

CS 213 – Software Methodology

Sesh Venugopal

UML Class Diagram - I

UML Diagrams

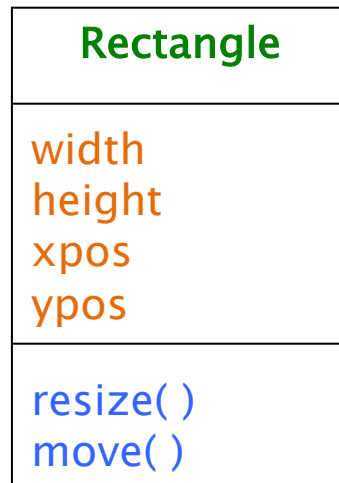
- UML stands for Unified Modeling Language, which is a (mainly) graphical notation used to express object-oriented design
- There are three kinds of UML diagrams that are used in practice:
 - **Class diagram**, used to show classes and the relationships between them
 - **Sequence diagram**, used to show sequences of activity when methods are invoked on classes (we will NOT be studying this)
 - **State diagram**, used to represent state-based designs

Class Diagram

- A class diagram shows classes and the relationships between them
- The simplest way to draw a class is like this:



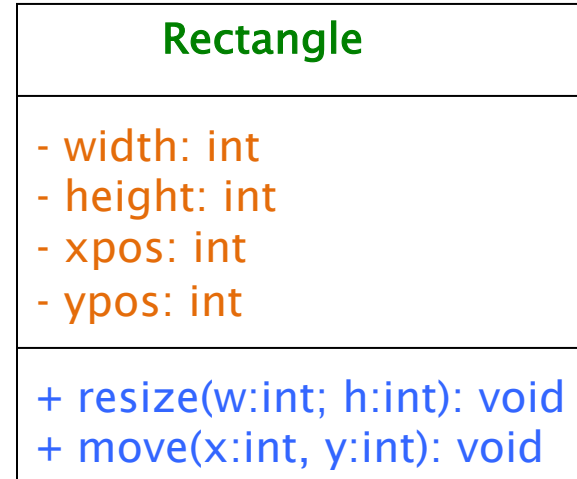
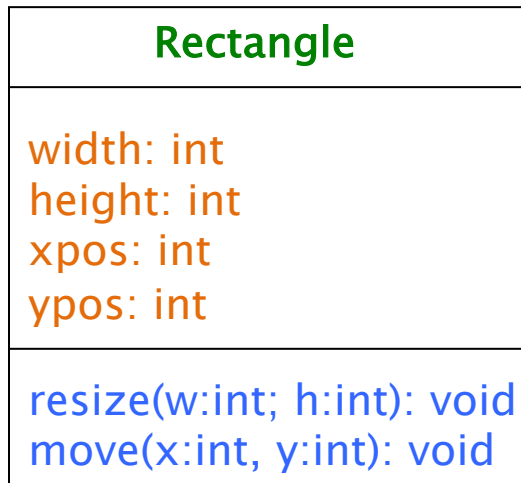
- Details may be added to the class like this:



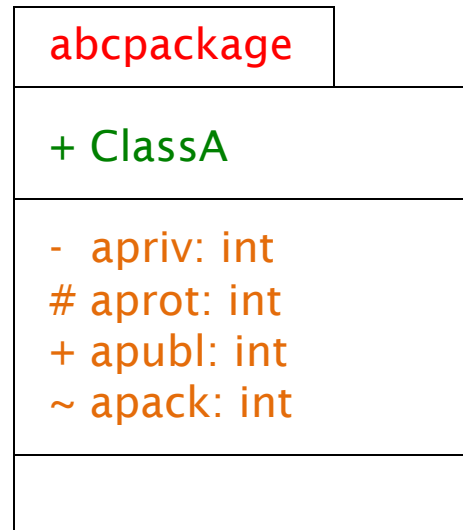
Attributes appear below the class name, and operations (methods) appear below the attributes

Class Diagram with Attributes and Methods

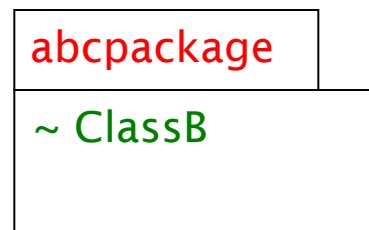
- An even greater level of detail can be added by specifying the type of each attribute, as well as type of each parameter and return type for each method:
- And the access level (private, public, etc.) for each member:



UML Notation for Access Levels

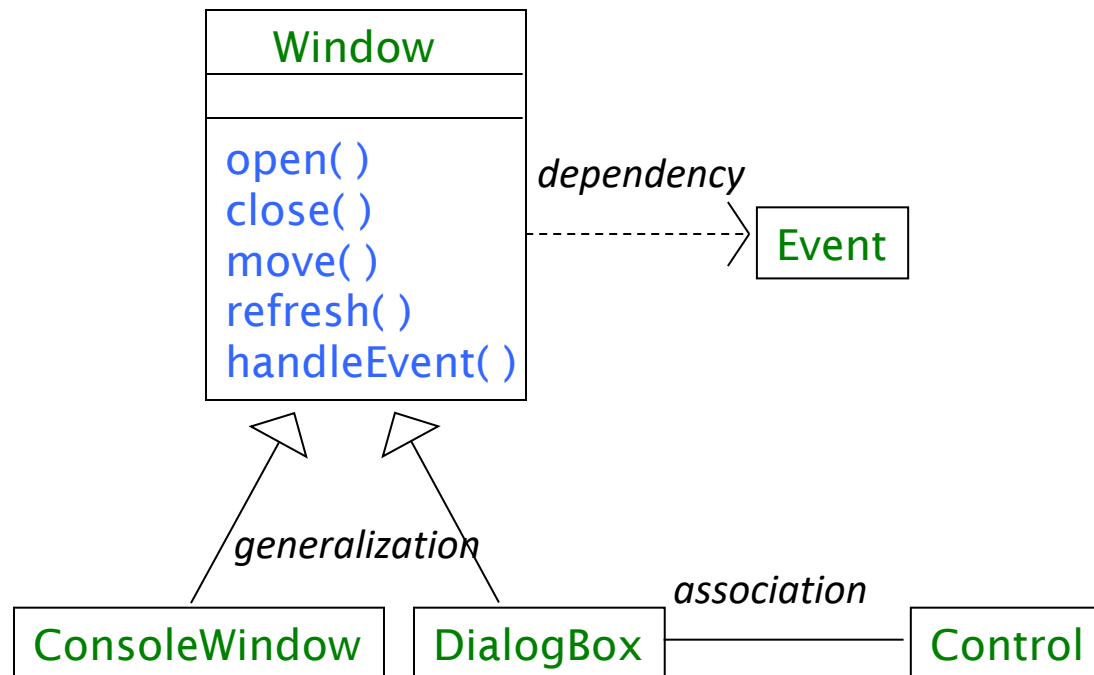


+ : public
: protected
~ : package
- : private



Class Diagram: Relationships

- Relationships between classes are essentially of three kinds: *generalization/specialization* (super/sub), *association*, and *dependency*

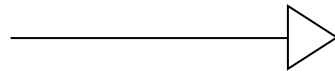


This example from “The Unified Modeling Language User’s Guide” by Booch, Rumbaugh, Jacobson

