DM550 / DM857
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM550/

http://imada.sdu.dk/~petersk/DM857/

ADVANCED OBJECT-ORIENTATION

Object-Oriented Design

- classes often do not exist in isolation from each other
- a vehicle database might have classes for cars and trucks
- in such situation, having a common superclass useful
- Example:

```
public class Vehicle {
    public String model;
    public int year;
    public Vehicle(String model, int year) {
        this.model = model;  this.year = year;
    }
    public String toString() {return this.model+" from "+this.year;}
}
```

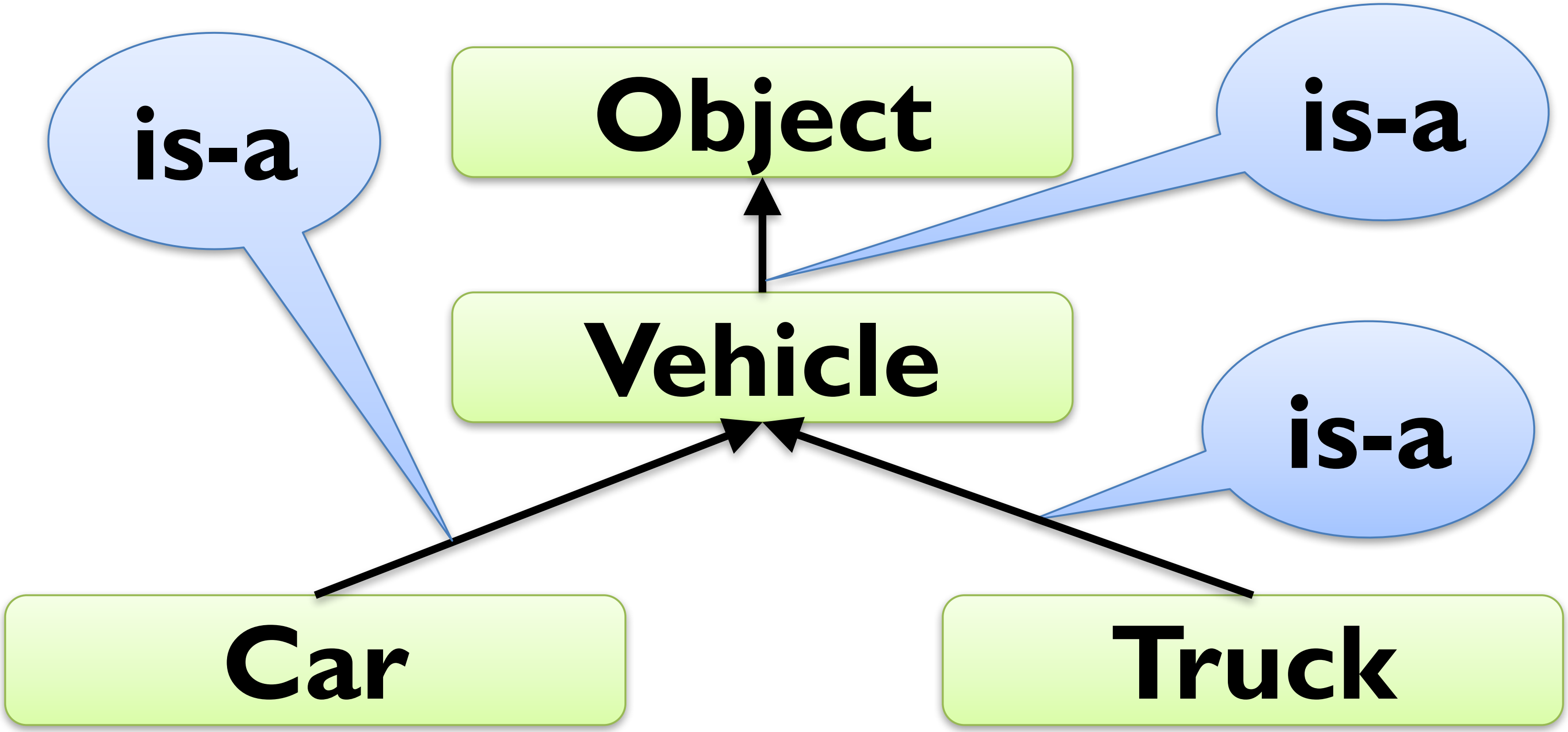
Extending Classes

- Car and Truck then *extend* the Vehicle class
- Example:

```
public class Car extends Vehicle {
    public String colour;
    public Car(string model, int year, String colour) {
        this.colour = colour;      // this makes NO SENSE
    }
    public String toString() {  return this.colour;  }
}
public class Truck extends Vehicle {
    public double maxLoad;
    …  }
```

