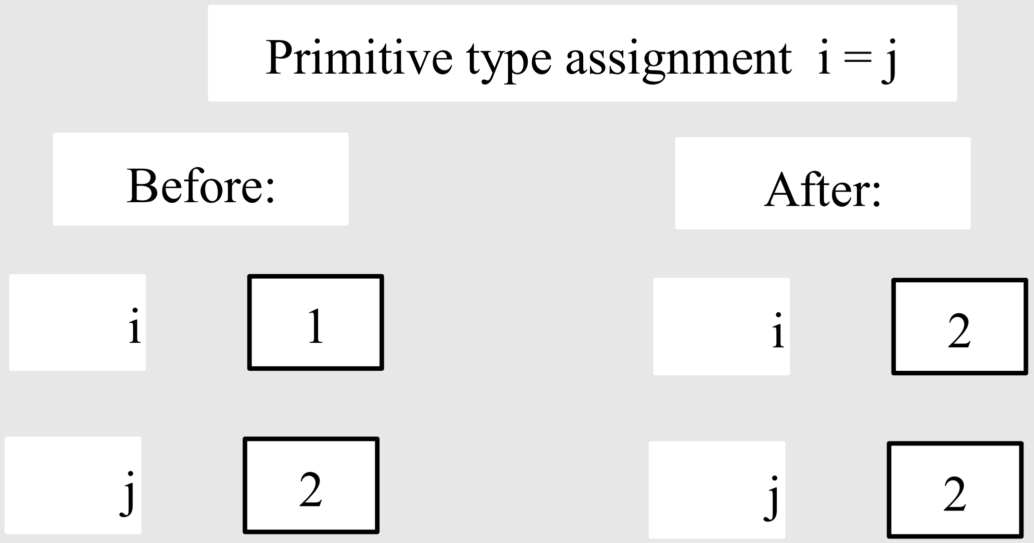
COIS2240 Lecture 3

Primitive Type Vs Object Type (Reference Type)



Primitive type assignment i = j

Before:

i 1

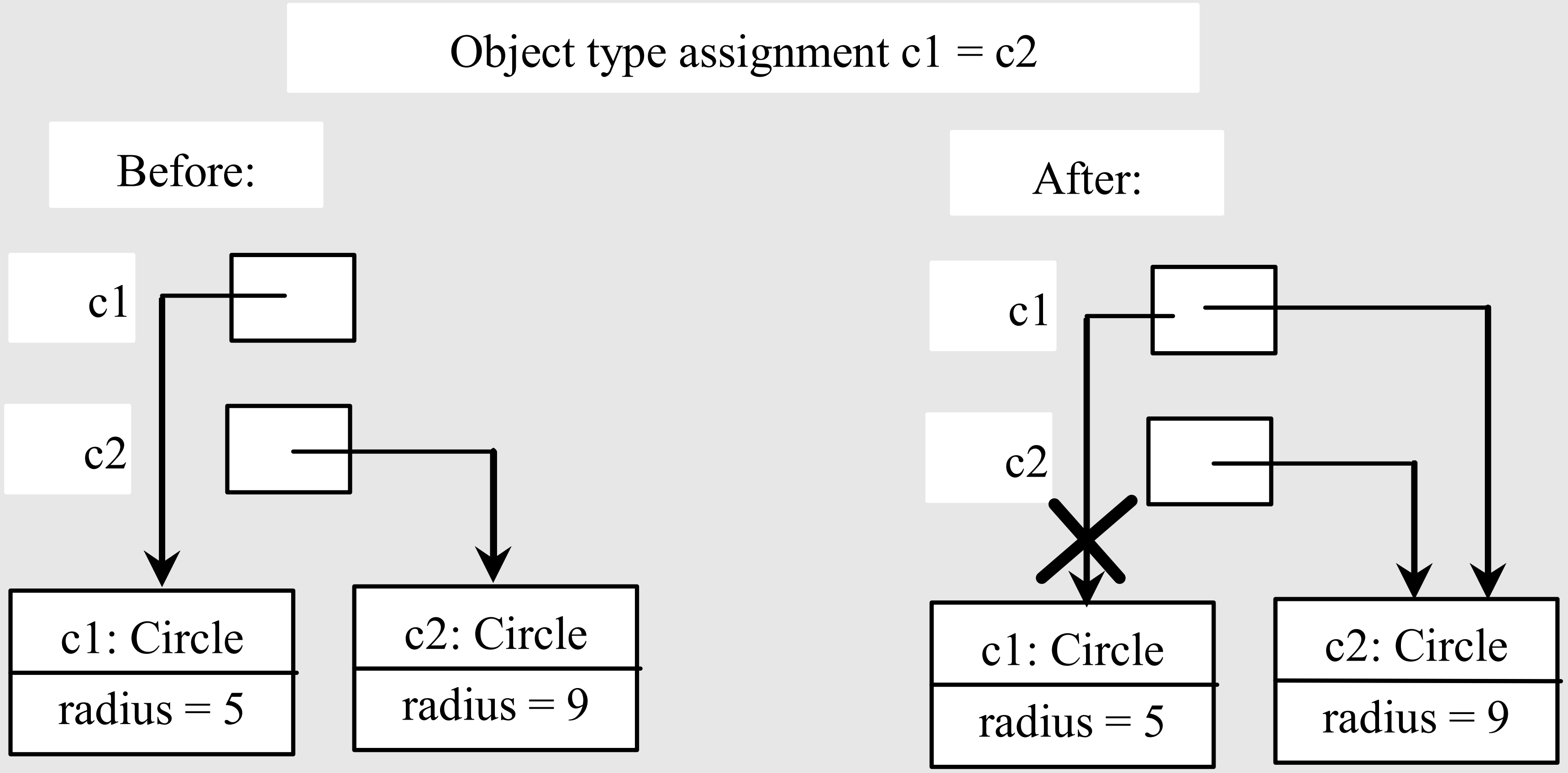
j 2

After:

i 2

j 2

What happens to *c1: Circle*?
Garbage Collector..



Object type assignment c1 = c2

Before:

c1

c2

c1: Circle
radius = 5

c2: Circle
radius = 9

After:

c1

c2

c1: Circle
radius = 5

c2: Circle
radius = 9

Arrays and Collections

Arrays are of fixed size and lack methods to manipulate them

`ArrayList` is the most widely used class to hold a *collection* of other objects

- More powerful than arrays, but less efficient

`Iterator`s are used to access members of `Vector`s

- Enumerations were formally used, but were more complex

```
a = new ArrayList();
Iterator i = a.iterator();
while(i.hasNext())
{
   aMethod(i.next());
}
```

Casting

Java is very strict about types

- If variable v is declared to have type X, you can only invoke operations on v that are defined in X or its superclasses
    - Even though an instance of a *subclass* of X may be actually stored in the variable
- If you *know* an instance of a subclass is stored, then you can *cast* the variable to the subclass
    - E.g. if I know a `Vector` contains instances of `String`, I can get the next element of its `Iterator` using:

      ```
              (String)i.next();
      ```
    - To avoid casting you could also have used templates:

      ```
      a = ArrayList<String>; i=a.iterator(); i.next()
      ```

Exceptions

Anything that can go wrong should result in the raising of an Exception

- Exception is a class with many subclasses for specific things that can go wrong

Use a try - catch block to trap an exception

```
try
{
   // some code
}
catch (ArithmeticException e)
{
  // code to handle division by zero
}
```

Packages and importing

A package combines related classes into subsystems

- All the classes in a particular directory

Classes in different packages can have the same name

- Although not recommended

Importing a package is done as follows:

```
import finance.banking.accounts.*;
```

Access control

Applies to methods and variables

- `public`
  - Any class can access
- `protected`
  - Only code in the package, or subclasses can access
- (blank)
  - Only code in the package can access
- `private`
  - Only code written in the class can access
  - Inheritance still occurs!

