

Inheritance

- Reuse of code
- Extension and intension
- Class specialization and class extension
- Inheritance
- The **protected** keyword revisited
- Inheritance and methods
- Method redefinition
- The *composite design pattern*
 - A widely example using inheritance
- Finally, the **final** keyword

How to Reuse Code?

- Write the class completely from scratch (one extreme)
 - What some programmers always want to do!
- Find an existing class that exactly match your requirements (another extreme)
 - The easiest for the programmer!
- Built it from well-tested, well-documented existing classes
 - A very typical reuse, called composition reuse!
- Reuse an existing class with inheritance
 - Requires more knowledge of the existing classes than composition reuse.
 - Today's main topic.

Composition is “flirting”, inheritance is “meet the parents”!

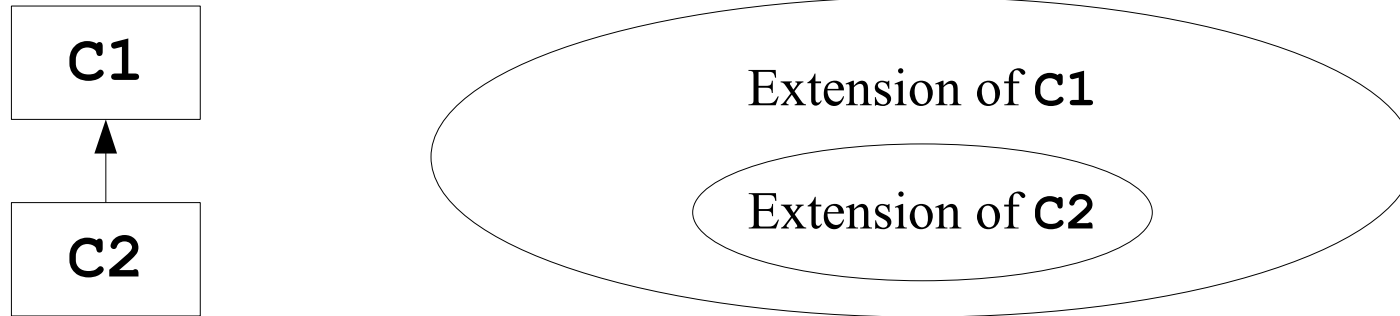
Class Specialization

- In *specialization* a class is considered an *Abstract Data Type* (ADT).
- The ADT is defined as a set of coherent values on which a set of operations (methods) are defined.
- A specialization of a class **C1** is a new class **C2** where
 - The instances of **C2** are a subset of the instances of **C1**.
 - Operations defined of **C1** are also defined on **C2**.
 - Operations defined on **C1** can be *redefined* in **C2**.



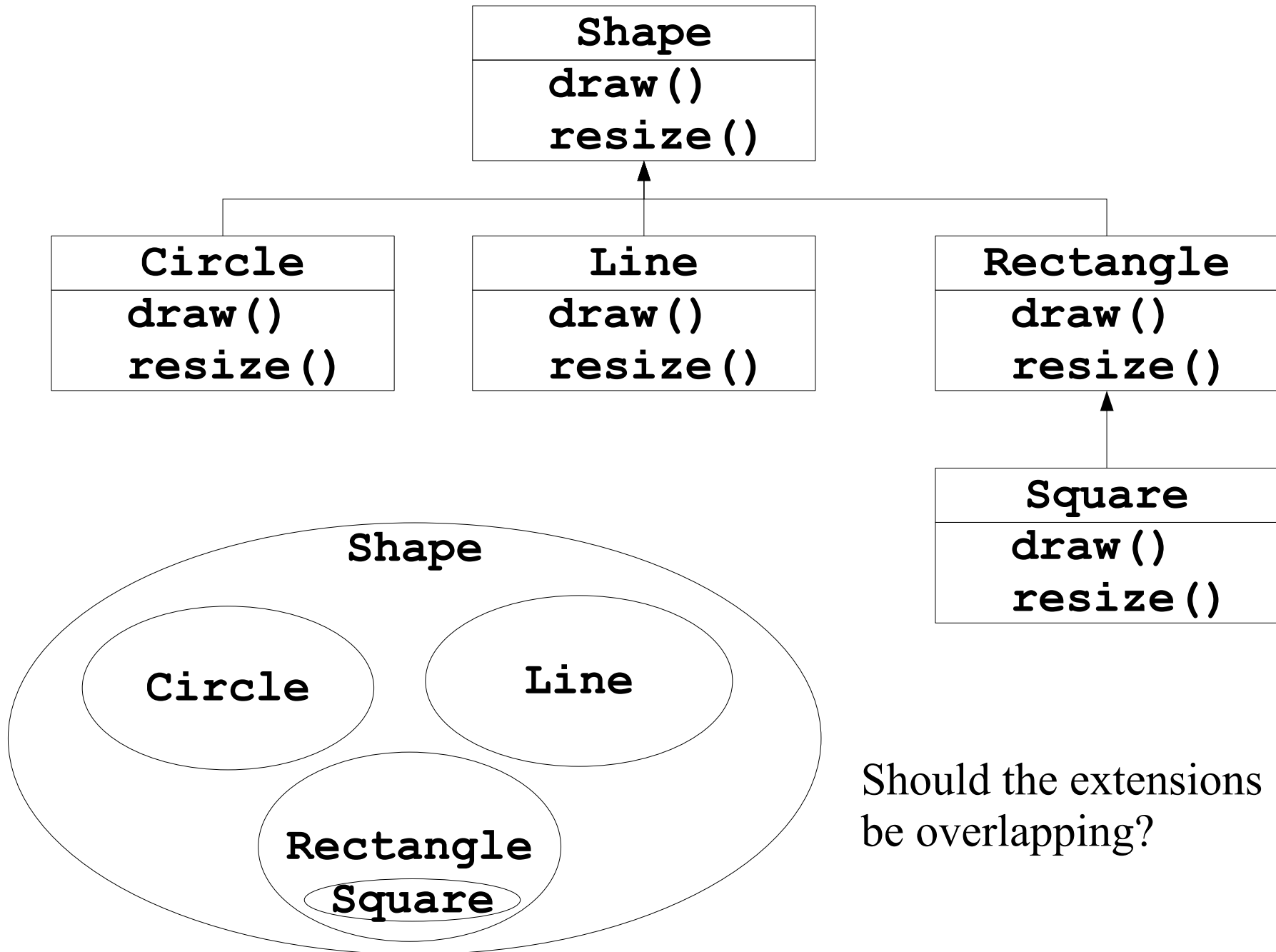
Extension

- The *extension* of a specialized class **C2** is a subset of the extension of the general class **C1**.



