Inheritance and Polymorphism, Part 1

- Reuse
- Specialization and Extension
- Inheritance
- Polymorphism and Dynamic Binding
- Inheritance and Methods

Reuse

When you need a new class you can:

- 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 than composition reuse.
  - Today's main topic.

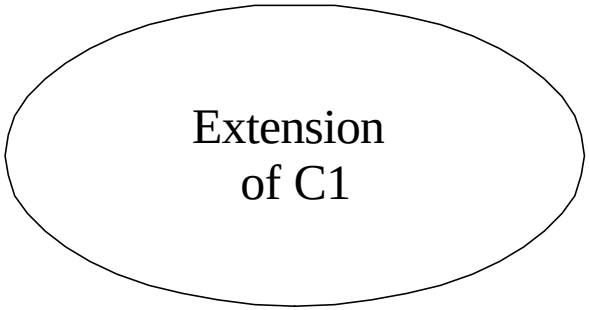
Class Specialization

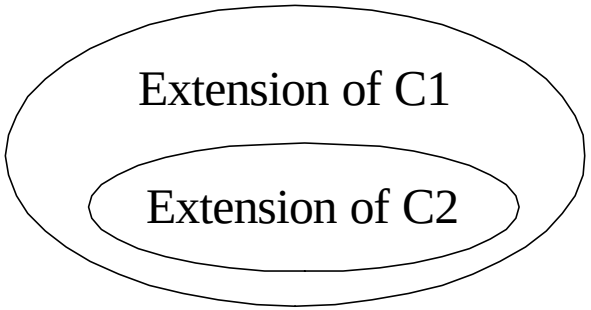
- In *Specialization* a class is considered an *Abstract Data Type* (ADT).
- The ADT is defined as a set of coherent values on which a set of operations is defined.

- 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

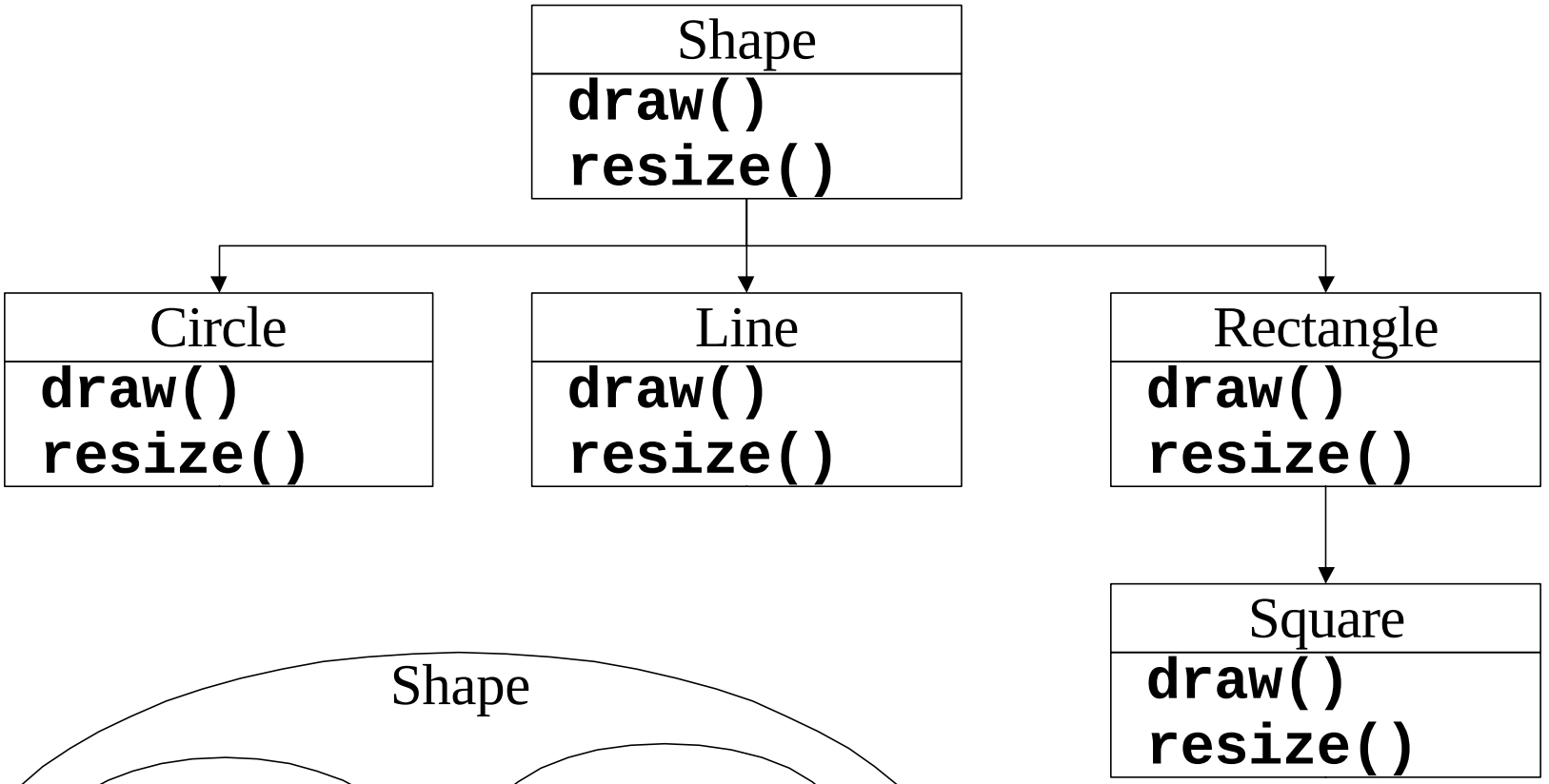
- The *extension* of a specialized class C2 is a subset of the extension of the general class C1.

