Inheritance

Syntax, is-a, extends, terminology, examples, class diagram, overriding

Reusing code

Avoid rewriting the same code for new applications that you've written before.

- Factor out what's common.
- Customize what's different.
- One benefit: modify in only one place if there's a bug.

Code reuse between classes comes from:

- Composition – *has-a* relationships
- Inheritance – *is-a* relationships

Factoring out what's common

A national transportation system consists in part of vehicles that move people and freight from one place to another. Every vehicle:

  - Has mileage
  - Is either traveling or not traveling
  - Can stop
  - Can go

- Highway vehicles also have licenses.
- Busses, which are highway vehicles, have passengers who get on and off.

This lecture builds on these concepts using inheritance.

Superclasses and subclasses

A **class hierarchy** consists of:

- A **superclass,** which is the base of the hierarchy.
- **Subclasses,** which are derived from the superclasses.

Java supports **single inheritance**:

- A class can have only one superclass (parent).
- A superclass can have many subclasses (children).

Inheritance terminology

We will create:

- A **base class** that will serve as a **parent** to other classes.
  Base class = parent = superclass
- **Child classes** that **extend** the parent.
  Child class = subclass
- Methods in the children that **override** those defined in the parent in order to customize them.

A class hierarchy consists of a superclass and its descendants.

More inheritance terminology

Look for the following terms too:

- *is-a* (vs *has-a*)
- extends
- protected
- super
- hiding
- @Override

Example base class

```
public class Vehicle {
   protected boolean traveling;
   private int mileage;

   public Vehicle() { }
   public Vehicle(int mileage) {
     this.mileage = mileage;
   }

   public void go(int distance) {
      traveling = true;
      mileage += distance;
    }
    public void stop() {
      traveling = false;
    }
    public int getMileage() {
       return mileage;
   }
}
```

Extending the base

```
public class HighwayVehicle extends Vehicle {
    private String license;
    private final int MAX_MILES_PER_DAY = 750;

    public HighwayVehicle(String license, int previous) {
        super(previous);
        this.license = license;
    }
    public HighwayVehicle(String license) {
        this.license = license;
    }

    public String getLicense() { return license; }

    @Override
    public void go(int distance) {
        if (distance < MAX_MILES_PER_DAY)
            super.go(distance);
    }
    public void go() { go(100); }  // Overloading go()
}
```

Vehicle terminology

`HighwayVehicle`

- **extends** `Vehicle`
- is a **child** of `Vehicle`
- is **derived** from `Vehicle` (derived class)
- is a **subclass** of `Vehicle`

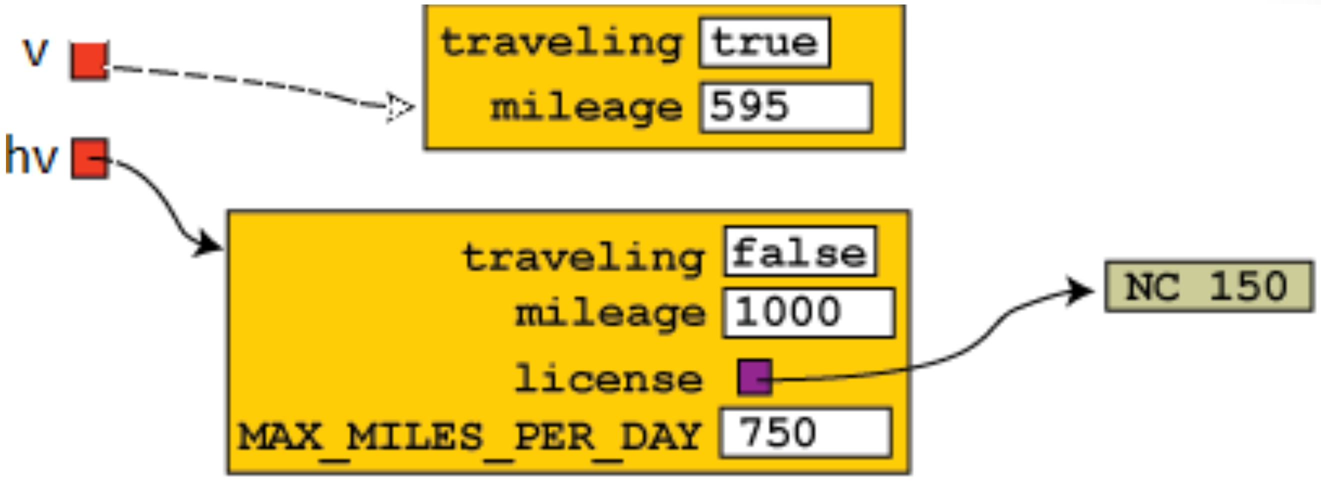
A `HighwayVehicle` ***is-a*** `Vehicle`. It can do everything a `Vehicle` can do.