- “is-a” Relationship
 - A **C2** object is a **C1** object (but not vice-versa).
 - There is an “is-a” relationship between **C1** and **C2**.
 - We will later discuss a “has-a” relationship

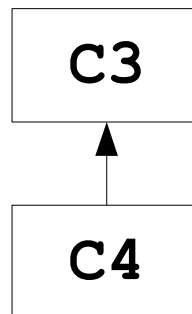
Class Specialization, Example



Should the extensions be overlapping?

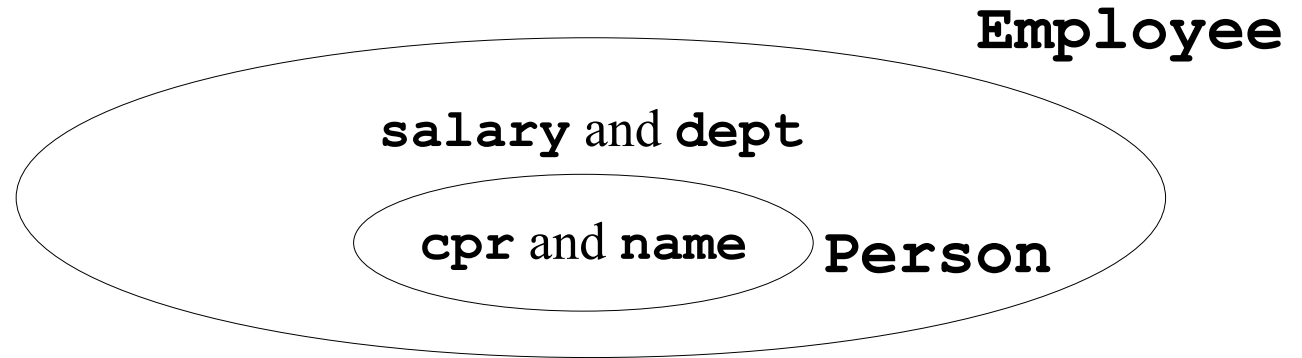
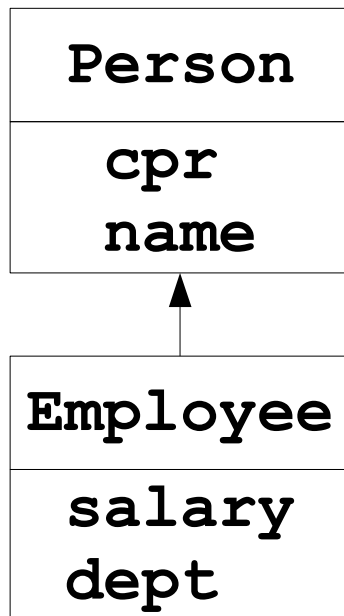
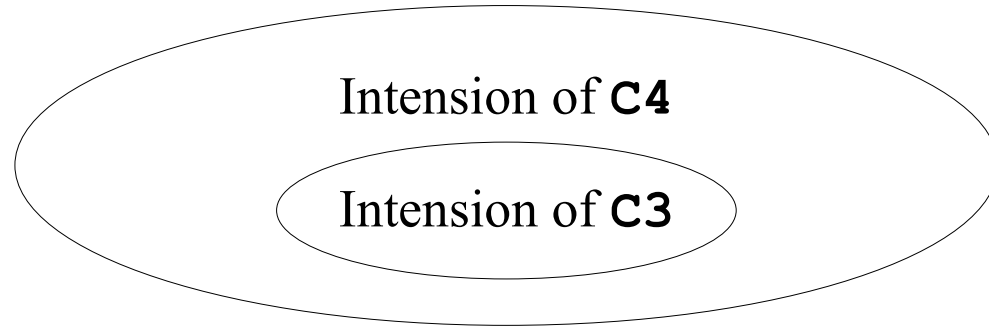
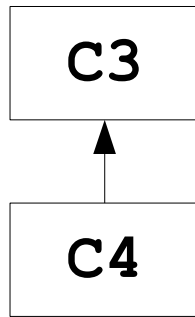
Class Extension

- In *class extension* a class is considered a *module*.
- A module is a syntactical frame where a number of variables and method are defined, found in, e.g., Modula-2 and PL/SQL.
- Class extension is important in the context of *reuse*. Class extension makes it possible for several modules to share code, i.e., avoid to have to copy code between modules.
- A class extension of a class **C3** is a new class **C4**
 - In **C4** new properties (variables and methods) are added
 - The properties of **C3** are also properties of **C4**



Intension

- The *intension* of an extended class **C4** is a superset of the intension of **C3**.

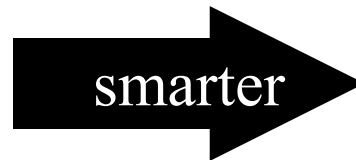
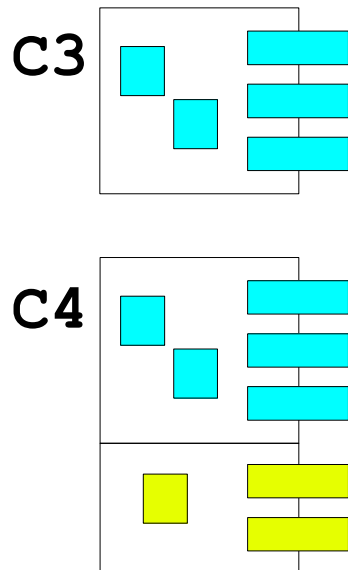


Inheritance

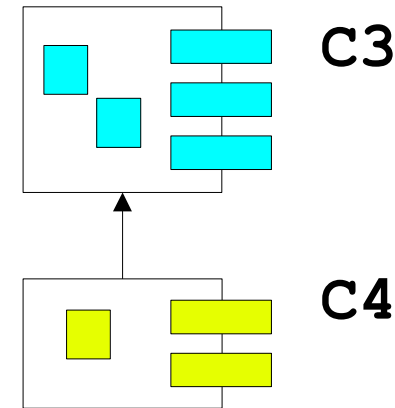
- Inheritance is a way to derive a new class from an existing class.
- Inheritance can be used for
 - Specializing an ADT, i.e., class specialization
 - Extending an existing class, i.e., class extension
 - Often both class specialization and class extension takes place when a class inherits from an existing class.