Implicit Import and Explicit Import

```
java.util.* ; // Implicit import
```

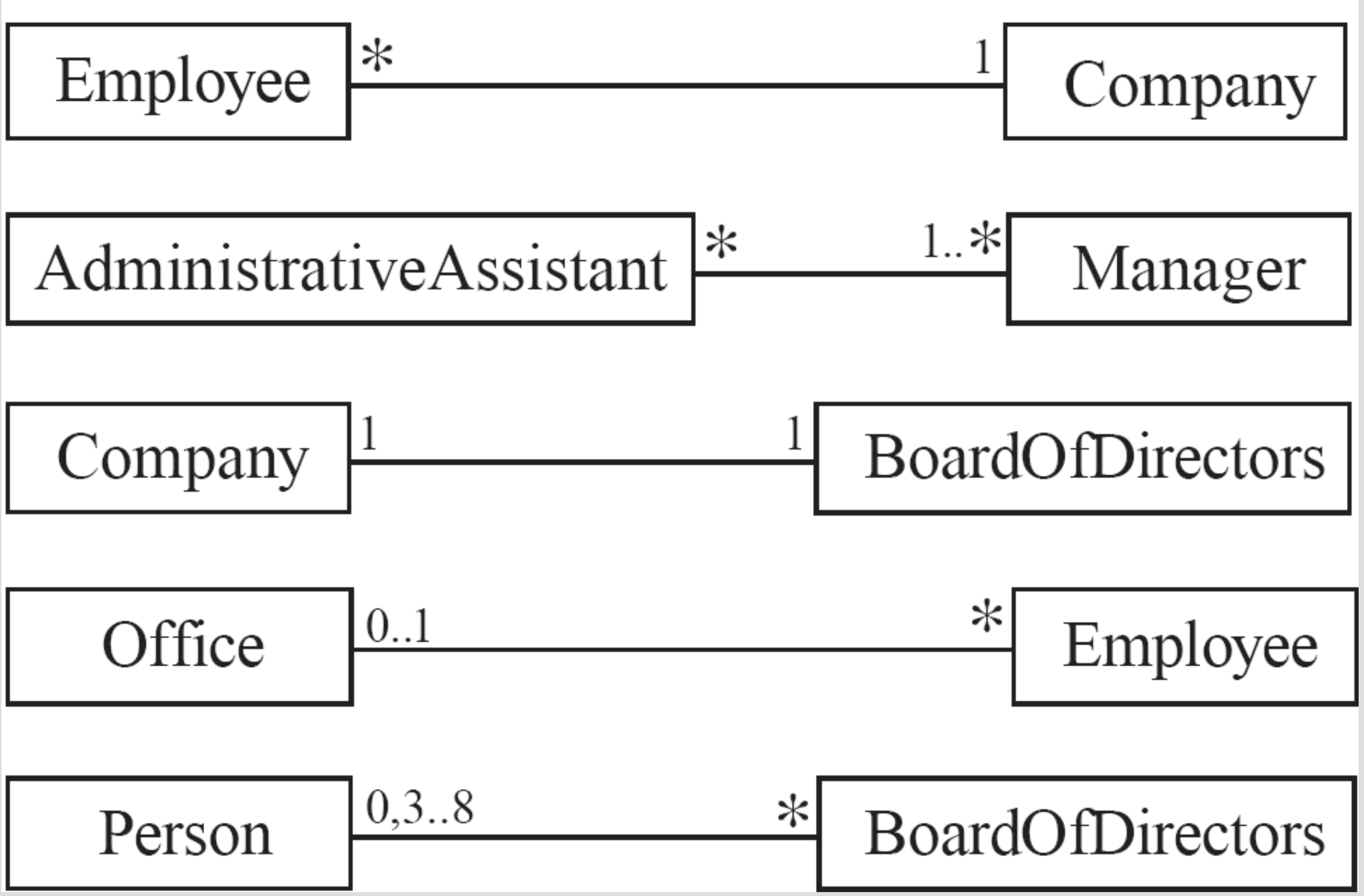
```
java.util.JOptionPane; // Explicit Import
```

No performance difference

Associations and Multiplicity

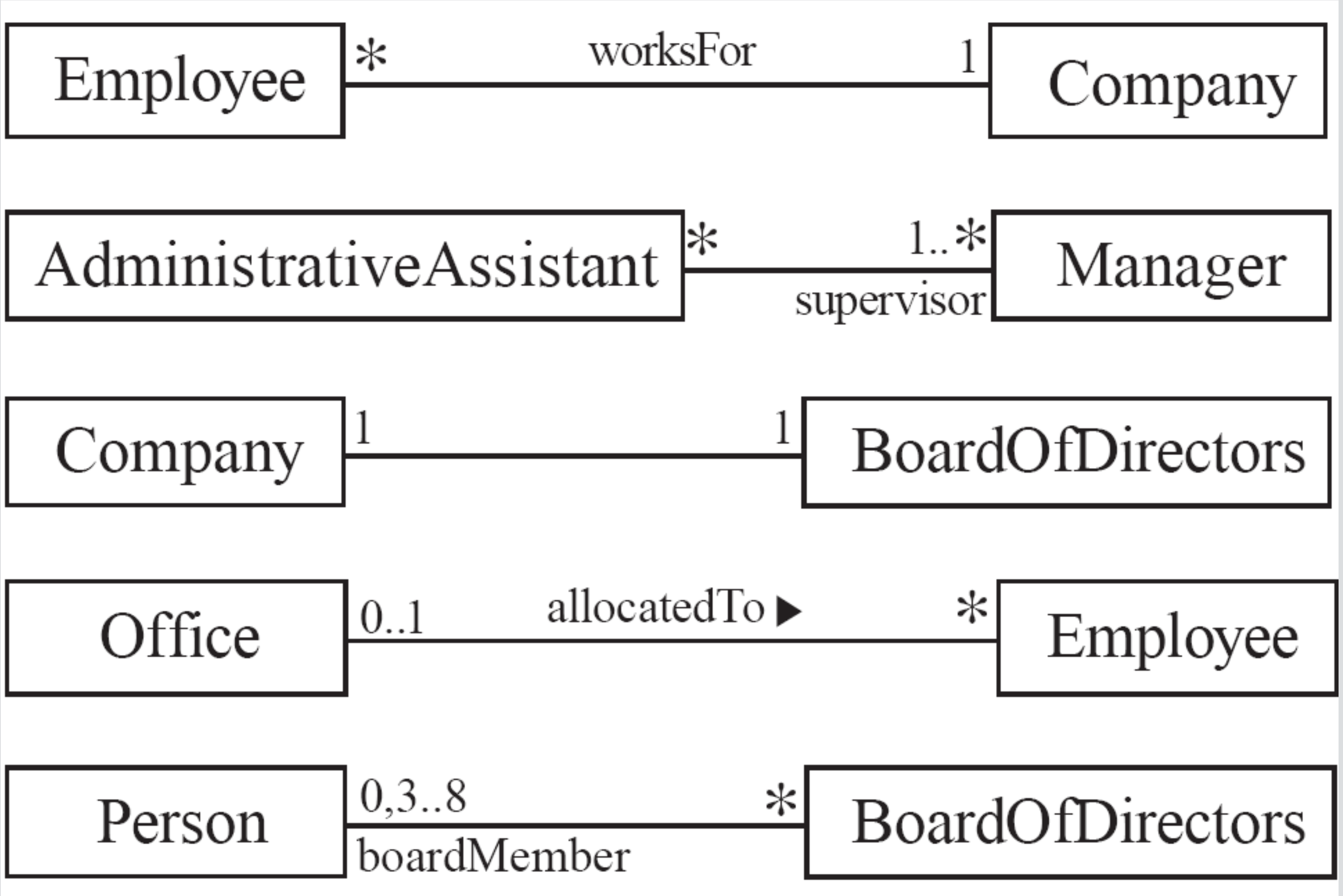
An *association* is used to show how two classes are related to each other

- Symbols indicating *multiplicity* are shown at each end of the association



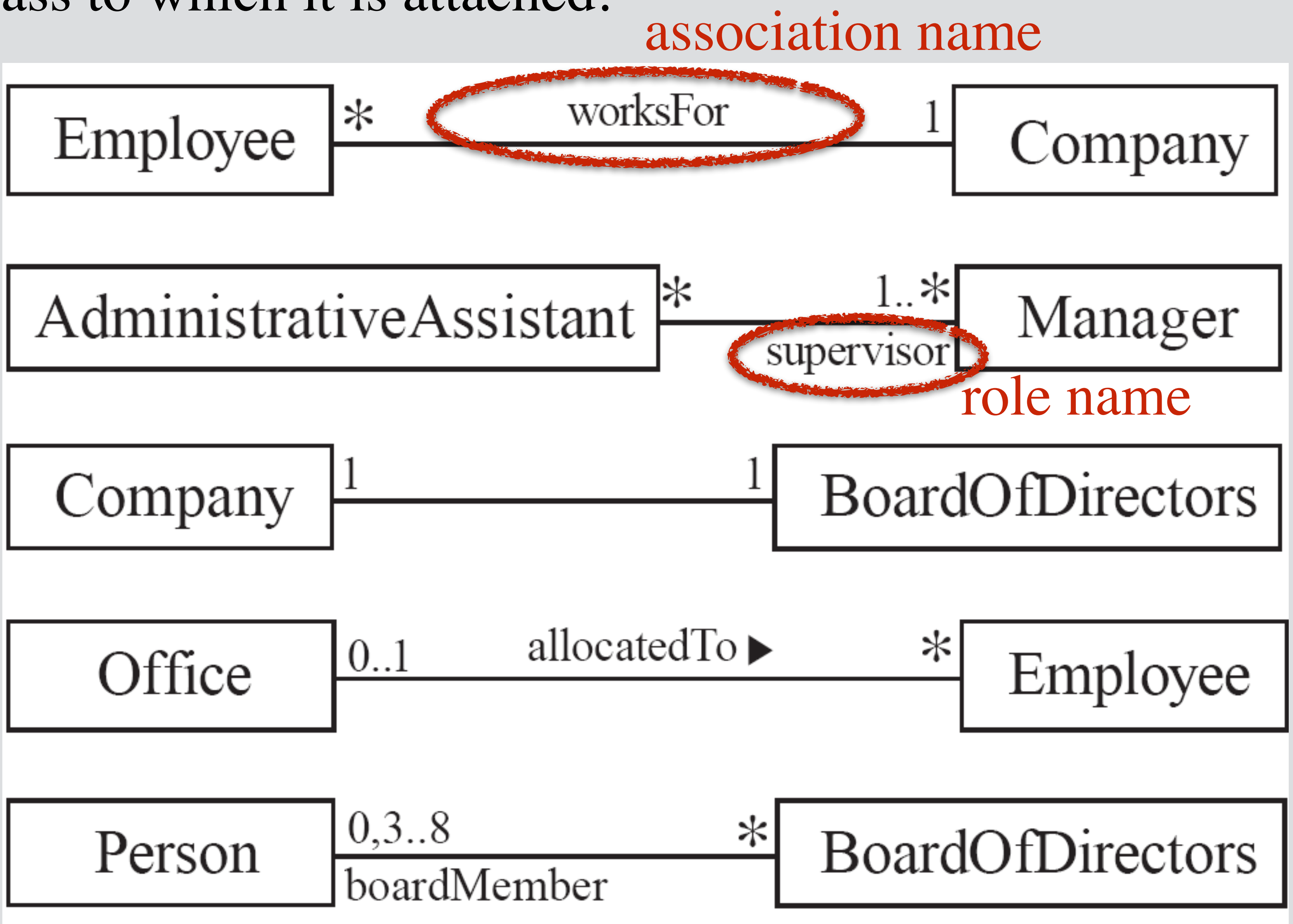
Labelling associations

- Each association can be labelled, to make explicit the nature of the association.
- A role name acts, in the context of the association, as an alternative name for the class to which it is attached.