Class Hierarchy

- class hierarchies are parts of class diagrams
- for our example we have:



Abstract Classes

- often, superclasses should not have instances
- in our example, we want no objects of class Vehicle
- can be achieved by declaring the class to be *abstract*
- Example:

```
public abstract class Vehicle {
    public String model;
    public int year;
    public Vehicle(String model, int year) {
        this.model = model;  this.year = year;
    }
    public String toString() {return this.model+" from "+this.year;}
}
```

Accessing Attributes

- attributes of superclasses can be accessed using “this”
- Example:

```
public class Car extends Vehicle {
    public String colour;
    public Car(string model, int year, String colour) {
        this.model = model;  this.year = year;  this.colour = colour;
    }
    public String toString() {
        return this.colour+" "+this.model+" from "+this.year;
    }
}
```

Accessing Superclass

- methods of superclasses can be accessed using “super”
- Example:

```
public class Car extends Vehicle {
    public String colour;
    public Car(String model, int year, String colour) {
        this.model = model;  this.year = year;  this.colour = colour;
    }
    public String toString() {
        return this.colour+" "+super.toString();
    }
}
```

Superclass Constructors

- constructors of superclasses can be accessed using “super”
- Example:

```
public class Car extends Vehicle {
    public String colour;
    public Car(string model, int year, String colour) {
        super(model, year);
        this.colour = colour;
    }
    public String toString() {
        return this.colour+" "+super.toString();
    }
}
```

Abstract Methods

- abstract method = method declared but not implemented
- useful in abstract classes (and later interfaces)
- Example:

```
public abstract class Vehicle {
    public String model;
    public int year;
    public Vehicle(string model, int year) {
        this.model = model;  this.year = year;
    }
    public String toString() {return this.model+" from "+this.year;}
    public abstract double computeResaleValue();
}
```

Implementing Abstract Methods

- abstract methods need to be implemented in concrete subclasses
- use same function signature, but without "abstract"
- Example:

```
public class Car extends Vehicle {
    ...
    public double computeResaleValue() {
        double value = 100000 * (this.model.startsWith("Audi") ? 6 : 4);
        value *= (this.year-2000)/20;
        return value;
    }
}
```

Interfaces

- different superclasses could have different implementations
- to avoid conflicts, classes can only extend one (abstract) class
- interfaces = abstract classes without implementation
- only contain public abstract methods (abstract left out)
- no conflict possible with different interfaces
- Example:

```
public interface HasValueAddedTax {
    public double getValueAddedTax(double percentage);
}
public class Car implements HasValueAddedTax {
    public double getValueAddedTax(double p) {  return 42000;  }
    …  }
```