Module Based vs. Object Oriented

Module based



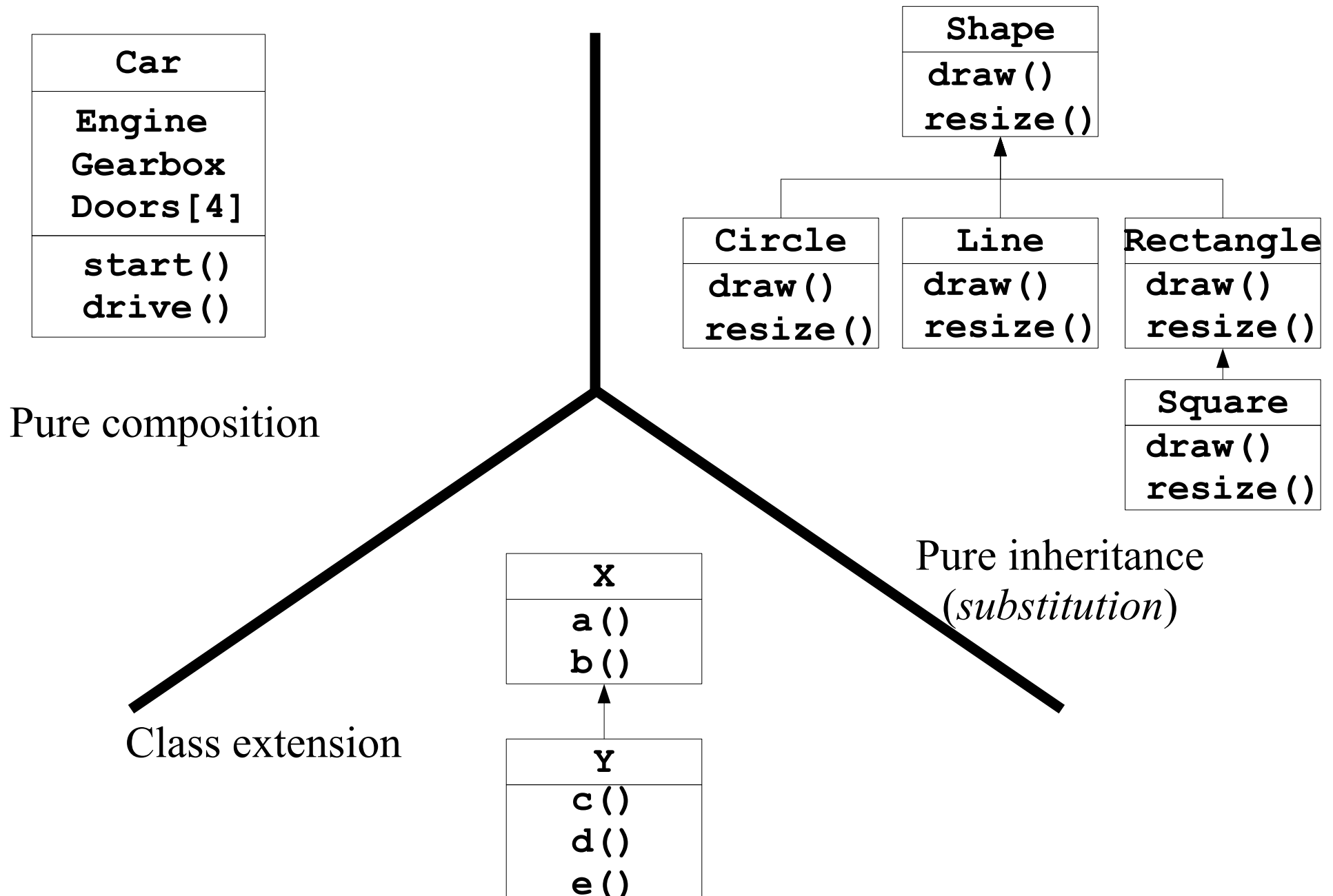
Object oriented



- Class **C4** is created by *copying* **C3**.
- There are **C3** and **C4** instances.
- Instance of **C4** have all **C3** properties.
- **C3** and **C4** are totally separated.
- Maintenance of **C3** properties must be done *two* places
- Languages, e.g., Ada, Modula2, PL/SQL

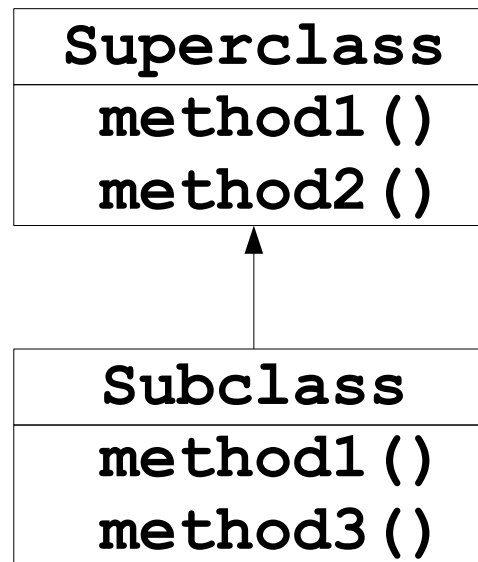
- Class **C4** *inherits* from **C3**.
- There are **C3** and **C4** instances.
- Instance of **C4** have all **C3** properties.
- **C3** and **C4** are closely related.
- Maintenance of **C3** properties must be done in *one* place.
- Languages, C++, C#, Java, Python, Smalltalk

Composition vs. Inheritance



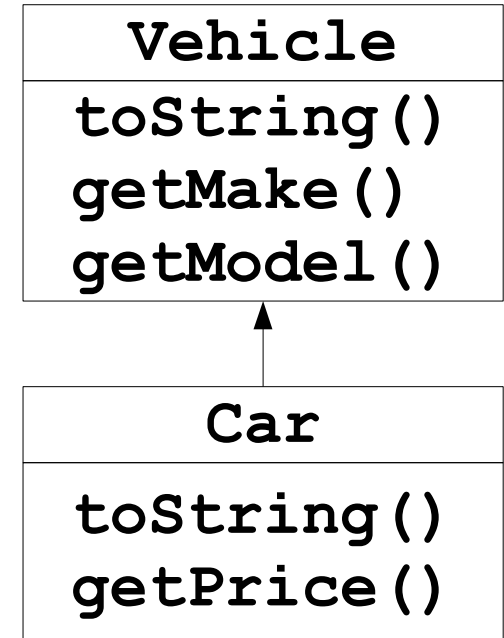
Inheritance in Java

```
class Subclass extends Superclass {  
    // <class body>  
}
```