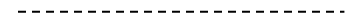
Class Diagram: Relationships

- Relationships between classes are represented by various kinds of lines

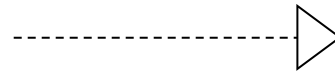
Inheritance



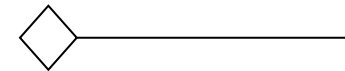
Association
Class



Interface
Implementation



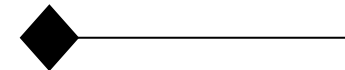
Aggregation



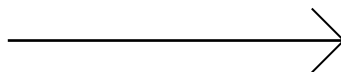
Bi-directional
Association



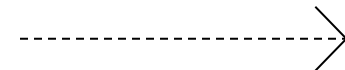
Composition



Uni-directional
Association

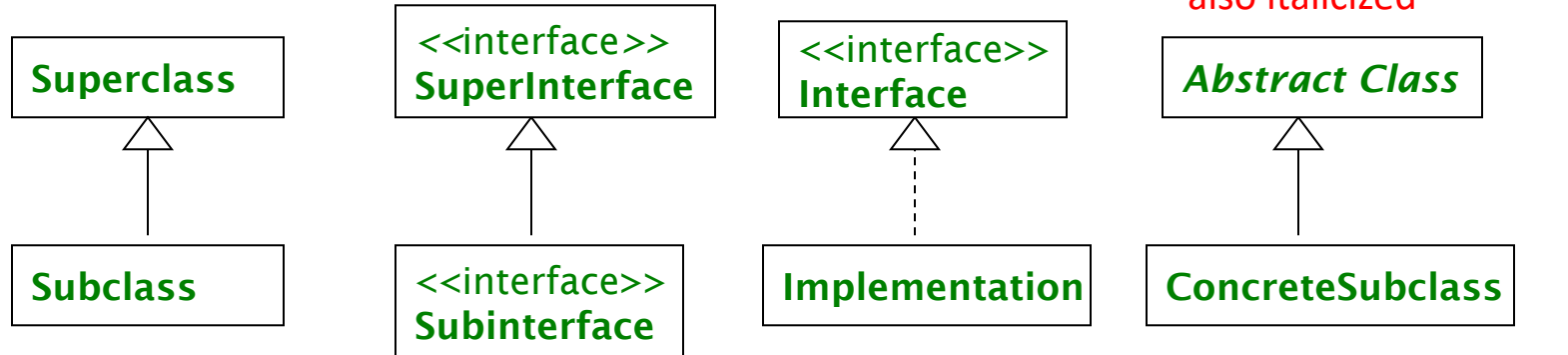


Dependency

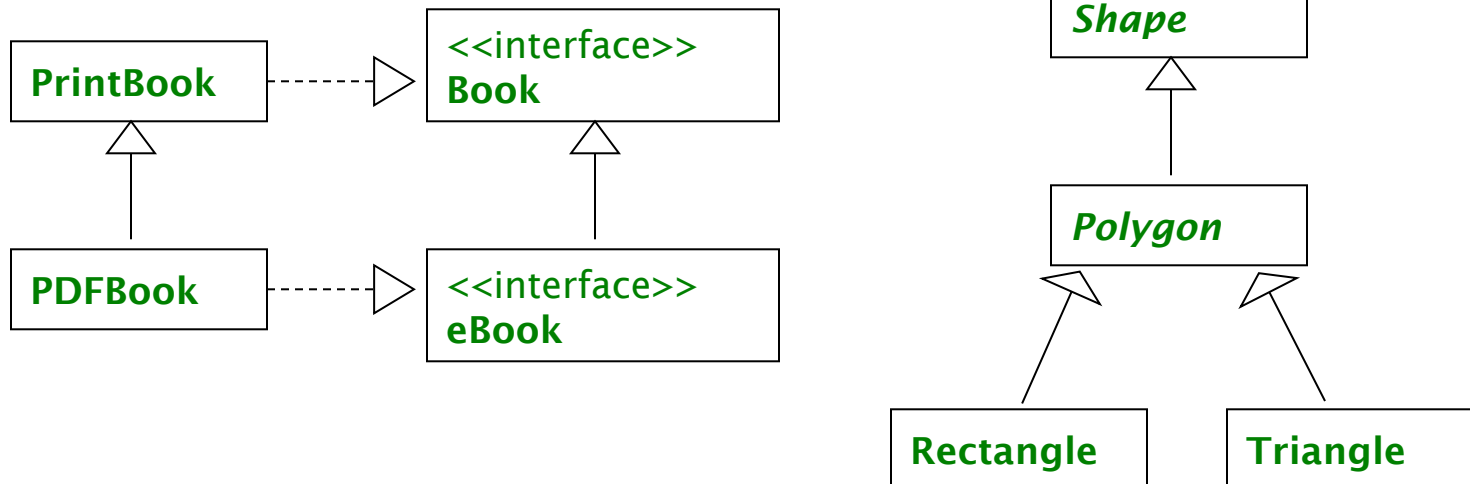


Generalization (and Interface Implementation)