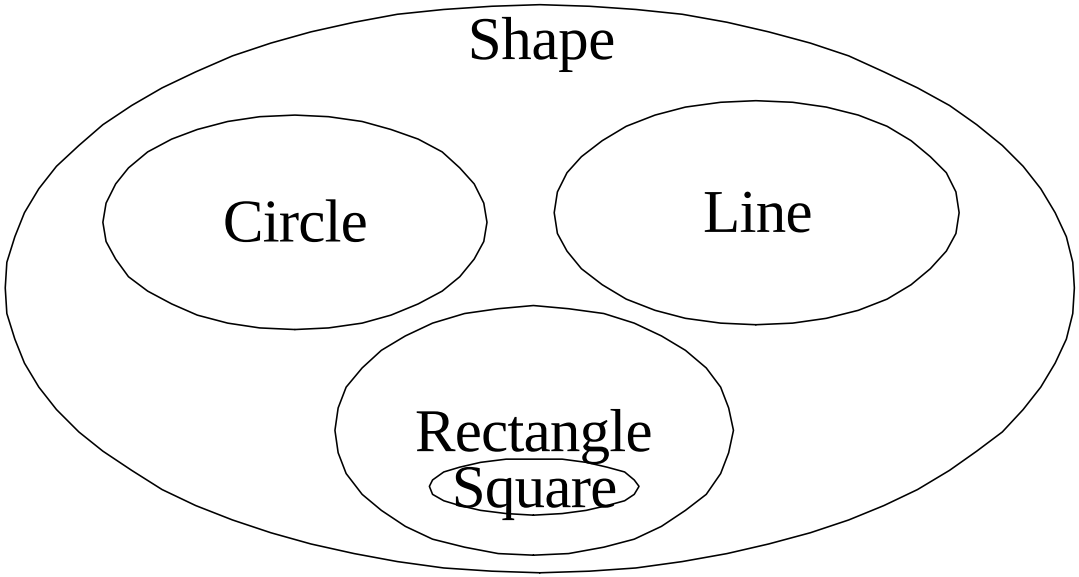




- "IS-A" Relationship
  - A C2 object is a C1 object (but not vice-versa).
  - There is an "is-a" relationship between C1 and C2.
  - We will later discuss a has-a relationship

Specialization, Example





Should the extension be overlapping?