Inheritance Example

```
public class Vehicle {
    private String make;
    private String model;
    public Vehicle() { make = ""; model = ""; }
    public String toString() {
        return "Make: " + make + " Model: " + model;
    }
    public String getMake() { return make; }
    public String getModel() { return model; }
}
// another file
public class Car extends Vehicle {
    private double price;
    public Car() {
        super(); // called implicitly can be left out
        price = 0.0;
    }
    public String toString() { // method overridden
        return "Make: " + getMake() + " Model: " + getModel()
            + " Price: " + price;
    }
    public double getPrice() { return price; }
}
```



Class Specialization and Class Extension

- The **Car** type with respect to extension and intension

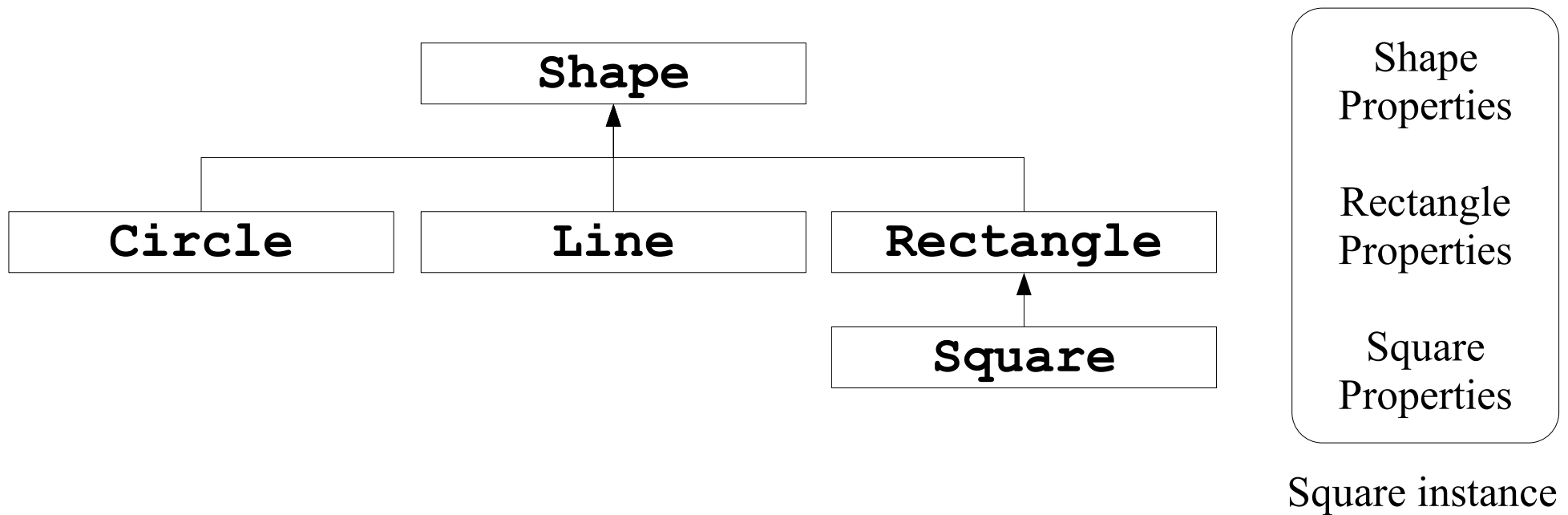
Class Extension

- **Car** is a class extension of **Vehicle**.
- The intension of **Car** is increased with the variable **price**.

Class Specialization

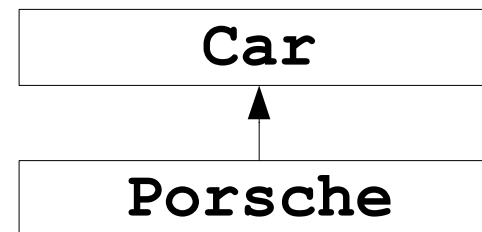
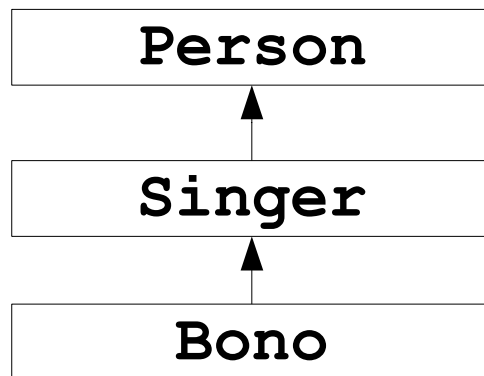
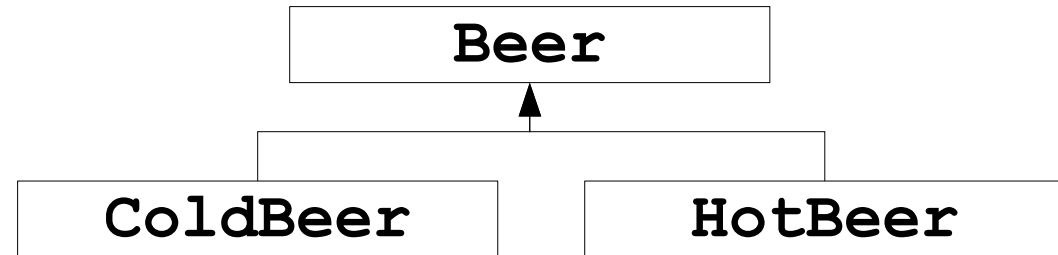
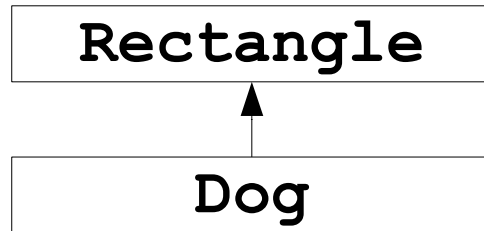
- **Car** is a class specialization of **Vehicle**.
- The extension of **Car** is decreased compared to the class **Vehicle**.

Instantiating and Initialization



- The **Square**, that inherits from **Rectangle**, that inherits from **Shape** is instantiated as a single object, with properties from the three classes **Square**, **Rectangle**, and **Shape**.