- Notation



- Examples

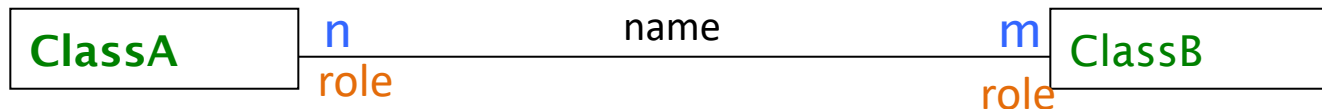


Association and Multiplicity

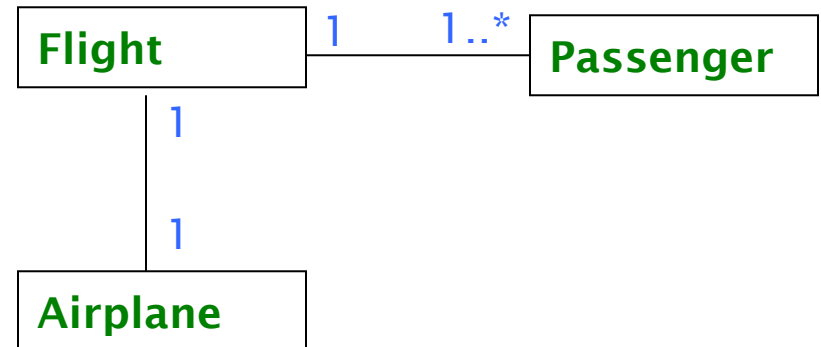
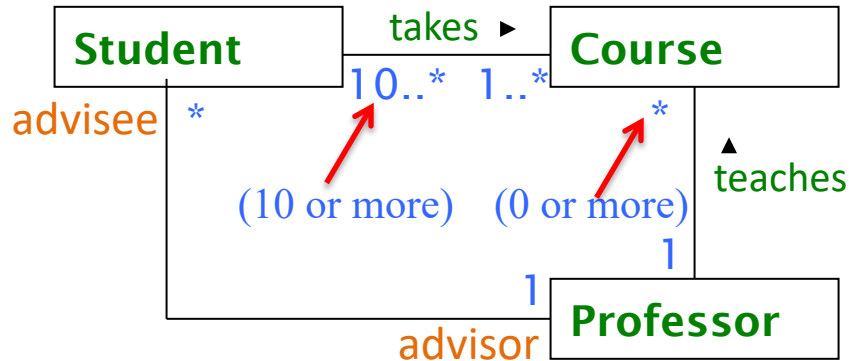
- An association is a general relationship between two classes, with options for name of association, and number of instances (multiplicity) of participation of each class

Each instance of ClassB is associated with n instances of ClassA

Each instance of ClassA is associated with m instances of ClassB



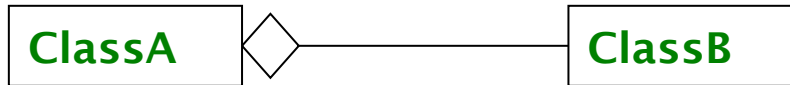
Association and Multiplicity: Examples



- Multiplicity can also be specified as one of the values an enumerated set such as 1, 3..5

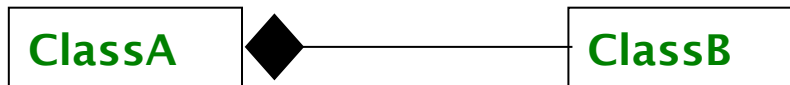
Aggregation and Composition

- Aggregation is a special kind of *association* that represents a *has-a* or whole-parts relationship – the *whole* is the aggregate class instance, and the *parts* are the component class instances