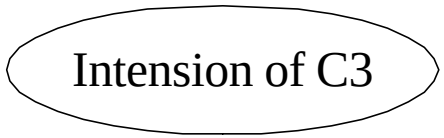
Class Extension

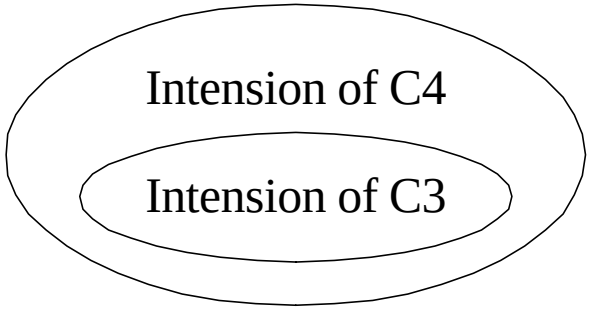
- In *extension* a class is considered a *module*.
- A module is a syntactical frame where a number of variables and method are defined, found in, e.g., Module-2 and PL/SQL.
- Extension is important in the context of *reuse*. Extension makes it possible for several modules to share code, i.e., avoid to have to copy code between modules.

- An 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.



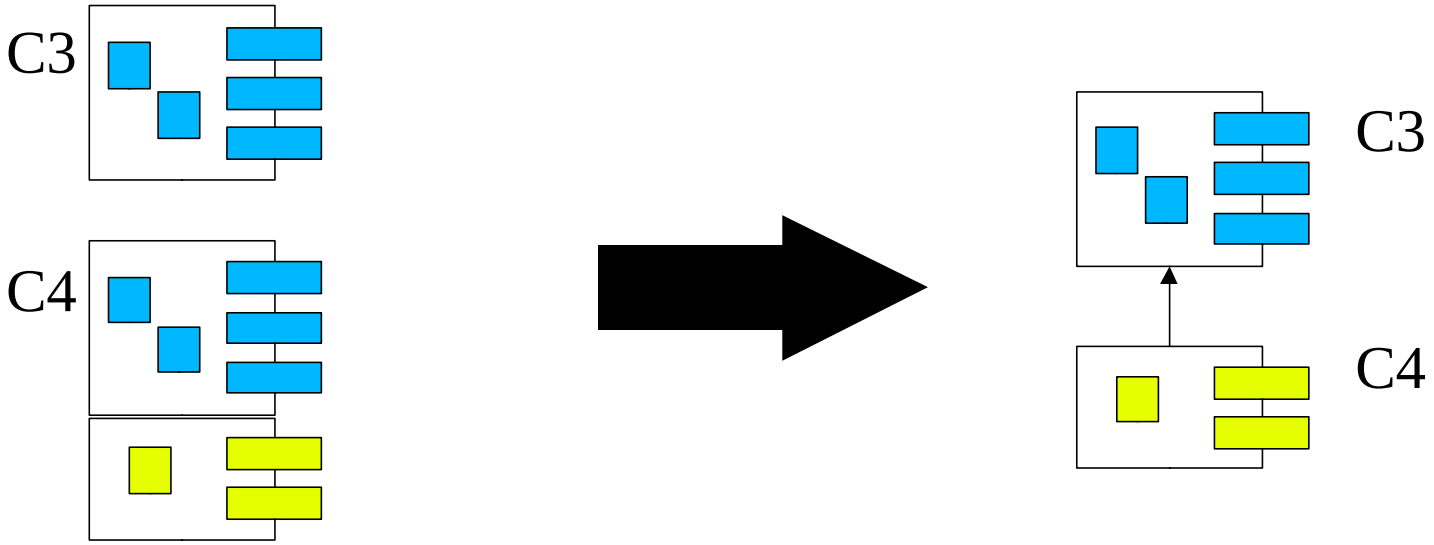


Inheritance

- Inheritance is a way to derive a new class from an existing class.