Inheritance Bad Examples



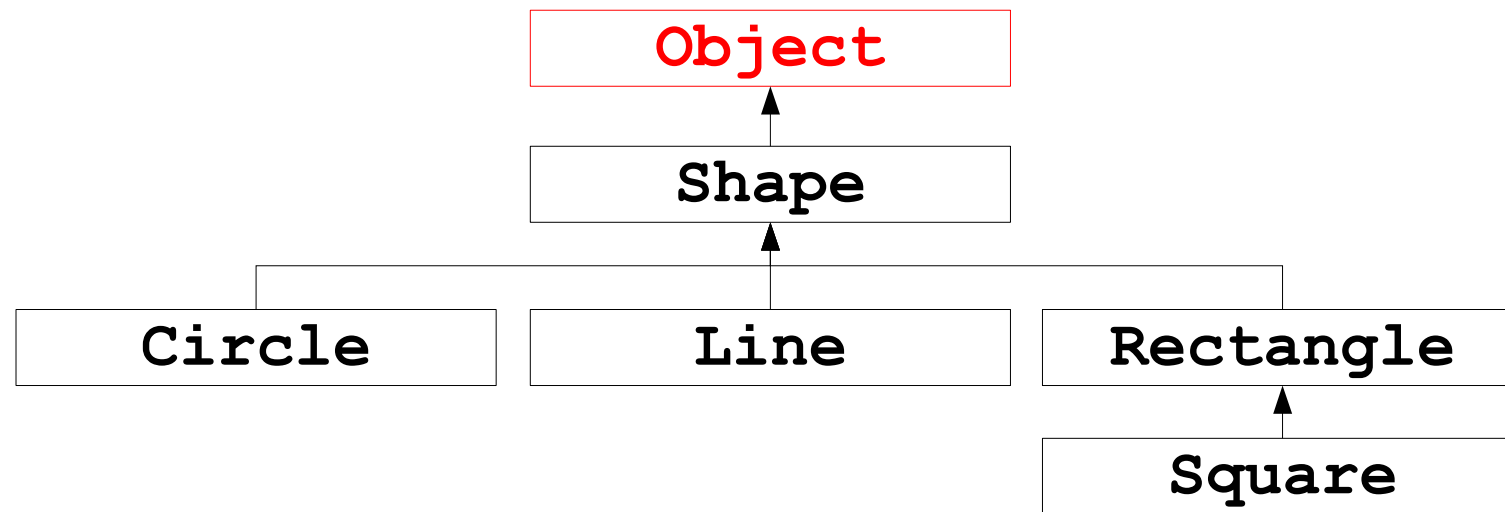
Inheritance and Constructors

- Constructors are not inherited.
- A constructor in a subclass must initialize variables in the class and variables in the superclass.
 - What about **private** fields in the superclass?
- It is possible to call the superclass' constructor in a subclass.
 - Default behavior: Superclass constructor called if exists

```
public class Vehicle{
    private String make, model;
    public Vehicle(String ma, String mo) {
        make = ma; model = mo;
    }
}
public class Car extends Vehicle{
    private double price;
    public Car() {
        // System.out.println("Start"); // not allowed
        super("", ""); // must be called
        price = 0.0;
    }
}
```


Class Hierarchies in Java

- Class **Object** is the root of the inheritance hierarchy in Java.
- If no superclass is specified a class inherits *implicitly* from **Object**.
- If a superclass is specified *explicitly* the subclass will inherit **Object**.



Order of Instantiation and Initialization

- The storage allocated for the object is initialized to binary zero before anything else happens.
- Static initialization is first done in the base class then the derived classes.
- The base-class constructor is called. (all the way up to **Object**).
- Member initializers are called in the order of declaration.
- The body of the derived-class constructor is called.

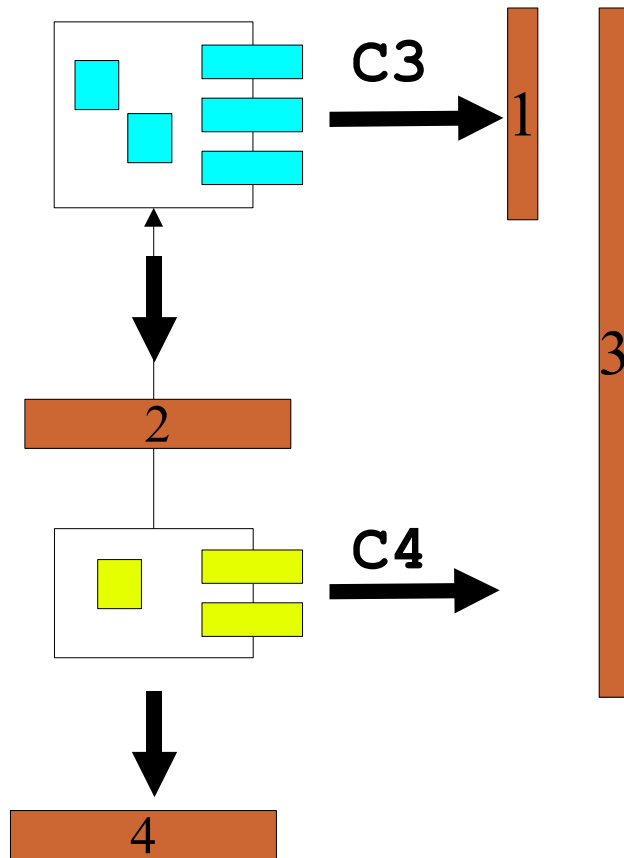
Inheritance and Constructors, cont.

```
class A {
    public A(){
        System.out.println("A()");
        // when called from B the B.doStuff() is called
        doStuff();
    }
    public void doStuff() { System.out.println("A.doStuff()"); }
}
class B extends A{
    int i = 7;
    public B(){System.out.println("B()");}
    public void doStuff() { System.out.println("B.doStuff() " + i); }
}


public class Base{
    public static void main(String[] args){
        B b = new B();
        b.doStuff();
    }
}

//prints
A()
B.doStuff() 0
B()
B.doStuff() 7
```

Interface to Subclasses and Clients



1. The properties of **C3** that clients can use.
2. The properties of **C3** that **C4** can use.
3. The properties of **C4** that clients can use.
4. The properties of **C4** that subclasses of **C4** can use.