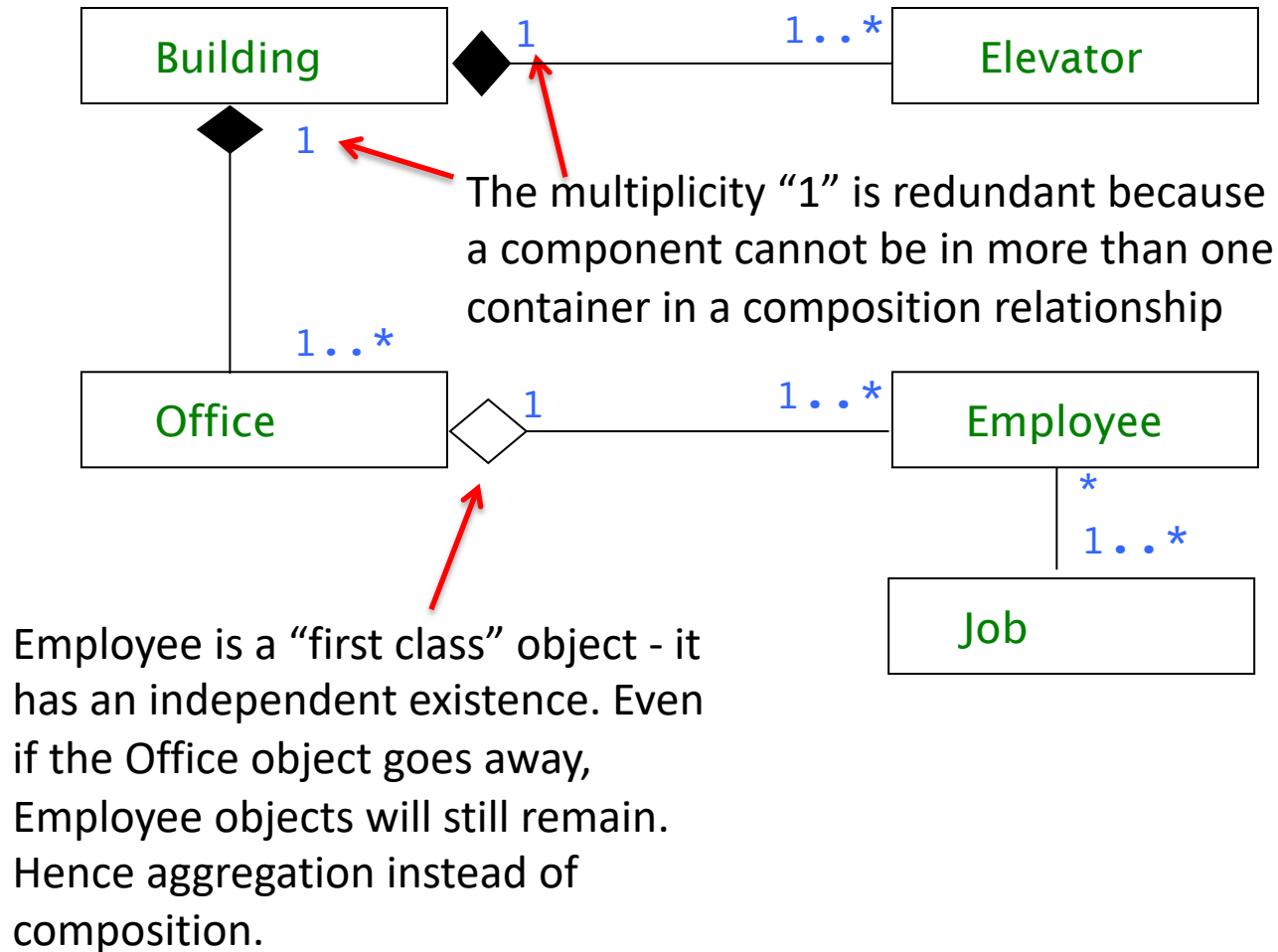
Each instance of **ClassA**
aggregates one or more
instance of **ClassB**

- Composition is a stronger form of aggregation, in which the *components* live or die with the containing class (the whole)—a deletion of the whole will lead to the deletion of the parts (an object may be a part of only one composite at a time)



Each instance of **ClassA**
is composed of one or more
instance of **ClassB**

Aggregation and Composition: Example



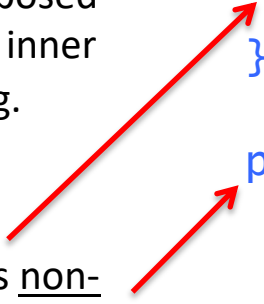
Aggregation and Composition

One possible implementation of a composition is to define the composed object (e.g. elevator/office) as an inner class of the composing object (e.g. building)

Elevator and Office are defined as non-static inner classes – creating an object of either requires a Building object