- Inheritance can be used for
  - Specializing an ADT.
  - Extend an existing class.
  - Often both specialization and extension takes place when a class inherits from an existing class.

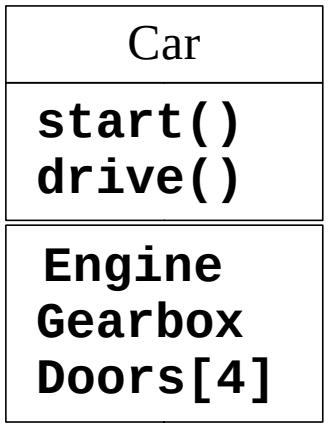
Inheritance and Class Extension



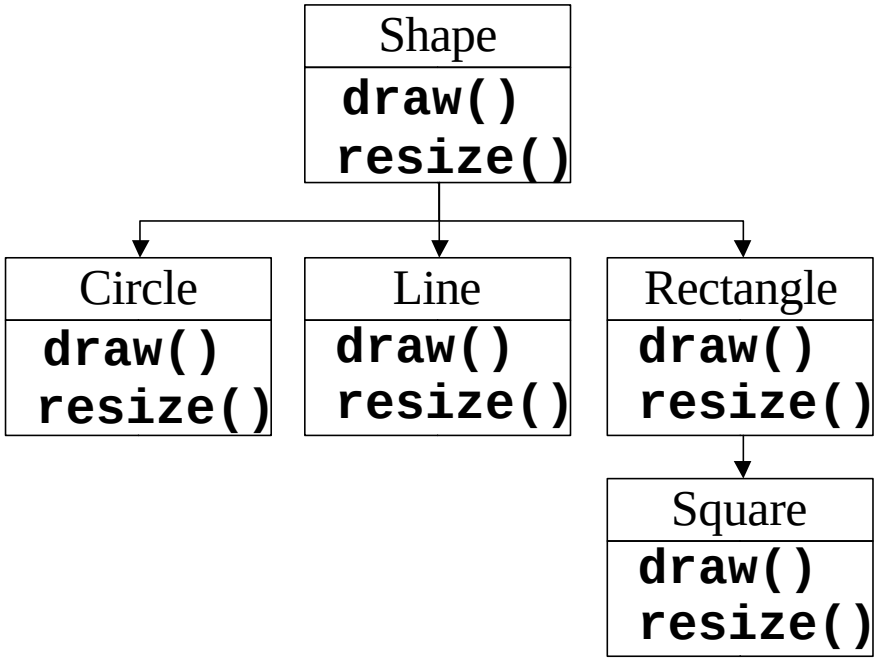
- Class C4 is created by *copying* C3.
- There are C3 and C4 instances.
- Instance of C4 have all C3 properties.
- C3 and C4 are totally separated.
- Maintenance of C3 properties must be done *two* places

- 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.