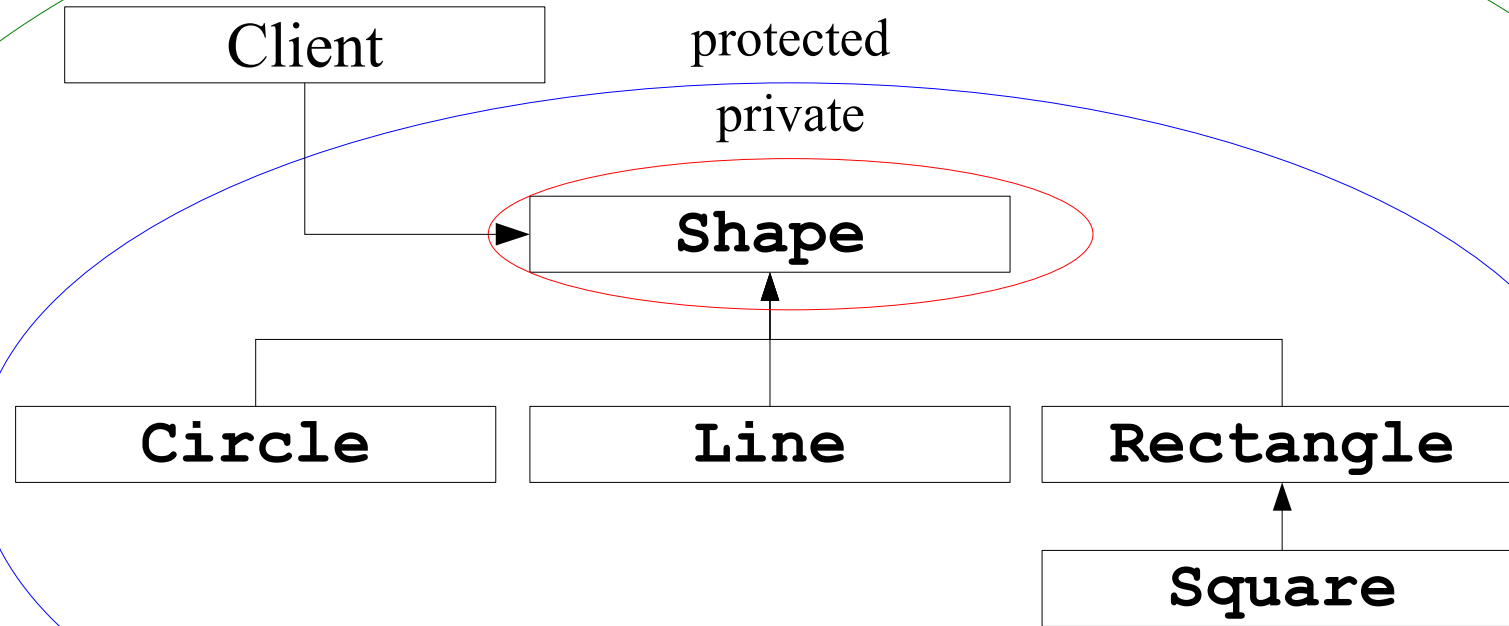
 = interface

protected, Revisited

- It must be possible for a subclass to access properties in a superclass.
 - **private** will not do, it is too restrictive
 - **public** will not do, it is too generous
- A *protected* variable or method in a class can be accessed by subclasses but not by clients.
- Which is more restrictive **protected** or package access?
- Change access modifiers when inheriting
 - Properties can be made “more public”.
 - Properties cannot be made “more private”.

protected, Revisited

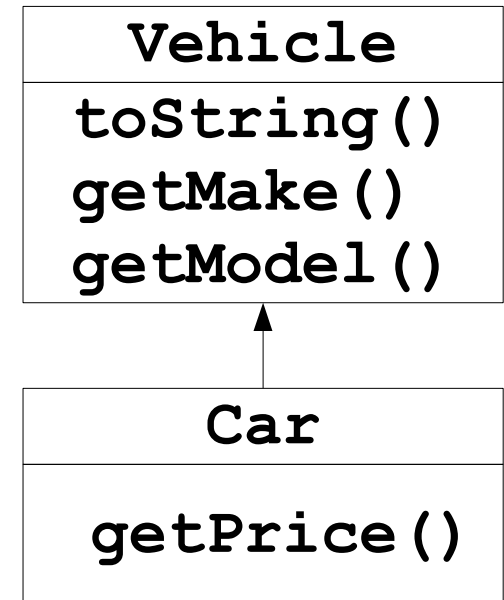
public



protected, Example

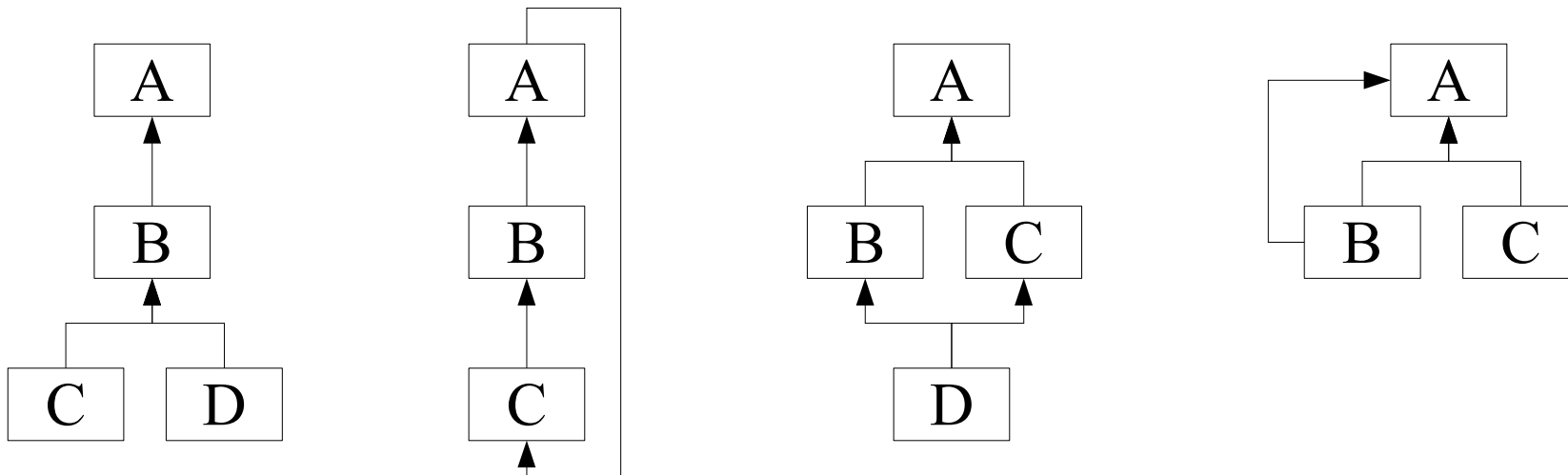
```
public class Vehicle1 {  
    protected String make;  
    protected String model;  
    public Vehicle1() { make = ""; model = "";}  
    public String toString() {  
        return "Make: " + make + " Model: " + model;  
    }  
    public String getMake() { return make;}  
    public String getModel() { return model;}  
}
```

```
public class Car1 extends Vehicle1 {  
    private double price;  
    public Car1() {  
        price = 0.0;  
    }  
    public String toString() {  
        return "Make: " + make + " Model: " + model  
            + " Price: " + price;  
    }  
    public double getPrice() { return price; }  
}
```