Labelling associations

- Each association can be labelled, to make explicit the nature of the association.
- A role name acts, in the context of the association, as an alternative name for the class to which it is attached.



In Java ..

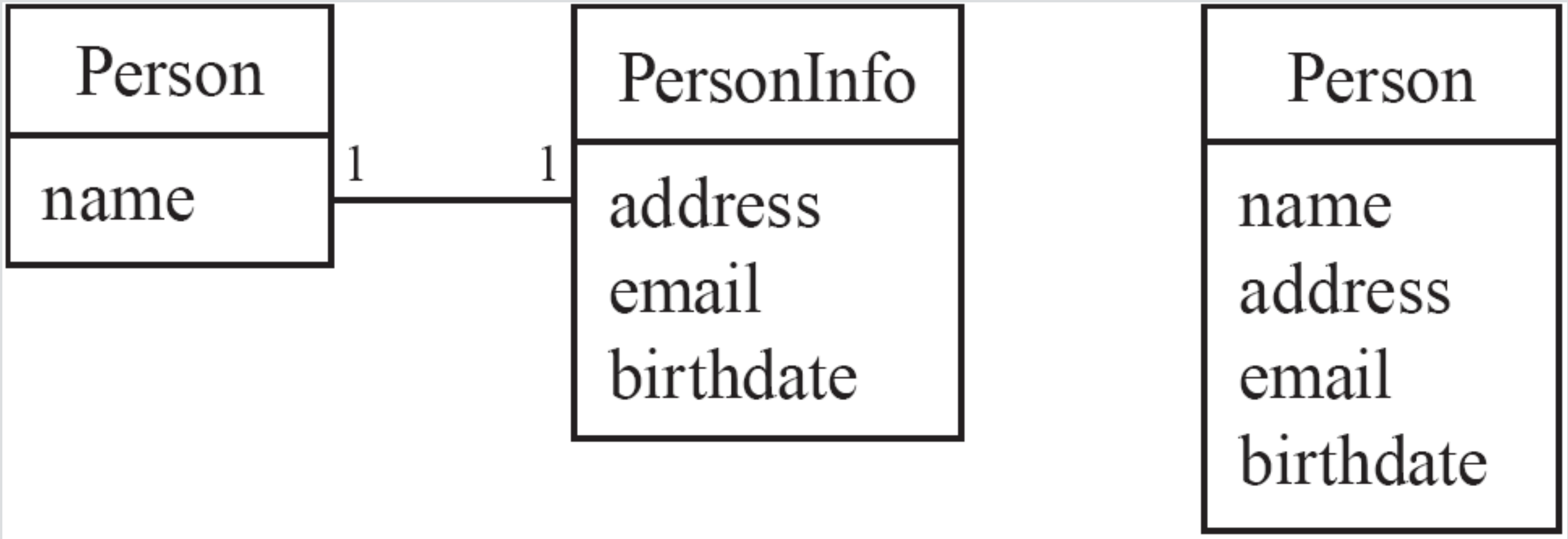
```
Class Manager
{
  …..
}



Class AdministrativeAssistant
{
  private Manager supervisor [5];
}
```

Analyzing and validating associations

One-to-one associations are less common.

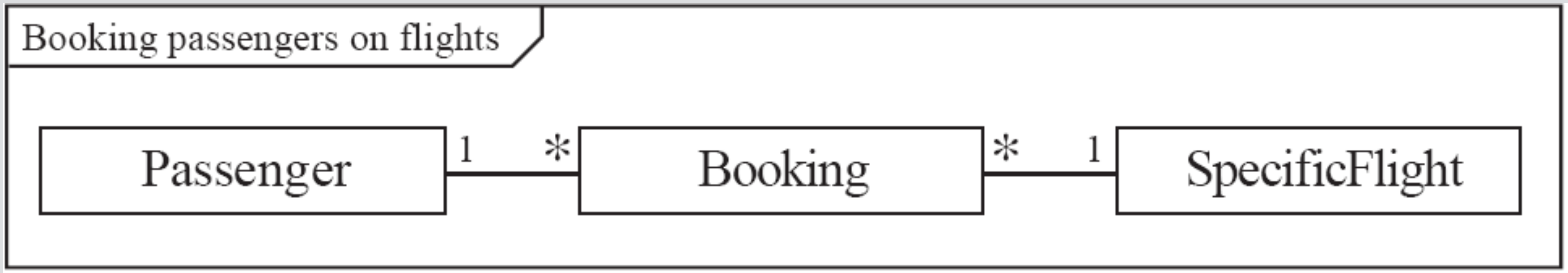
Avoid unnecessary one-to-one associations

Avoid this **do this**



A more complex example

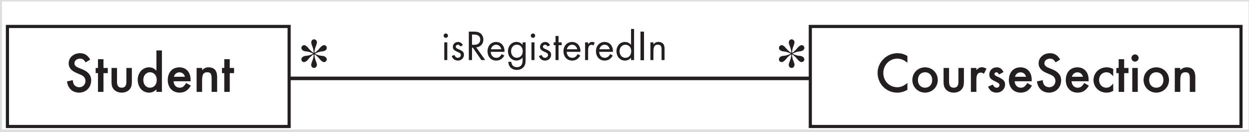
- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



- The *frame* around this diagram is an optional feature that any UML 2.0 may possess.

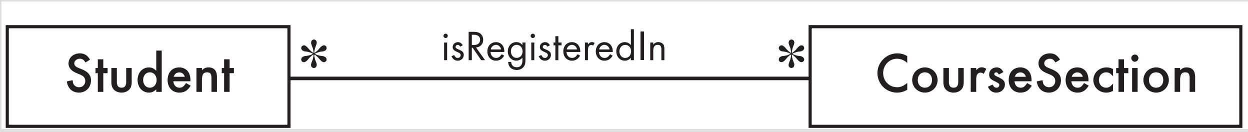
Association classes

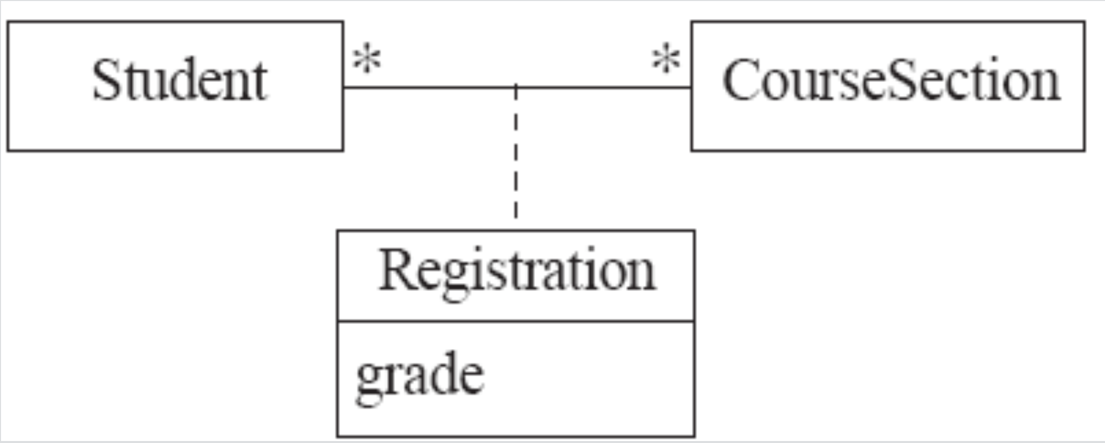
- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent

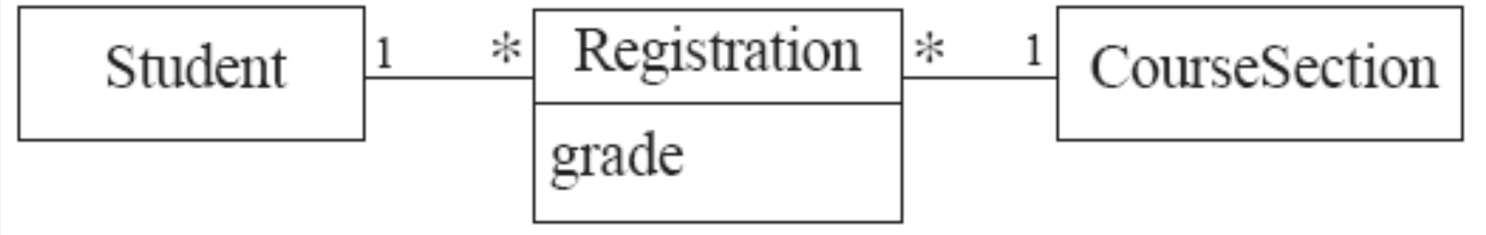


Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent

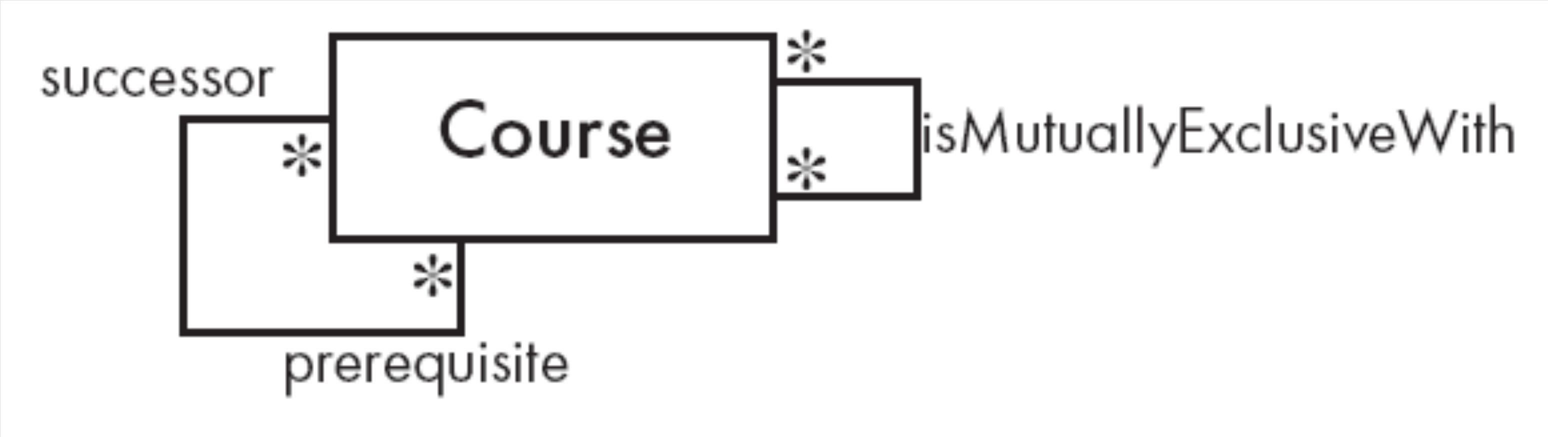






Reflexive associations

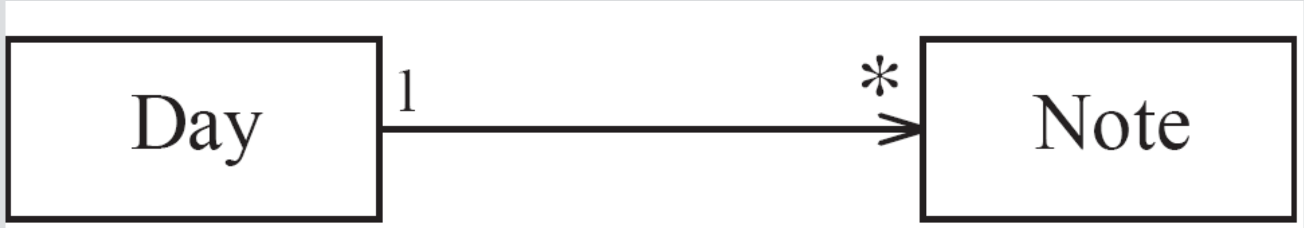
- It is possible for an association to connect a class to itself



Open in Umple

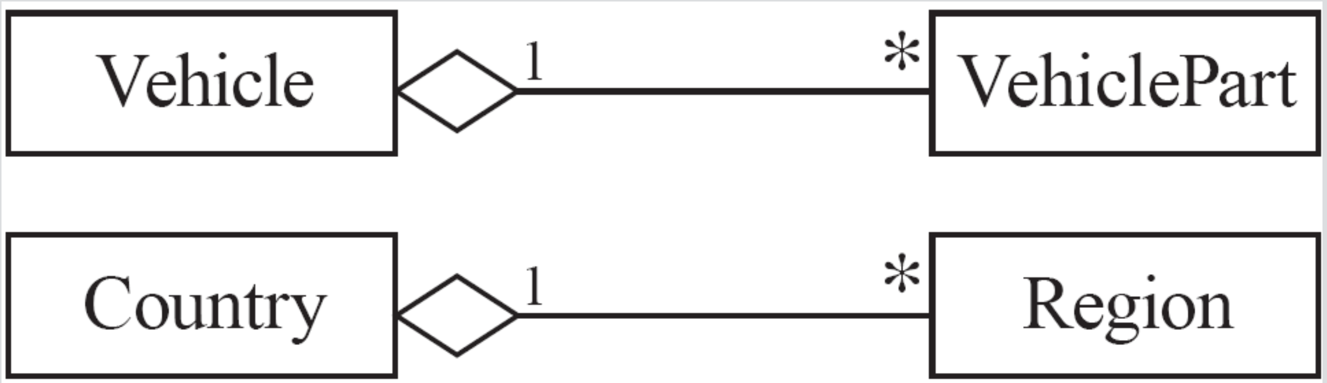
Directionality in associations

- **Associations are by default *bi-directional***
- **It is possible to limit the direction of an association by adding an arrow at one end**



Aggregation

- Aggregations are special associations that represent ‘part-whole’ relationships.
  - The ‘whole’ side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`



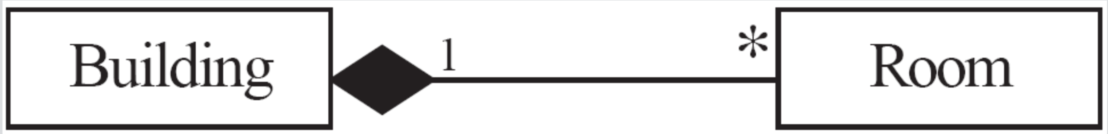
When to use an aggregation

As a general rule, you can mark an association as an aggregation if the following are true:

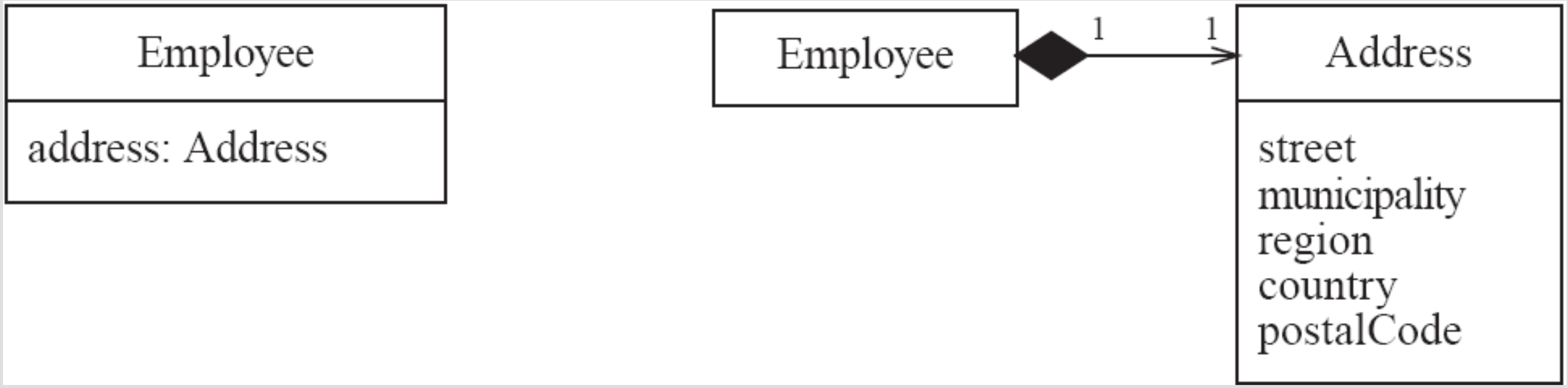
- You can state that
  - the parts ‘are part of’ the aggregate
  - or the aggregate ‘is composed of’ the parts
- When something owns or controls the aggregate, then they also own or control the parts