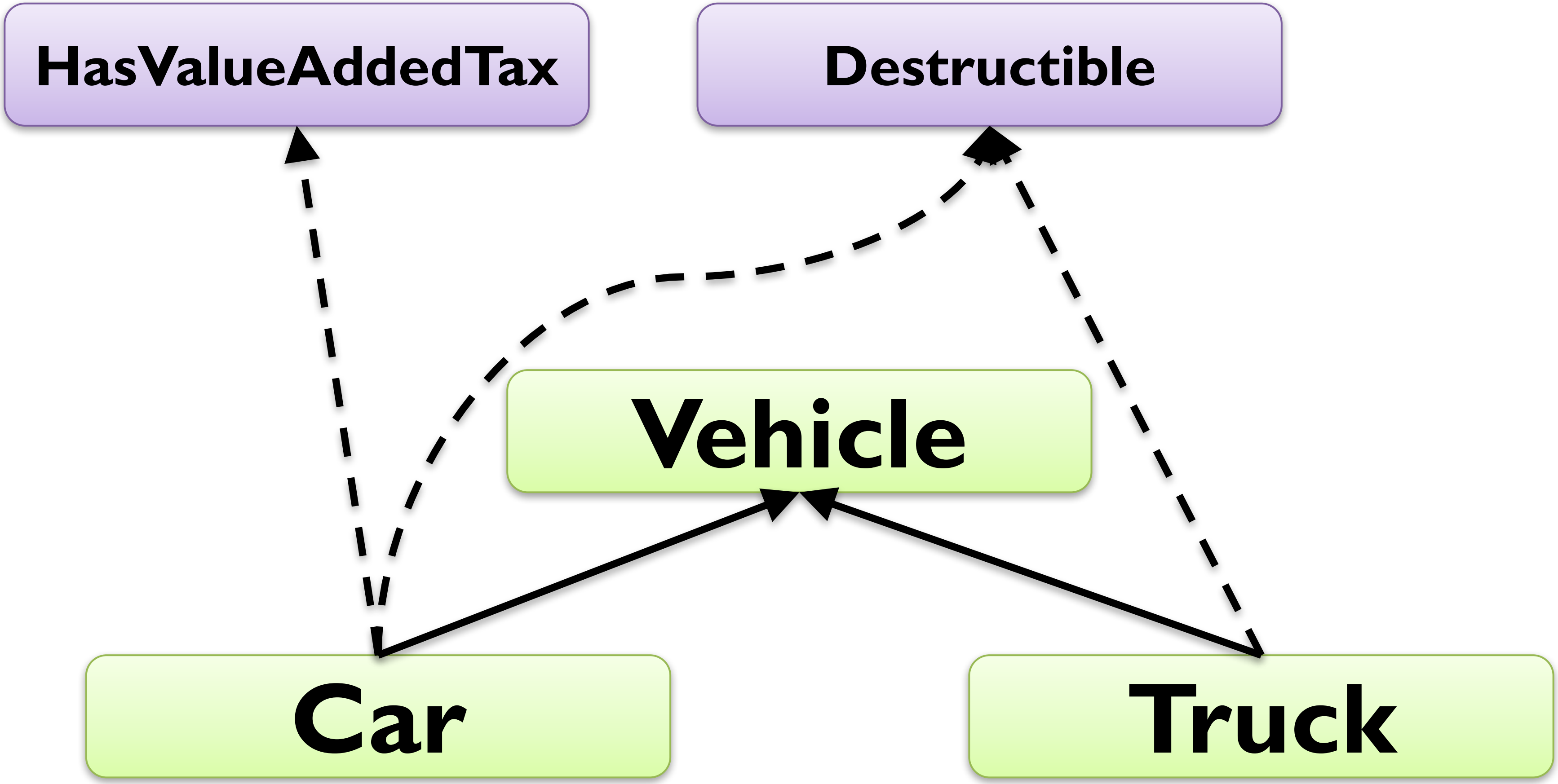
Interfaces

- Example:

```
public interface HasValueAddedTax {
    public double getValueAddedTax(double percentage);
}
public interface Destructible {
    public void destroy();
}
public class Car implements HasValueAddedTax, Destructible {
    public double getValueAddedTax(double p) {  return 42000;  }
    public void destroy() {  this.model = "BROKEN";  }
    …
}
```

Interface and Class Hierarchy

- interfaces outside normal class hierarchy



Inner Classes

- classes and interfaces can be nested
- inner class = class contained in another class
- Example:

```
public abstract class Vehicle {
    …
    public interface Destructible {
        public void destroy();
    }
    public class Car extends Vehicle implements Destructible {
        …
    }
}
```

Local Classes

- classes and interfaces can be declared in function bodies
- local class = class contained in the body of a function or method
- Can obviously not be public

- Example:

```
public static void main(String[] args) {
   class Bicycle implements Destructible {
        public void destroy() { System.out.println("Ouch!"); }
   }
   new Bicycle().destroy();
}
```

Anonymous (Sub-)Classes

- possible to create anonymous classes
- often used to instantiate abstract classes or interfaces
- body of class defined after constructor call
- Example:

```
public class FarmVillain {
    public static void main(String[] args) {
        Vehicle x = new Vehicle("Volvo T230",1971) {
            public double computeResaleValue() {
                return 25000;
            }
        };
}   }
```

Final Modifier

- variables only assigned once can be declared final
- multiple assignment to final variable results in compiler error
- Example:

```
final int x;
x = 42; // ok
x = 23; // ERROR
```

Local and Anonymous Classes

- local and anonymous classes can access local variables and parameters IF they are final
- Example:

```
public static makeTractor(String model, int year, final int base) {
    final double factor = (year-1920)/100;
    return new Vehicle(model,year) {
        public int computeResaleValue() {
            return base*factor;
        }
    };
}
```