Class Hierarchies in General

- Class hierarchy: a set of classes related by inheritance.
- Possibilities with inheritance
 - Cycles in the inheritance hierarchy is not allowed.
 - Inheritance from multiple superclass may be allowed.
 - Inheritance from the same superclass more than once may be allowed.



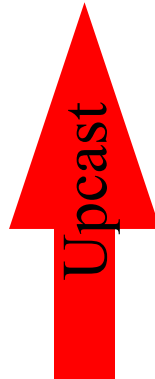
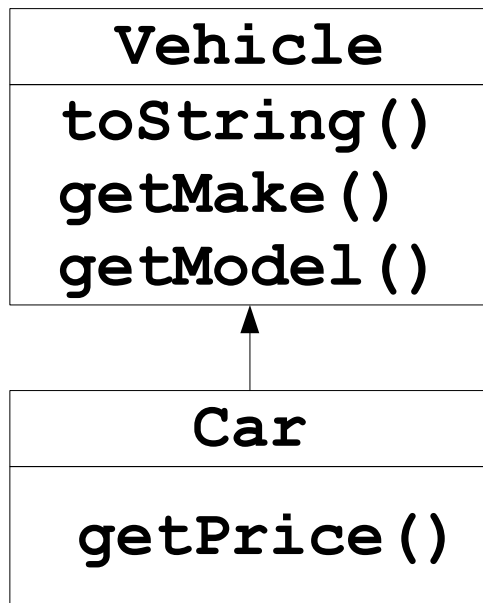
- “Multiple and repeated inheritance is a basic feature of Eiffel.”
[Meyer pp. 62].

Method/Variable Redefinition

- *Redefinition*: A method/variable in a subclass has the same as a method/variable in the superclass.
- Redefinition should change the *implementation* of a method, not its *semantics*.
- Redefinition in Java class B inherits from class A if
 - Method: Both versions of the method is available in instances of B. Can be accessed in B via **super**.
 - Variable: Both versions of the variable is available in instances of B. Can be accessed in B via **super**.
- “There are no language support in Java that checks that a method redefinition does not change the semantics of the method. In the programming language Eiffel assertions (pre- and post conditions) and invariants are inherited.” [Meyer pp. 228].

Upcasting

- Treat a subclass as its superclass



```
// example
Car c = new Car();
Vehicle v;
v = c;           // upcast

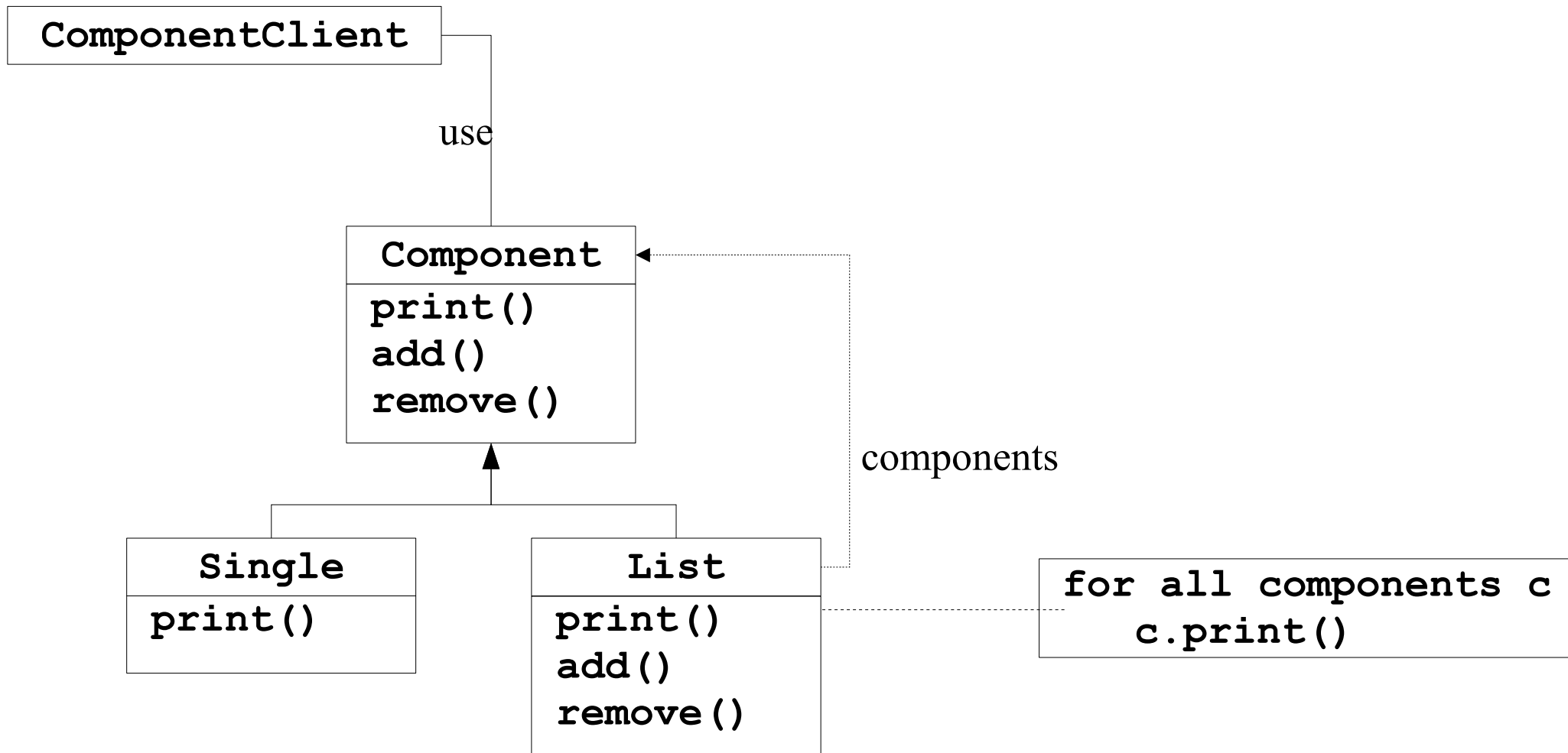
v.toString();   // okay
v.getMake();    // okay
//v.getPrice(); // not okay
```

- Central feature in object-oriented program
 - Covered in the next lecture
- Should be obvious that a method/field cannot be made more “private” in a subclass when redefining method/field.
 - However, it can be made more public.