Composition

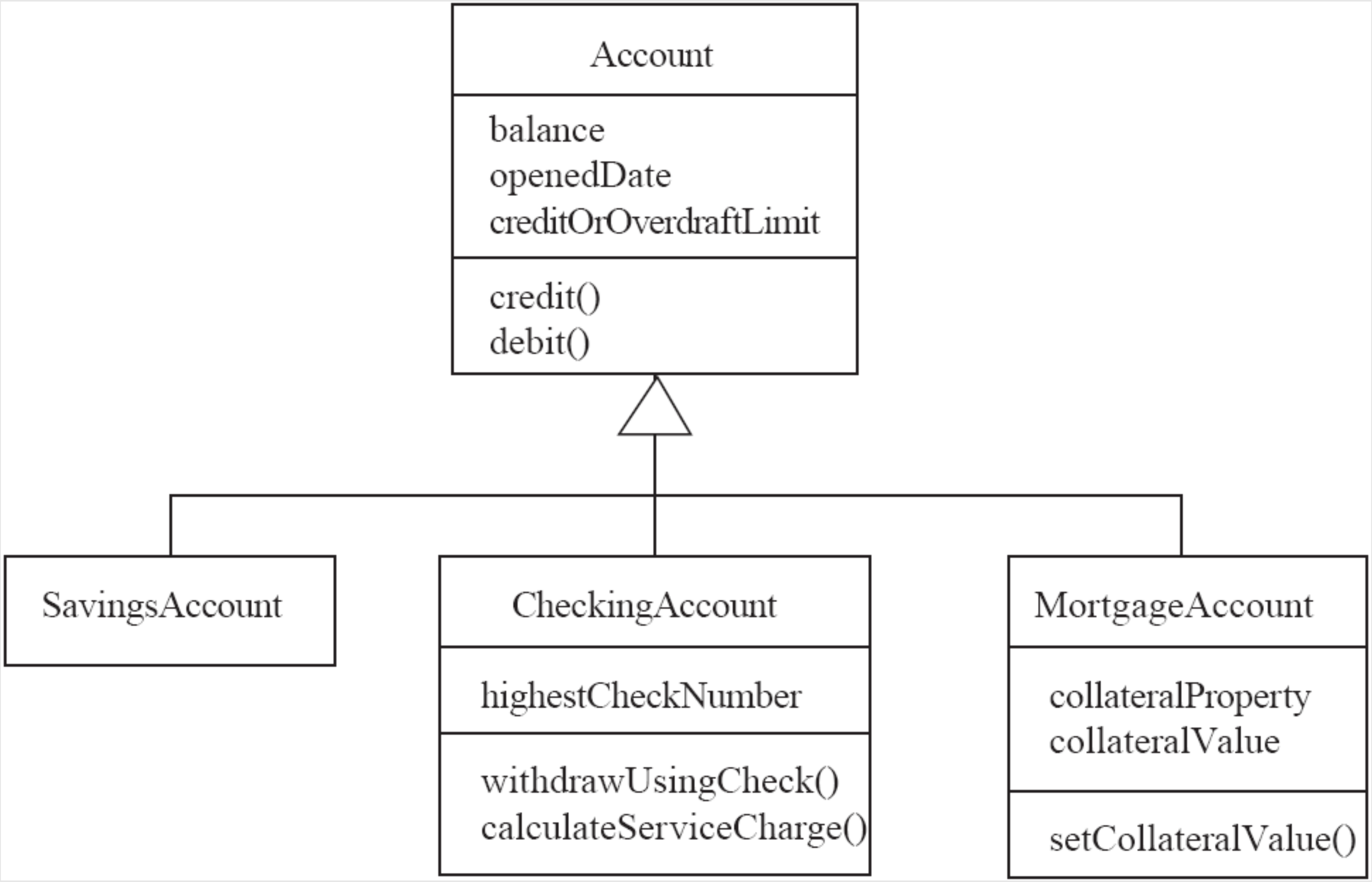
- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well



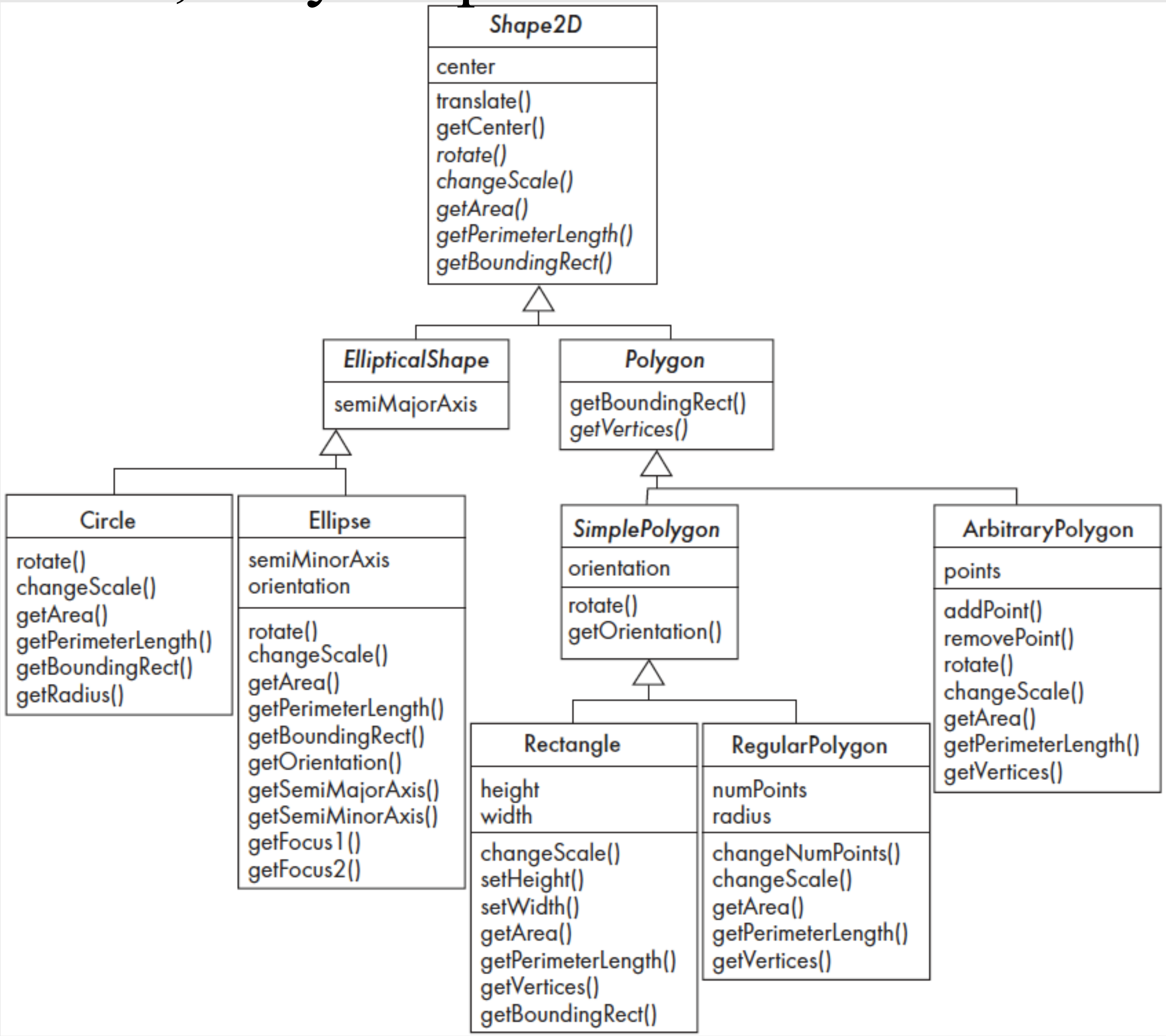
- Two alternatives for addresses



Make Sure all Inherited Features Make Sense in Subclasses



Inheritance, Polymorphism and Variables



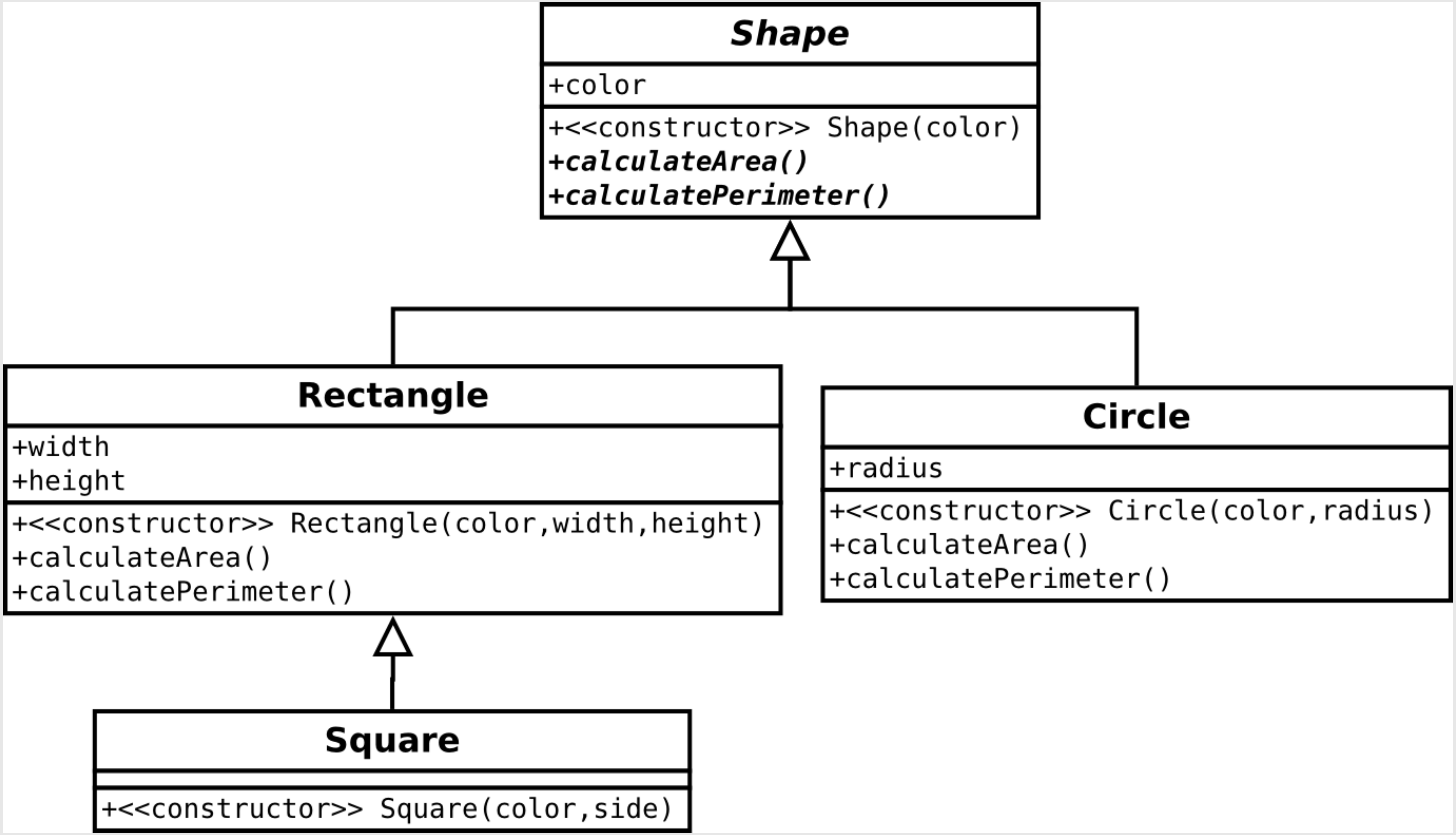
Inheritance, Abstract Classes and Methods

-To justify the existence of a subclass, there must be a different attribute in the subclass or an operation that is done differently in the subclass.