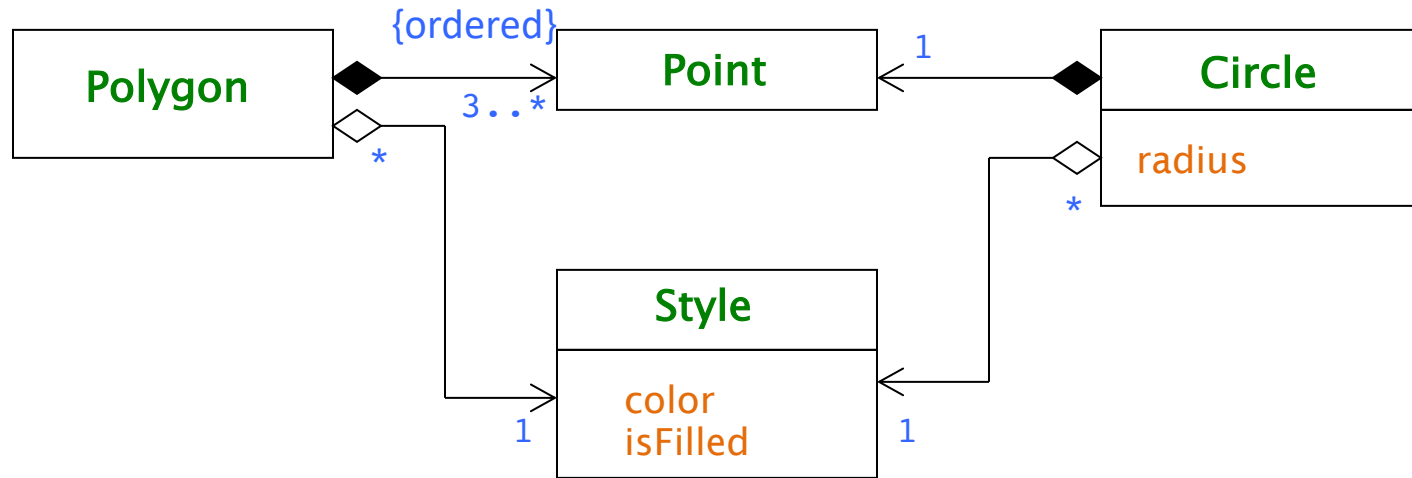
Deleting the whole must result in deleting the parts – implementation wise this applies to languages that do NOT have garbage collection (e.g. C++) because memory for components must be explicitly freed

```
public class Building {  
    private class Elevator {  
        ...  
    }  
  
    private class Office {  
        private ArrayList<Employee>  
            employees;  
        ...  
    }  
  
    private Elevator[] elevators;  
    private Office[] offices;  
  
    public Building(int enum,  
                    int onum) {  
        elevators = new Elevator[enum];  
        offices = new Office[onum];  
        ...  
    }  
    ...  
}
```



Example of Aggregation and Composition

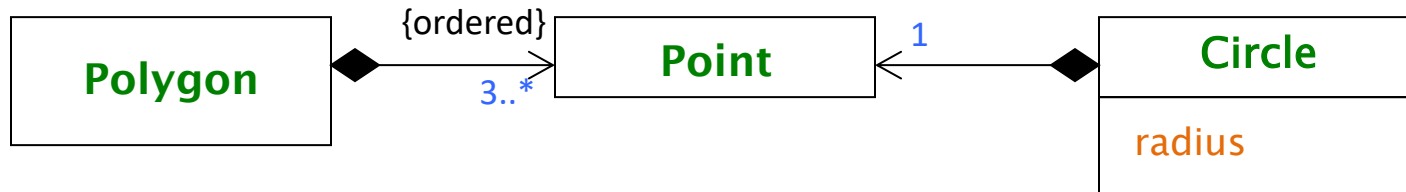
From “UML Distilled” By Martin Fowler with Kendall Scott



Point participates in two composition relationships:
what could this mean?

All associations have a directional arrow: what could this mean?

Class participates in multiple compositions



```
public class Polygon {
    // implementation must
    // ensure at least 3 points
    // in sequence
    private Point[] points;
    ...
}
```

```
public class Point {
    private int x,y;
    ...
}
```

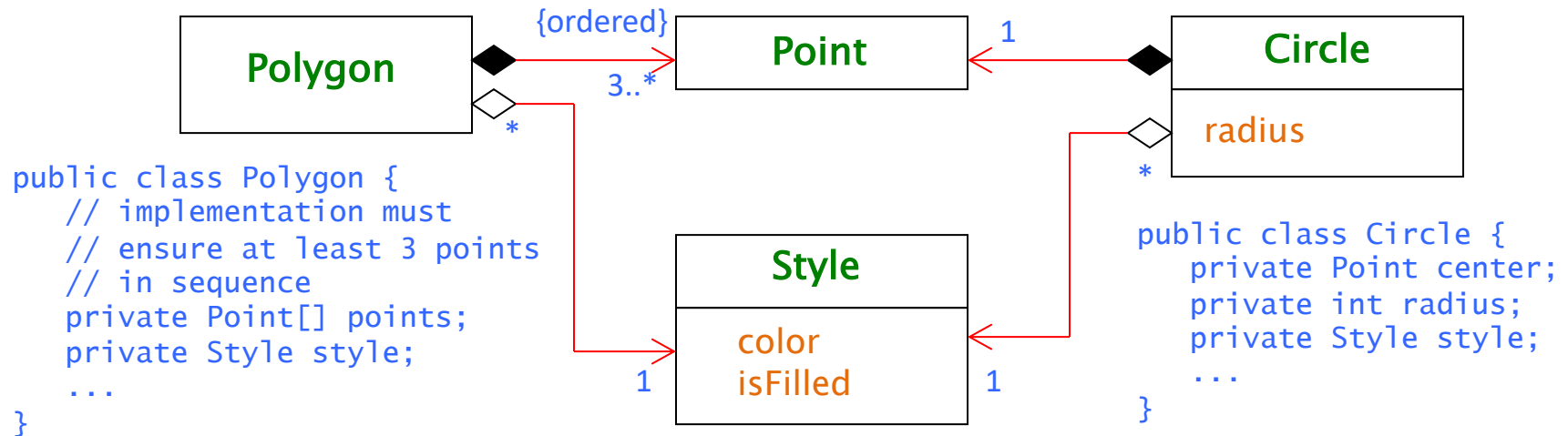
```
public class Circle {
    private Point center;
    private int radius;
    ...
}
```