The Ikea Component List Problem

- A part can be just the part itself (a brick).
- A part can consists of part that can consists of parts and so on.
As an example a garden house consists of the following parts
 - Garden house
 - ◆ walls
 - ◆ door
 - ▲ knob
 - ▲ window
 - frame
 - glass
 - ◆ window
 - ▲ frame
 - ▲ glass
 - ◆ floor
- Regardless whether it is a simple or composite part we just want to print the list.

Design of The Ikea Component List



- The *composite design pattern*
 - Used extensively when building Java GUIs (AWT/Swing)

Implementation of The Ikea Component List

```
public class Component{
    public void print(){
        System.out.println("Do not call print on me!");    }
    public void add(Component c){
        System.out.println("Do not call add on me!");}
}

public class Single extends Component{
    private String name;
    public Single(String n){ name = n; }
    public void print(){System.out.println(name);}
}

public class List extends Component{
    // uses parent class
    private Component[] comp; private int count;
    public List(){ comp = new Component[100]; count = 0; }
    public void print(){ for(int i = 0; i <= count - 1; i++){
        comp[i].print();
    }
}
    public void add(Component c){ comp[count++] = c;}
```

Implementation of The Ikea Component List

```
public class ComponentClient{ // Ikea
    public Component makeWindow(){ // helper function
        Component win = new List();
        win.add(new Single("frame")); win.add(new Single("glass"));
        return win;
    }
    public Component makeDoor(){ // helper function
        Component door = new List();
        door.add(new Single("knob")); door.add(makeWindow());
        return door;
    }
    public Component makeGardenHouse(){ // helper function
        Component h = new List();
        h.add(makeDoor()); h.add(makeWindow()); // etc
        return h;
    }
    public static void main(String[] args){
        ComponentClient c = new ComponentClient();
        Component brick = new Single("brick");
        Component myHouse = c.makeGardenHouse();
        brick.print();
        myHouse.print();
    }
}
```

Evaluation of The Ikea Component List

- Made **List** and **Single** classes look alike when printing from the client's point of view.
 - The main objective!
- Can make instances of **Component** class, not the intension
 - Can call dummy add/remove methods on these instances
- Can call add/remove method of **Single** objects, not the intension.
- Fixed length, not a great implementation
- Nice design!

The **final** Keyword

- Fields

- Compile-time constant (very useful)

```
final static double PI = 3.14
```

- Run-time constant (useful)

```
final int RAND = (int) Math.random * 10
```

- Arguments (not very useful)

```
double foo (final int i)
```

- Methods

- Prevents overridden in a subclass (use this very carefully)
- Private methods are *implicitly* final

- Final class (use this very carefully)

- Cannot inherit from the class

- Many details on the impacts of **final**.

Summary

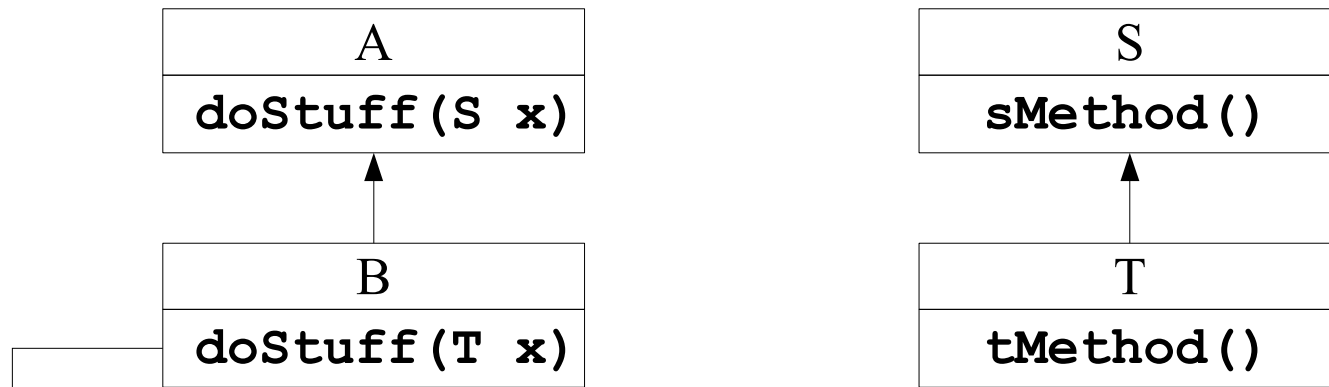
- Reuse
 - Use composition when ever possible more flexible and easier to understand than inheritance.
- Java supports specialization and extension via inheritance
 - Specialization and extension can be combined.
- A subclass automatically gets the fields and method from the superclass.
 - They can be redefined in the subclass
- Java supports single inheritance, all have **Object** as superclass
- Designing good reusable classes is (very) hard!
 - `while (!goodDesign()) { reiterateTheDesign(); }`

Method Combination

Different method combination

- It is programmatically controlled
 - Method doStuff on A controls the activation of doStuff on B
 - Method doStuff on B controls the activation of doStuff on A
 - *Imperative method combination*
- There is an overall framework in the run-time environment that controls the activation of doStuff on A and B.
 - doStuff on A should not activate doStuff on B, and vice versa
 - Declarative method combination
- Java support imperative method combination.

Changing Parameter and Return Types



```
class B extends A {  
    void doStuff (T x) {  
        x.tMethod();  
    }  
}
```

```
A a1 = new A();  
B b1 = new B();  
S s1 = new S();
```

```
a1 = b1;
```

```
a1.doStuff (s1); // can we use an S object here?
```

Covarians and Contravarians

- *Covarians*: The type of the parameters to a method varies in the same way as the classes on which the method is defined.
- *Contravarians*: The type of the parameters to a method varies in the opposite way as the classes on which the method is defined.