-An operation should be declared to exist at the highest class in the hierarchy where it makes sense.

- The *operation* may be *abstract* (lacking implementation) at that level
- If so, the *class* also must be *abstract*
  - No instances can be created
  - The opposite of an abstract class is a *concrete* class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - Leaf classes must have or inherit concrete methods for all operations
  - Leaf classes must be concrete

Inheritance, Abstract Classes and Methods



Shape

+color

+<<constructor>> Shape(color)
+calculateArea()
+calculatePerimeter()

Rectangle

+width
+height

+<<constructor>> Rectangle(color,width,height)
+calculateArea()
+calculatePerimeter()

Circle

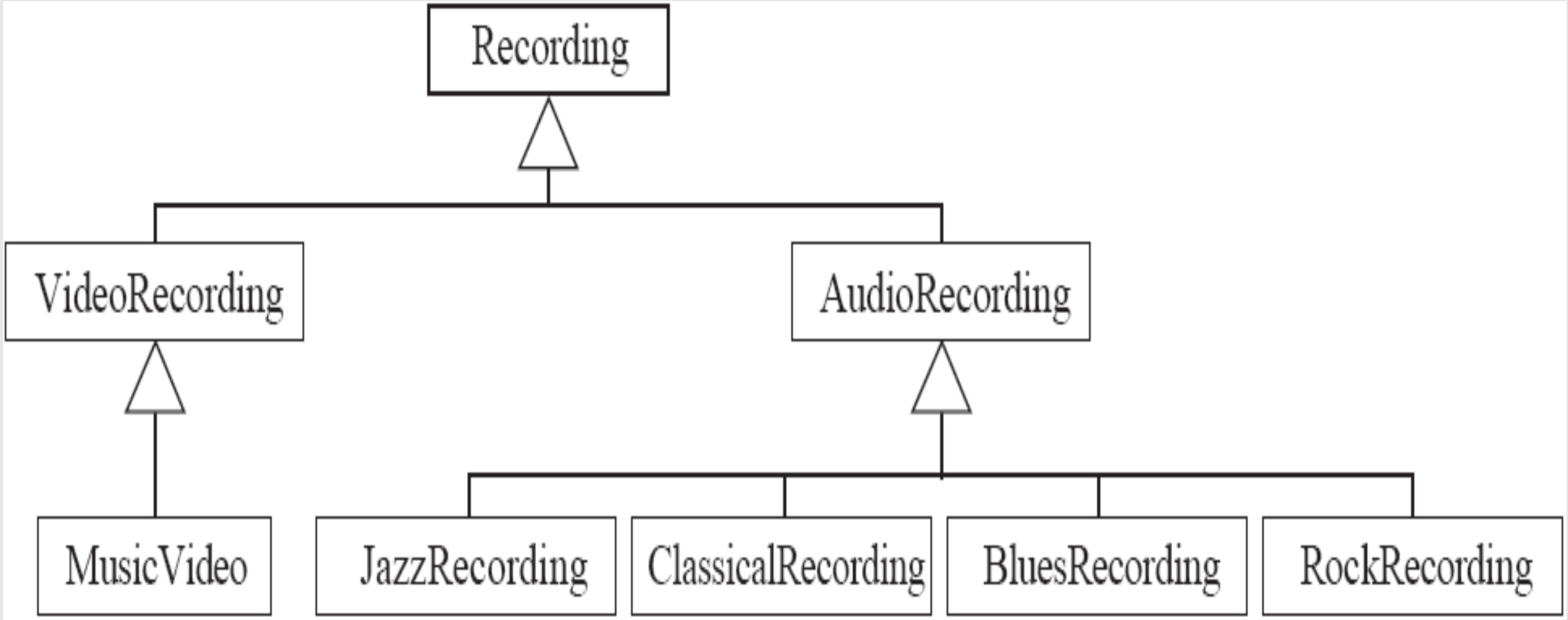
+radius

+<<constructor>> Circle(color,radius)
+calculateArea()
+calculatePerimeter()

Square

+<<constructor>> Square(color,side)

Avoid Unnecessary Generalizations

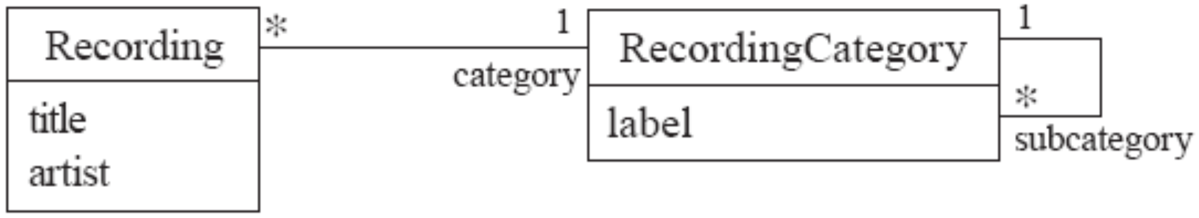


A hierarchy of classes in which there would not be any differences in operations. This should be avoided

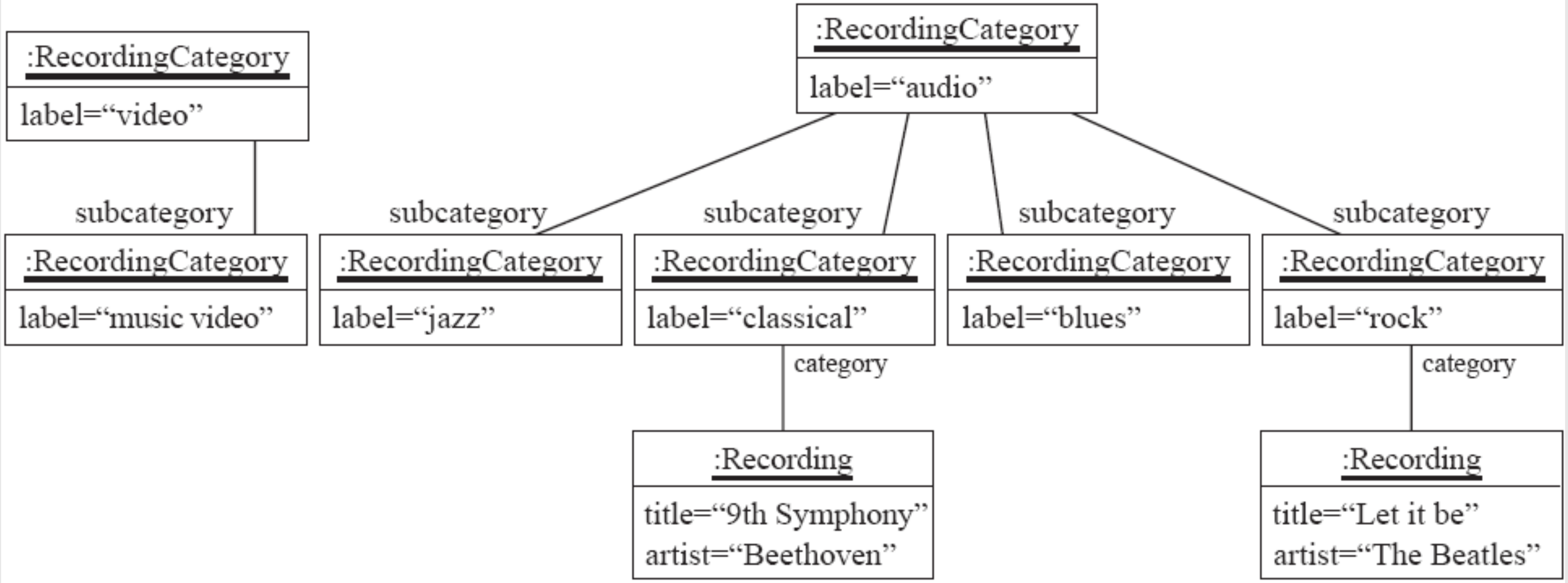
Inappropriate hierarchy of classes, which should be instances

The next slide shows a way to solve this…

To Solve Overuse of Generalization



(a)



(b)