Composition vs. Inheritance

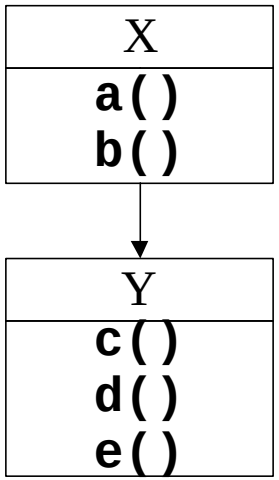


Pure Composition



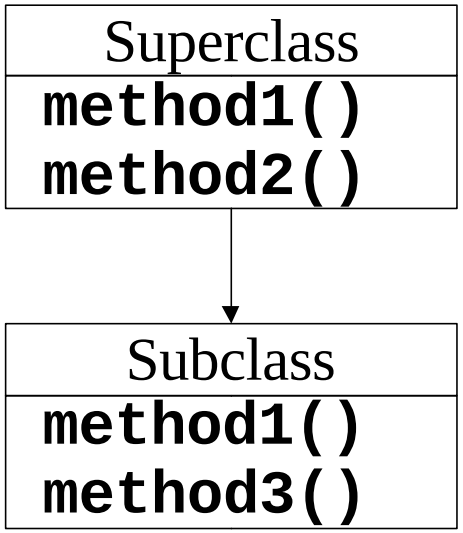
Pure Inheritance
(*substitution*)

Extension



Inheritance in Java

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



Inheritance Example

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

Vehicle Specialization and Class Extension

- The Car type with respect to extension and intension

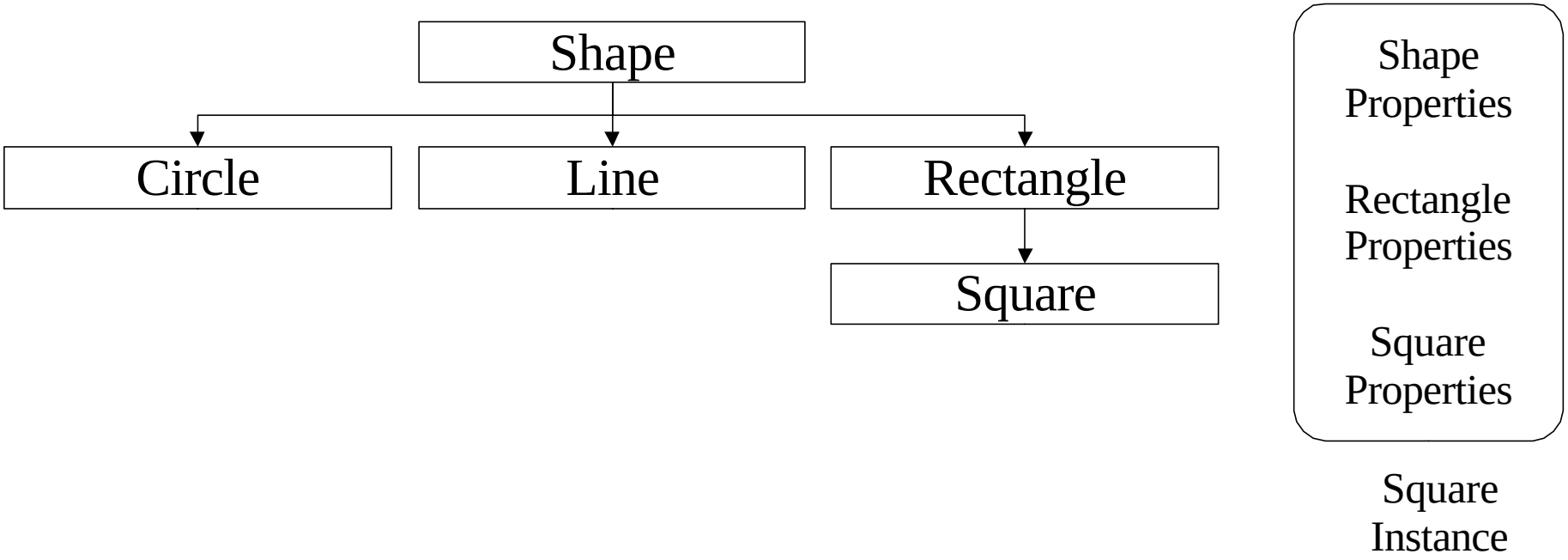
Extension

- **`Car`** is an extension of **`Vehicle`**.
- The intension of **`Car`** is increased with the variable **`price`**.