`Vehicle`

- is a **base** class
- is the **parent** of `HighwayVehicle`
- is the **superclass** of `HighwayVehicle`

Instances and memory

```
Vehicle v = new Vehicle();
v.go(595);
HighwayVehicle hv = new HighwayVehicle("NC 150", 600);
hv.go(400);
hv.stop();
```



`hv` *is-a* `Vehicle`. It has all of the fields (data members) that `v` has, plus more. It also overrides the method `go()`.

What does a child class inherit?

A child inherits its parent’s public and protected members. (It does not inherit private members.)

- Inherited members can be used directly (same as in parent).
- The child can redefine parent class members:
  - Redefining the same non-static method as in the parent (with the same signature) is called **overriding**.
  - Redefining the same data member as in the parent is called **hiding**.
  - Redefining the same static method as in the parent is also called **hiding**.
- Constructors are NOT inherited.

A child can declare new methods and data that are not in the parent.

Overloading, overriding, and hiding

- **Overloading**. Declaring a method (in the same class hierarchy) with the same name but with different signatures. You can overload constructors or ordinary methods.
- **Overriding**. Declaring a method in a child class with the same signature as one in the base.
- **Hiding**. Overriding a static method, declaring an instance variable of the same name in the case of data.

Overriding restrictions and conventions

Static.

- If a method is declared static in the base, only another static method can override it.
- If a method is declared static in the child, any method it overrides must be declared static.

Access control. A method that overrides a base class method cannot have more restrictive access than what was declared in the base.

@Override. Annotation to tell the compiler that there must be a method in the base class that this one overrides.

Declaring as one type, instantiating as another

```
Vehicle myCar; // Declared type is Vehicle
myCar = new HighwayVehicle(“NC 6543”, 20000);
```

Legal. Since a `HighwayVehicle` *is-a* `Vehicle`, this is valid.

```
HighwayVehicle myTruck;
myTruck = new Vehicle();  // Oops!
```

Illegal. Some `Vehicles` may not be `HighwayVehicles`.

Visibility and inheritance

A class contains all of the fields of the parent.

If parent declared the field as private:

- The child still contains the field (`mileage` is still part of each `HighwayVehicle`).
- The child, just like all other classes, cannot access the field directly.

A member (field or method) that that the parent declares as **protected**:

- Can be accessed directly by its child classes. (The child "inherits" these members.)
- Can also be accessed directly by any class in the same package – sometimes not a desired result.

A member (field or method) that that the parent declares as public is also considered as a public member in the child class.

Constructor call basics

Classes are instantiated/built from the inside out.

- When you call a constructor for a child class, the parent part is built first.
- Building the parent requires a call to a parent constructor.
- Any explicit call to a parent constructor must be the first statement executed in the child constructor.
- Explicit calls to the parent use the keyword **super**.
- The call to a parent constructor can be implicit. That call is made prior to any other code in the child constructor.

Any implicit call requires that the parent class have a null constructor.

Constructors are not inherited!

Using `super` and `this`

The keyword `this` in class `A`:

- Can be used in `A` to modify any of the non-static members of `A`. [Often required to resolve name conflicts with parameters or local variables.]
- Can be used in `A` to call a constructor of `A`.

If class `B` is a subclass of class `A`, the keyword `super`:

- Can be used in `B` to call methods of `A` that it overrides.
- Can be used in `B` to call a constructor of `A`.
  Any call to `super` in a constructor of `B` must be the first statement in the constructor.

Constructor definitions

`Vehicle` constructors

```
public Vehicle(int mileage) {
    this.mileage = mileage;
}

public Vehicle() { }
```

`HighwayVehicle` constructors

```
public HighwayVehicle(String license, int previous) {
    super(previous);
    this.license = license;
}

public HighwayVehicle(String license) {
    this.license = license;
}
```

Extending the hierarchy down

```
public class Bus extends HighwayVehicle {
    private int passengers;

    public Bus(String license) { super(license);  }
    public void boardPassenger() {
        if (!traveling)
            passengers++;
    }
    public void dropOffPassenger() {
        if (!traveling)
            passengers--;
    }
    @Override
    public void go(int x) {
       super.go(x);
       passengers = 0;
    }
}
```