Interfaces

Like abstract classes, but cannot have executable statements

- Define a set of abstract operations that make sense in several classes

A class can implement any number of interfaces

- It must have concrete methods for the operations

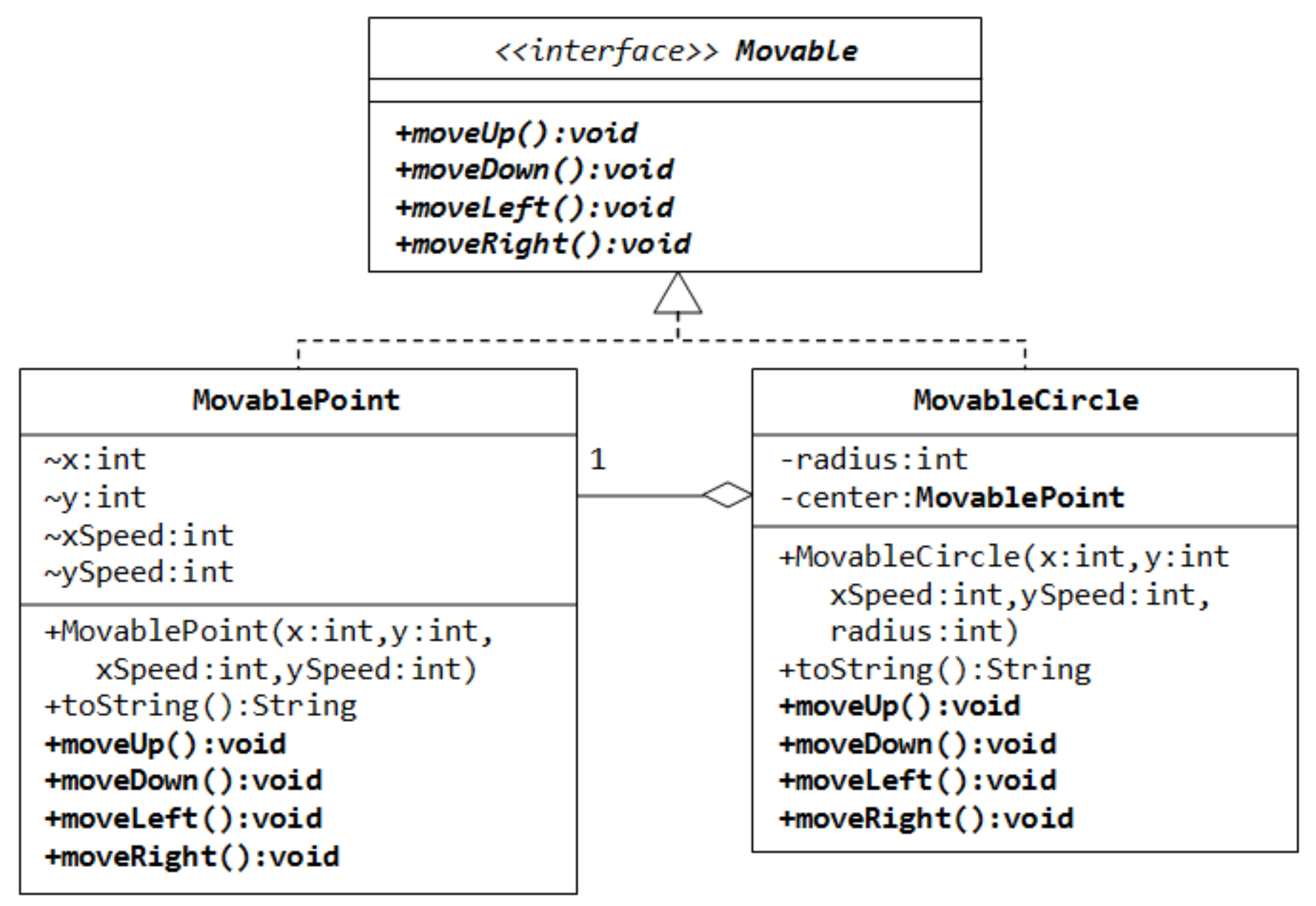
You can declare the type of a variable to be an interface

- This is just like declaring the type to be an abstract class

Important interfaces in Java's library include

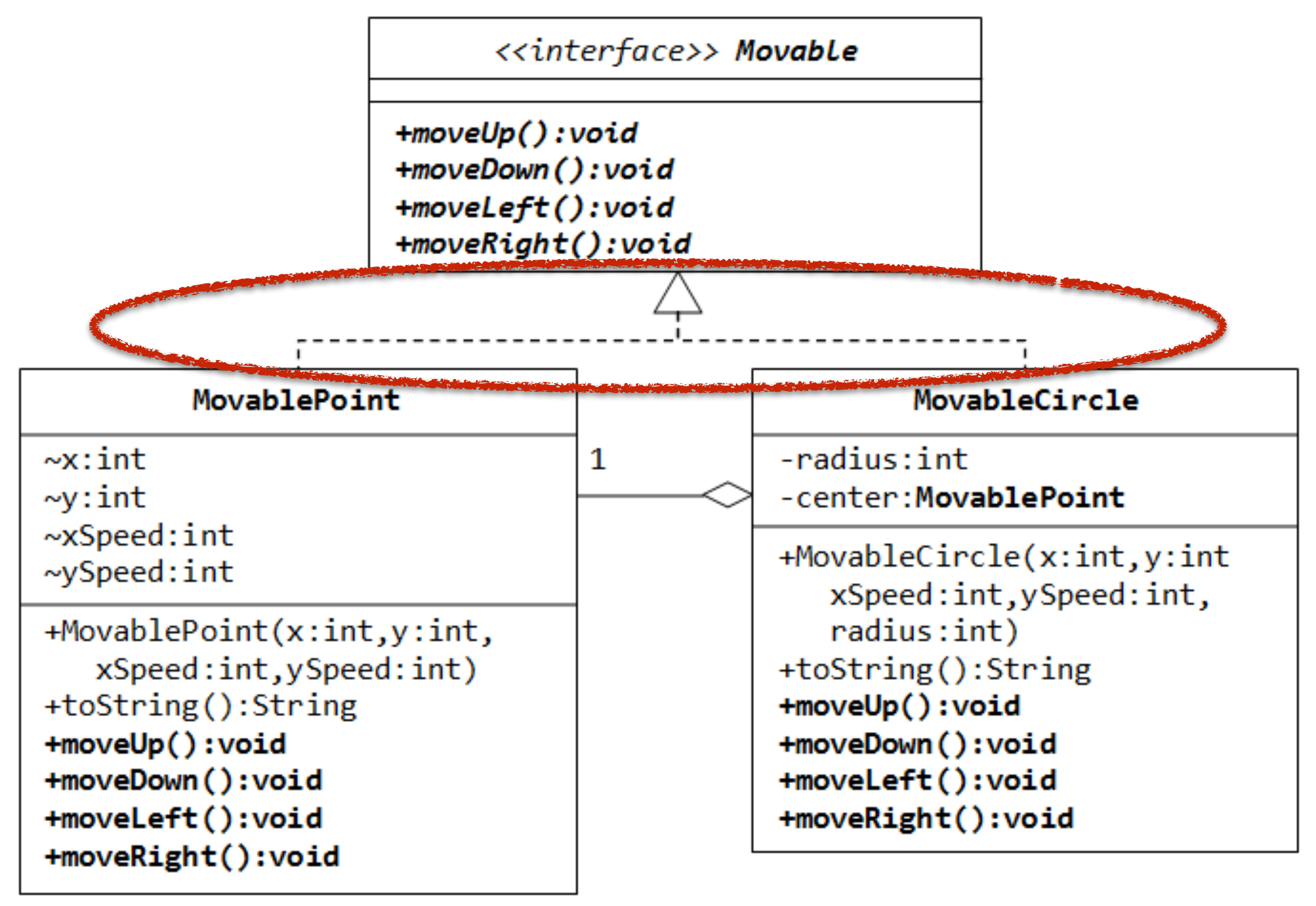
- Runnable, Collection, Iterator, Comparable, Cloneable

Interfaces



In Java: class MovablePoint *implements* Movable

Interfaces



<<interface>> *Movable*

+moveUp():void
+moveDown():void
+moveLeft():void
+moveRight():void

MovablePoint

~x:int
~y:int
~xSpeed:int
~ySpeed:int

+MovablePoint(x:int,y:int,
xSpeed:int,ySpeed:int)
+toString():String
+moveUp():void
+moveDown():void
+moveLeft():void
+moveRight():void

MovableCircle

-radius:int
-center:**MovablePoint**

+MovableCircle(x:int,y:int
xSpeed:int,ySpeed:int,
radius:int)
+toString():String
+moveUp():void
+moveDown():void
+moveLeft():void
+moveRight():void

In Java: class MovablePoint *implements* Movable

Overriding

A method would be inherited, but a subclass contains a new version instead

- For restriction
  - E.g. `scale(x,y)` would not work in `Circle`
- For extension
  - E.g. `SavingsAccount` might charge an extra fee following every debit
- For optimization
  - E.g. The `getPerimeterLength` method in `Circle` is much simpler than the one in `Ellipse`

How a decision is made about which method to run

1. If there is a concrete method for the operation in the current class, run that method.
2. Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.
3. Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.
4. If no method is found, then there is an error
   - In Java and C++ the program would not have compiled

Dynamic binding

Occurs when decision about which method to run can only be made at *run time*

- Needed when:
    - A variable is declared to have a superclass as its type, and
    - There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses

Dynamic binding

```
class Vehicle {

    public void start() {
        System.out.println("Inside start method of Vehicle");
    }
}

class Car extends Vehicle {
   @Override
    public void start() {
        System.out.println("Inside start method of Car");
    }
}
```

Dynamic binding

```
public class DynamicBindingTest {

    public static void main(String args[]) {
        Vehicle vehicle = new Car(); //here Type is vehicle
but object will be Car
        vehicle.start();    //Car's start called because start() is
overridden method
    }
}
```

Concepts that Define Object Orientation

The following are necessary for a system or language to be OO

- Identity
  - Each object is *distinct* from each other object, and *can be referred to*
  - Two objects are distinct *even if they have the same data*
- Classes
  - The code is organized using classes, each of which describes a set of objects
- Inheritance
  - The mechanism where features in a hierarchy inherit from superclasses to subclasses
- Polymorphism
  - The mechanism by which several methods can have the same name and implement the same abstract operation.