Specialization

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

Instatianting and Initialization



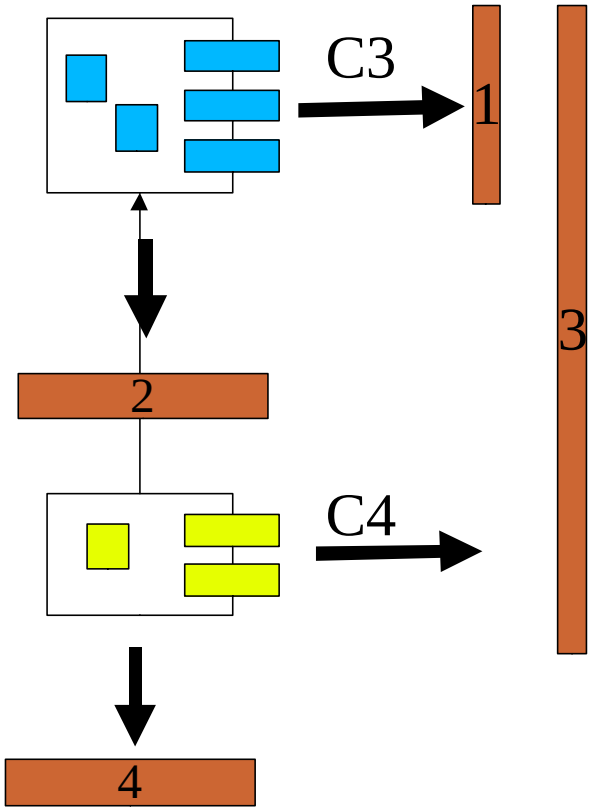
- The Square, that inherits from Rectangle, that inherits from Shape is instantiated as a single object, with properties from the three classes Square, Rectangle, and Shape.

Inheritance and Constructors

- Constructors are not inherited.
- A constructor in a subclass must initialize variables in the class and variables in the superclass.
- It is possible to call the superclass' constructor in a subclass.

```
public class Vehicle{
   protected String make, model;
   public Vehicle() {
      make = ""; model = "";
   }
}
public class Car extends Vehicle{
   private double price;
   public Car() {
      super();
      price = 0.0;
   }
}
```

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.

`protected`, Revisited

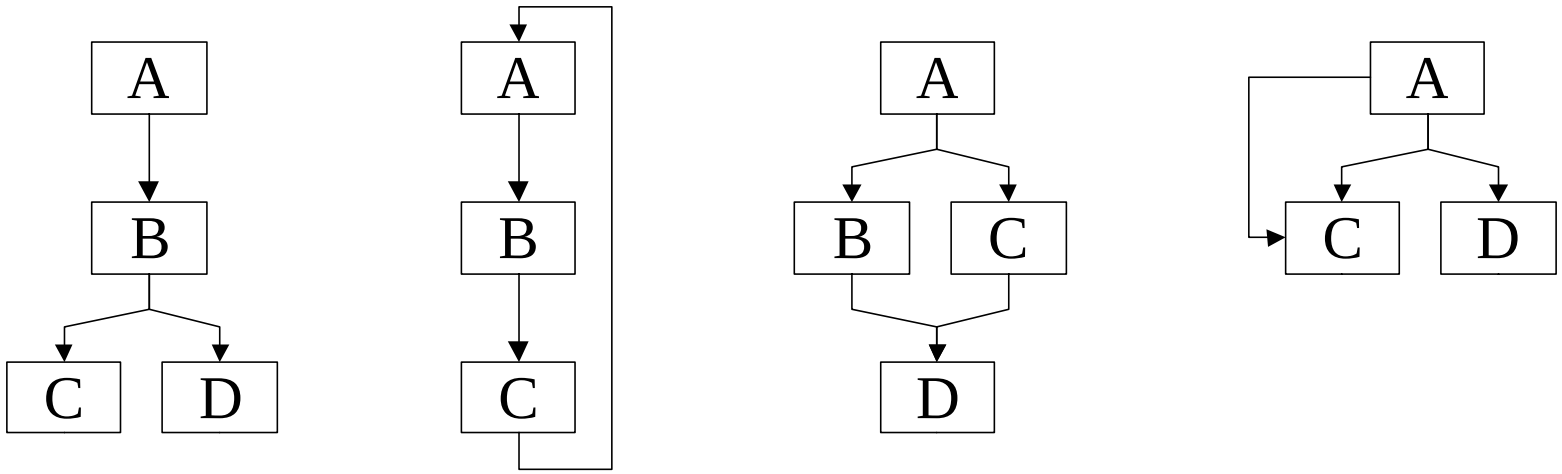
- It must be possible to for a subclass to access properties in a superclass.
  - **`private`** will not do, it is to restrictive
  - **`public`** will not do, it is to generous