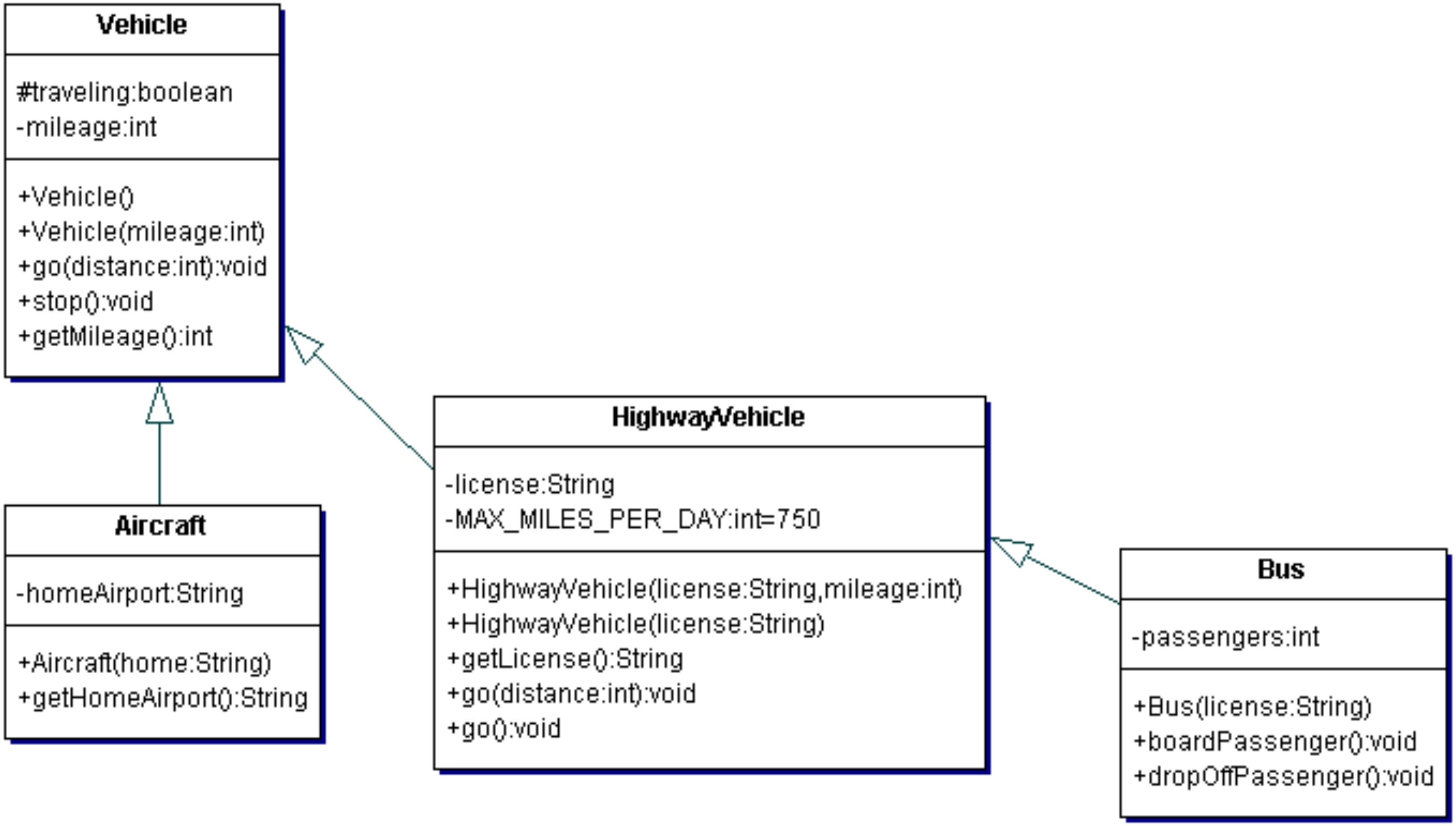
Extending the hierarchy across

```
public class Airplane extends Vehicle{
    private String homeAirport;

    public Airplane (String homeAirport) {
        this.homeAirport = homeAirport;
    }

    // More members here
}
```

UML diagram of the class hierarchy



Vehicle

#traveling:boolean
-mileage:int

+Vehicle()
+Vehicle(mileage:int)
+go(distance:int):void
+stop():void
+getMileage():int

Aircraft

-homeAirport:String

+Aircraft(home:String)
+getHomeAirport():String

HighwayVehicle

-license:String
-MAX_MILES_PER_DAY:int=750

+HighwayVehicle(license:String,mileage:int)
+HighwayVehicle(license:String)
+getLicense():String
+go(distance:int):void
+go():void

Bus

-passengers:int

+Bus(license:String)
+boardPassenger():void
+dropOffPassenger():void

Class hierarchy code

```
HighwayVehicle h = new HighwayVehicle(); //2 constructor calls

Bus b = new Bus("NC-bus 123");  // 3 constructor calls

b.go(595); // calls HighwayVehicle.go(). traveling is now true
b.stop();  // calls Vehicle.stop(). traveling is now false
b.boardPassenger();   // there is now one passenger
```

Inheritance vs Composition

- Use inheritance when similarities between classes make them fall into natural hierarchies.
- Do not use inheritance when a subclass cannot be substituted everywhere for a superclass.
- Use composition when one class can provide services for another class.
- Inheritance is considered to be "more brittle" than composition (harder to customize and modify). When you have a choice, pick composition!