  - The ability to associate many meanings with one method name.
  - Accomplished through a mechanism known as **`late binding`** or **`dynamic binding`**.

Vehicle Hierarchy



Identifying Classes of Vehicles

```
public class Vehicle {
public void identify(){ System.out.println("Vehicle"); }
}


public class Automobile extends Vehicle {
public void identify(){ System.out.println("Automobile"); }
}

public class Aircraft extends Vehicle {
public void identify(){ System.out.println("Aircraft"); }
}

public class Watercraft extends Vehicle {
public void identify(){ System.out.println("Watercraft"); }
}
```

- We have implemented the identify() method defined in the base class and overidden in the derived classes. Each is a more specific definition of the base class' method.

The Vehicle Classes

In the VehicleDemo, we ask each Vehicle to identify itself.

- This is a poor example of OOP as we will see...

```
public class VehicleDemo {
   public void identifyYourself(Automobile a){
      a.identify();
   }
   public void identifyYourself(Aircraft a){
      a.identify();
   }
   public void identifyYourself(Watercraft a){
      a.identify();
   }

  public static void main(String[] args){
    Automobile m = new Automobile();
    Watercraft w = new Watercraft();
    Aircraft a = new Aircraft();

    VehicleDemo demo = new VehicleDemo();
    demo.identifyYourself(m);
    demo.identifyYourself(a);
    demo.identifyYourself(w);
  }
}
```

Output

```
Automobile
Aircraft
Watercraft
```

Problems with VehicleDemo?

- The VehicleDemo class contains a type-specific version of identifyYourself for each type of Vehicle.
- What if we add more types of Vehicles?
- Wouldn’t it be nice to write just one identifyYourself method that works for all Vehicles?

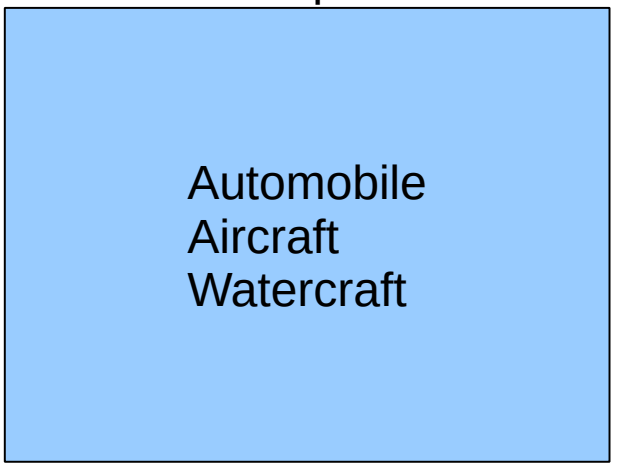
NewVehicleDemo

```
public class NewVehicleDemo {
    public void identifyYourself(Vehicle v){
        v.identify();
    }

    public static void main(String[] args){
        Automobile m = new Automobile();
        Watercraft w = new Watercraft();
        Aircraft a = new Aircraft();

        NewVehicleDemo demo = new NewVehicleDemo();
        demo.identifyYourself(m);
        demo.identifyYourself(a);
        demo.identifyYourself(w);
    }
}
```

Output



How Does NewVehicleDemo work?

- Associating the appropriate method definition with the method invocation is known as binding.
- Early binding occurs when the method definition is associated with its invocation when code is compiled.
  - With early binding, the method invoked is determined by the reference variable type.
- How can the compiler know which Vehicle's identify method to call in identifyYourself? It can't!

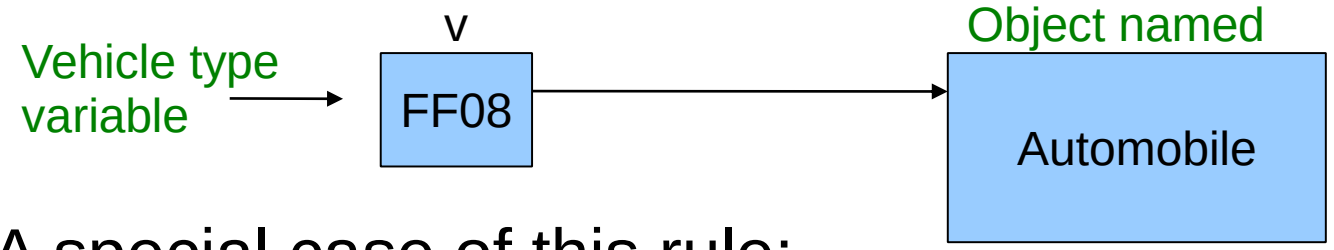
Late Binding

- The solution is to use late (dynamic) binding.
- Late binding
  - The appropriate method definition is associated with its invocation *at run-time*.
  - The method invoked is determined by the *type of object* to which the variable refers, **NOT** by the type of the reference variable.
- Java uses late binding for all methods except
  - `final`,
  - private (which are implicitly final), and
  - static methods.