- A *protected* variable or method in a class can be accessed by subclasses but not by clients.

- Change access modifiers when inheriting
  - Properties can be made "more public".
  - Properties cannot be made "more private".

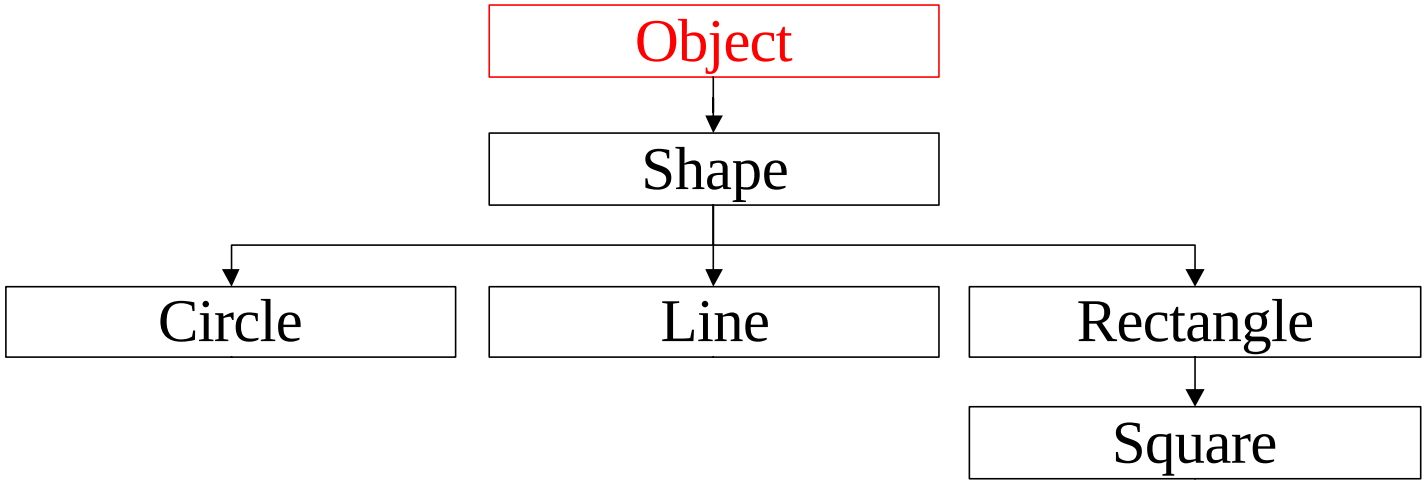
Class Hierarchies

- 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.



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`**.



Static and Dynamic Type

- The *static type* of a variable/parameter is the declaration type.
- The *dynamic type* of a variable/parameter is the type of the object the variable/parameter refers to.

```
class A {
  // body
}
class B extends A {
  // body
}
public static void main (String args[]) {
   A x;            // static type A
   B y;            // static type B

   x = new A();    // dynamic type A
   y = new B();    // dynamic type B
   x = y;          // dynamic type B
}
```

Polymorphism

- *Polymorphism:* The ability of a variable or argument to refer at run-time to instances of various classes.

```
Shape s = new Shape();
Circle c = new Circle();
Line l = new Line();
Rectangle r = new Rectangle();

s = l;
l = s; // is this legal?
```

- The assignment **`s = l`** is legal if the static type of **`l`** is **`Shape`** or a subclass of **`Shape`**.
- This is *static type checking* where the type comparison rules can be done at compile-time.