Other Key Concepts

Abstraction

- Object -> something in the world
- Class -> objects
- Superclass -> subclasses

Modularity

- An object-oriented system can be constructed *entirely* from a set of classes, where each class takes care of a particular subset of the functionality (functionality related to a given type of data), rather than having the functionality spread out over many parts of the system.

Encapsulation

- Details can be hidden in classes
- This gives rise to *information hiding*:
  - Programmers do not need to know all the details of a class

The this Keyword

- The this keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class’s *hidden data fields*.
- Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

Reference the Hidden Data Fields

```
public class F {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    F.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of F.
F f1 = new F(); F f2 = new F();

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructor

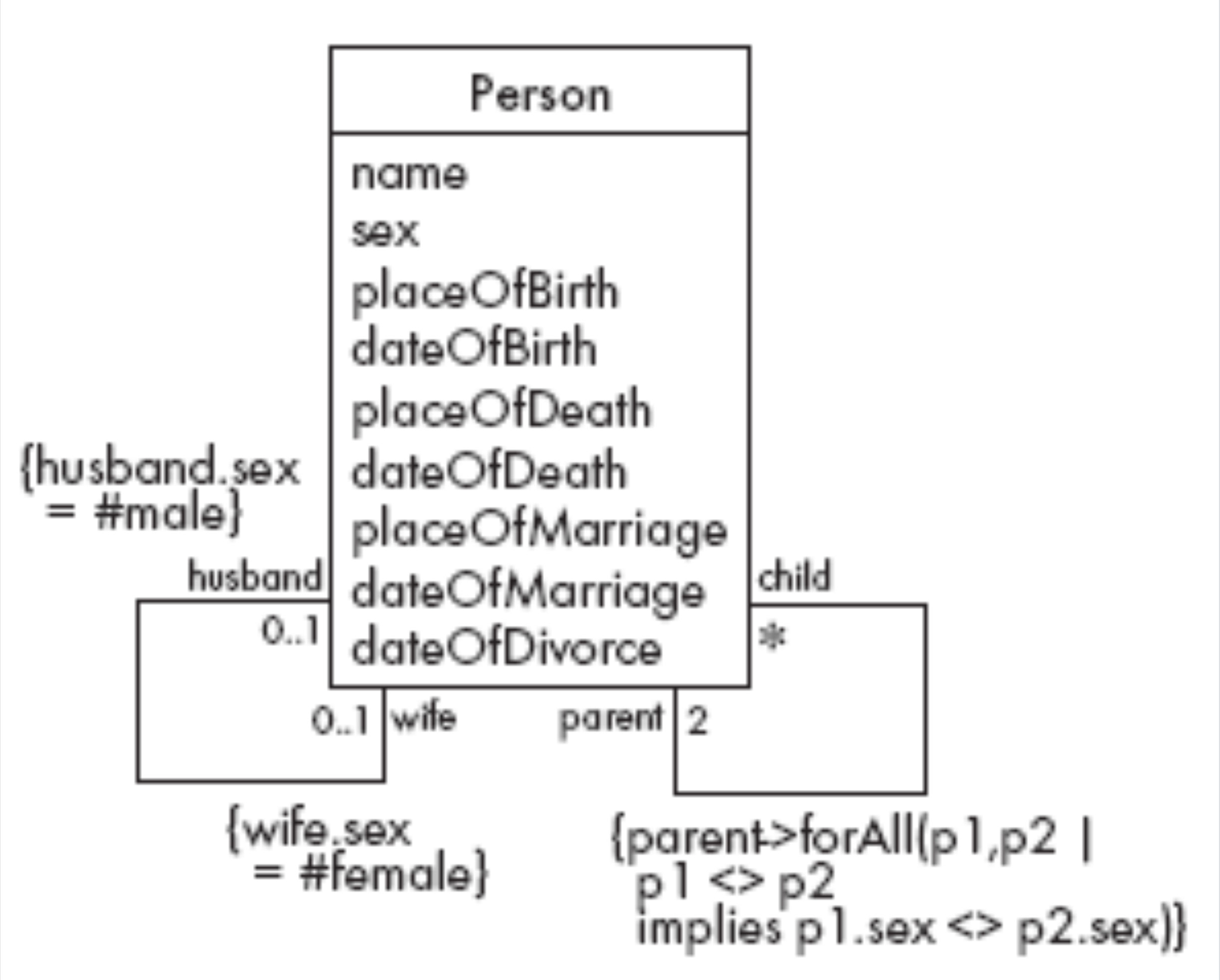
```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

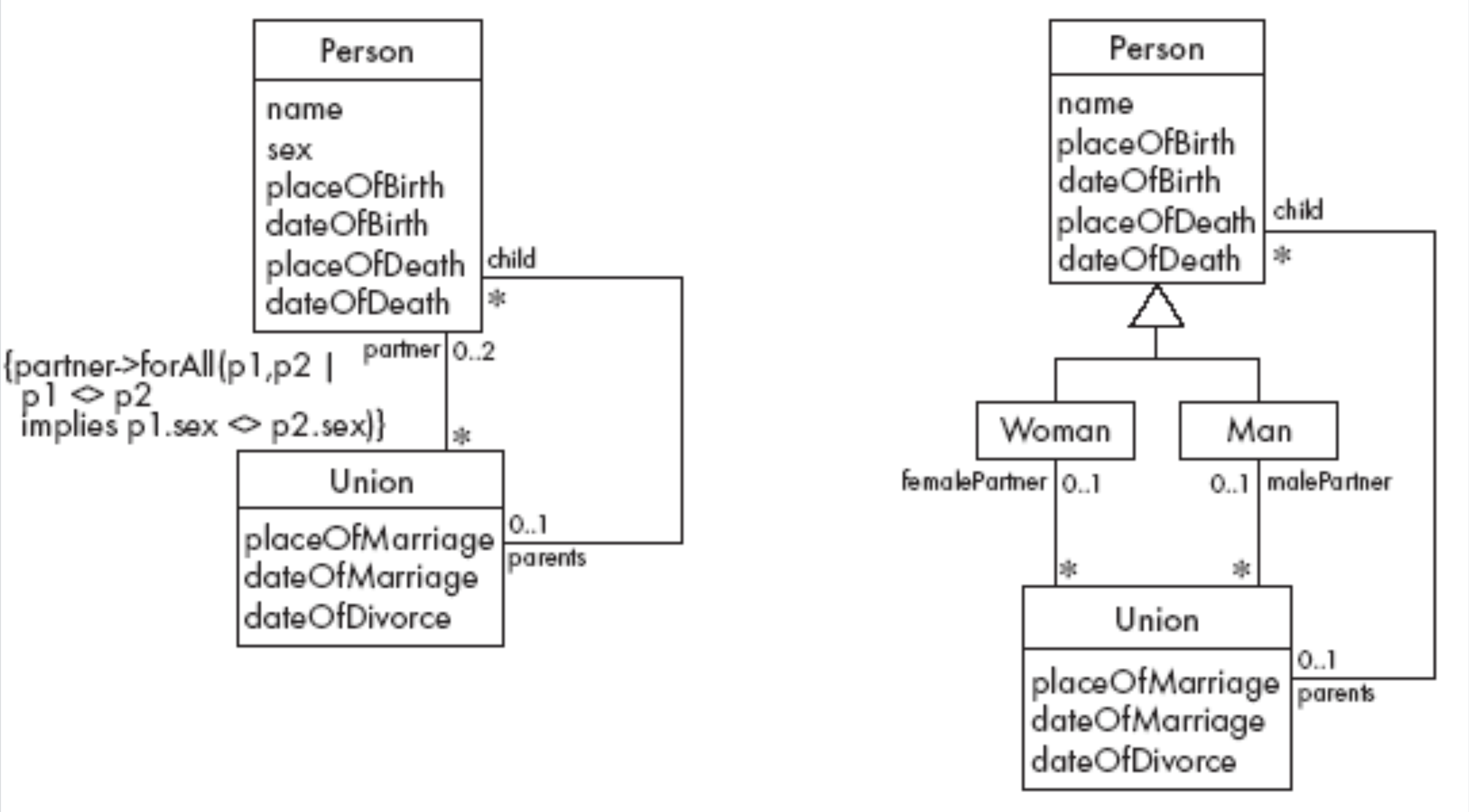
Every instance variable belongs to an instance represented by this, which is normally omitted

Generalization Example



- Problems
  - A person must have two parents
  - Marriages not properly accounted for

Generalization Example


Person
name
sex
placeOfBirth
dateOfBirth
placeOfDeath
dateOfDeath
child
*
partner
0..2
{partner->forAll(p1,p2 |
p1 <> p2
implies p1.sex <> p2.sex)}
*
Union
placeOfMarriage
dateOfMarriage
dateOfDivorce
0..1
parents
Person
name
placeOfBirth
dateOfBirth
placeOfDeath
dateOfDeath
child
*
Woman
Man
femalePartner
0..1
0..1
malePartner
*
*
Union
placeOfMarriage
dateOfMarriage
dateOfDivorce
0..1
parents