Point participates in two compositions. The qualifier (design wise) is that the Point instance in the Circle is different from any of the Point instances in Polygon.

So that if a Circle instance, or Polygon instance, is no longer active, the contained Point instances can be safely destroyed.

However, in Java, which implements automatic garbage collection, this restriction does not apply. An instance of Point used in Circle can also be used in Polygon: if the Polygon instance goes out of scope, only the contained instances of Point that DO NOT have a reference from elsewhere will go out of scope as well. (If an instance is referred from a Circle instance, it will not be garbage collected.)

The meaning of directed associations



Polygon “knows” about its Style and Point associations (and so they are fields), but Style and Point do not know about their Polygon associations

```

public class Point {
    private int x,y;
    ...
    // NO REFERENCE TO
    // Polygon or Circle
}

```

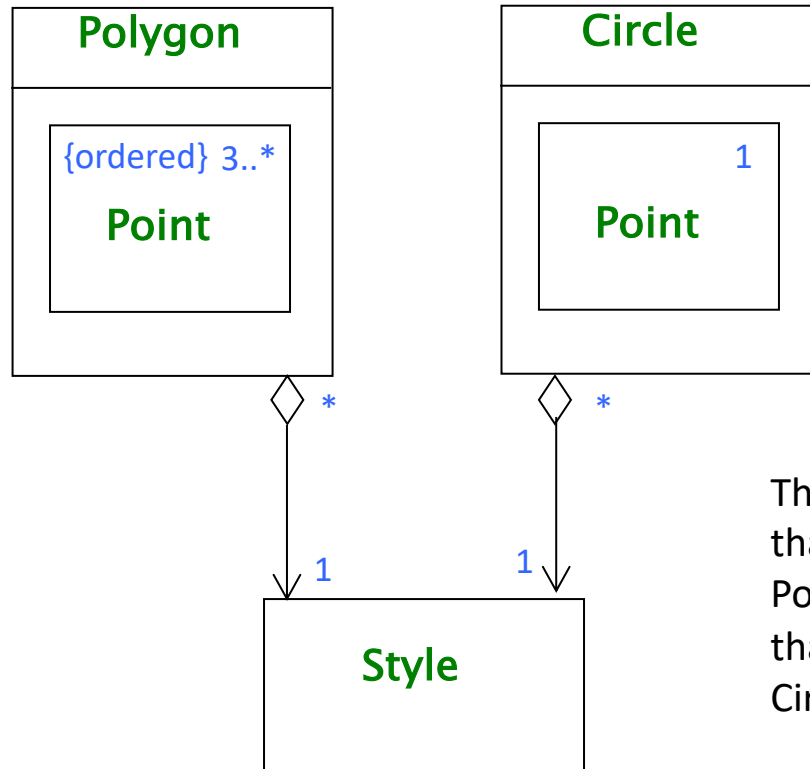
```

public class Style {
    private Color color;
    private boolean isFilled;
    ...
    // NO REFERENCE TO
    // Polygon or Circle
}

```

Circle “knows” about its Style and Point associations (so, fields), but Style and Point do not know about their Circle associations

Alternative Notation for Composition



This notation makes it more obvious that the **Point** instances contained in **Polygon** are (design wise) different than the **Point** instances contained in **Circle**