Why Polymorphism?

- Separate interface from implementation.
- Allows programmers to isolate type specific details from the main part of the code.
- Code is simpler to write and to read.
- Can change types (and add new types) with this propagates to existing code.

Dynamic Binding

- Dynamic binding is not possible without polymorphism.

```
class A {
   void doSomething(){
      ...
   }
}
```

```
class B extends A {
   void doSomething (){
      ...
   }
}
```

```
A x = new A();
B y = new B();
x = y;
y.doSomething(); // on class A or class B?
```

- *Dynamic binding*: The dynamic type of **x** determines which method is called
  - Dynamic binding is not possible without polymorphism.
- *Static binding*: The static type of **x** determines which method is called.

Dynamic Binding, Example

```
class Shape {
   draw() {}
}
class Circle extends Shape {
   draw() {System.out.println ("Circle");
}
class Line extends Shape {
   draw() {System.out.println ("Line");
}
class Rectangle extends Shape {
   draw() {System.out.println ("Rectangle");
}

public static void main (String args[]){
   Shape[] s = new Shape[3];
   s[0] = new Circle();
   s[1] = new Line();
   s[2] = new Rectangle();
   for (int i = 0; i < s.length; i++){
      s[i].draw(); // prints Circle, Line, Rectangle
   }
}
```

Method 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`**.

The `final` Keyword

- Final fields
  - Compile time constant

```
final static int PI = 3.14
```

  - Run-time constant

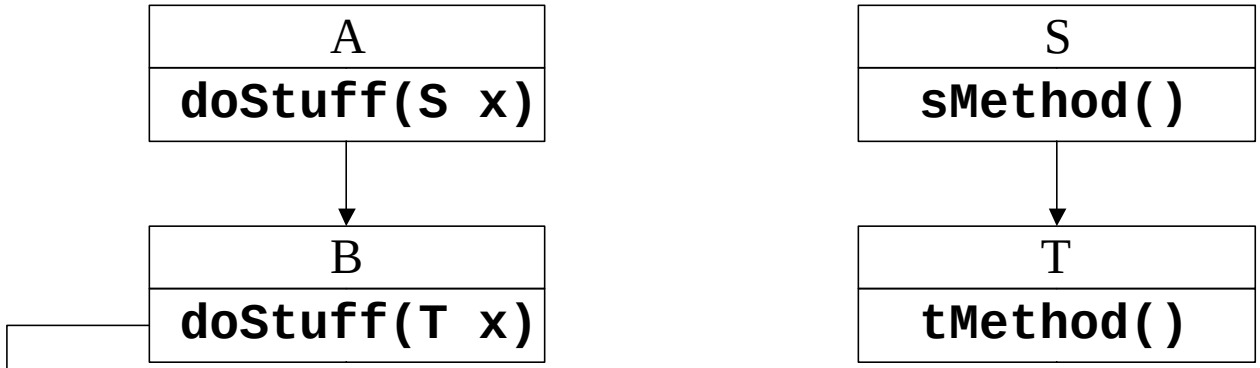
```
final int RAND = (int) Math.random * 10
```

- Final arguments

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

- Final methods
  - Prevents overwriting in a subclass
  - Private methods are implicitly final
- Final class
  - Cannot inherit from the class

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

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.

Summary

- Designing good reusable classes is hard!
- Reuse
  - Use composition when ever possible more flexible and easier to understand.
- Java supports specialization and extension via inheritance
  - Specialization and extension can be combined.
- Polymorphism is a pre request for dynamic binding an central to the object-oriented programming paradigm.