An Object Knows the Definitions of Its Methods

- The type of a class variable determines which method names can be used with the variable.
  - However, the object named by the variable determines which definition with the same method name is used.

```
Vehicle v = new Automobile();
```

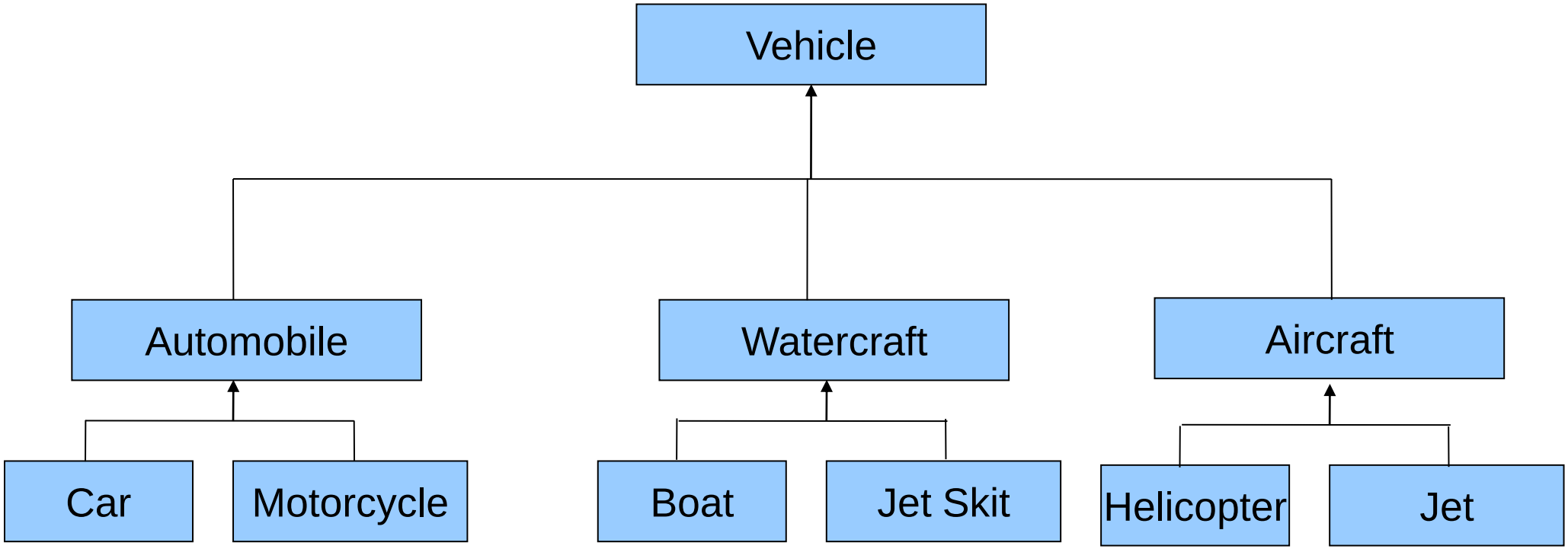


- A special case of this rule:
  - The type of a class variable determines which method names and members the compiler recognizes for the parameter.
  - The argument determines which definition of the method name is used.

Using Polymorphism

- How do we take advantage of Polymorphism?
  - Write code to talk to base class objects (e.g. use base class references as method parameters)
  - Late binding will ensure that the appropriate method definition is used, even if a reference to a derived class is passed to the method.

More Vehicles



Extensibility

- Suppose more Vehicles were added to the hierarchy as shown in the previous diagram.
- All of these new classes work correctly with the old, unchanged identify() method of the NewVehicleDemo class because identifyYourself()'s parameter is a base class reference type(Vehicle).
- In a well designed OOP program, most of your methods will follow the model of identifyYourself() and communicate with a base class reference and let late binding and polymorphism determine which class' identify() method to call.
- Such a program is called ***extensible*** because you can add new functionality by deriving new classes from the base class without changing existing code.

The `final` Modifier

- A *method* marked **`final`** indicates that it cannot be overridden with a new definition in a derived class.
  - If **`final`**, the compiler can use early binding with the method.

```
public final void someMethod() { . . . }
```

- A *class* marked **`final`** indicates that it cannot be used as a base class from which to derive any other classes.

Late Binding with **`toString`**

- Because all classes created extend from Object, our classes inherit the toString method and can be printed using

```
System.out.println( );
```

as in this code snippet:

```
Vehicle auto = new Automobile( );
System.out.println(auto);
```

- This works because of late binding.

Late Binding with `toString`

- One definition of the method `println` takes a single argument of type `Object`:

```
public void println(Object theObject)
{
  System.out.println(theObject.toString());
}
```

  - In turn, It invokes the version of `println` that takes a `String` argument.

Note that the `println` method was defined before the Vehicle class existed.

- Because of late binding, the `toString` method from the Vehicle class is used, not the `toString` from the `Object` class.

Upcasting and Downcasting

- **Upcasting** occurs when an object of a derived class is assigned to a variable of a base class (or any ancestor class).

```
Vehicle v;                                  // base class
Automobile auto = new Automobile();         // derived class
v = auto;                                   // upcasting
v.identify();                          // prints automobile
```

Or we could do something equivalent, such as

```
Vehicle v = new Automobile();
```

- Because of late binding, **`identify()`** uses the definition of **`identify()`** given in the **`Automobile`** class.

Upcasting and Downcasting

- **Downcasting** occurs when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class).
  - Downcasting must be done very carefully.
  - In many cases it doesn't make sense, or is illegal:

```
void doSomething(Vehicle v1) {
    Automobile a1 = (Automobile) v1;  // could generate an error
    a1 = v1;                          // will generate an error
}
```

  - There are times when downcasting is necessary; e.g., inside the **`equals`** method for a class.
    - How can we make sure a Vehicle is an Automobile?