Lecture 3

Objects, UML, and Java

Suleman Shahid
suleman.shahid@lums.edu.pk

Abdul Ali Bangash
abdulali@lums.edu.pk

Department of Computing Science

LUMS University

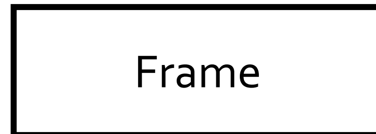
CMPUT 360 – Introduction to Software Engineering

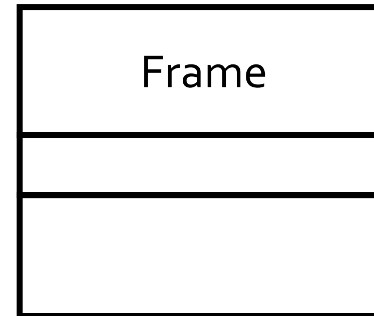
Slides adapted from Henry Tang, Dr. Abram Hindle

Images reproduced in these slides have been included under section 29 of the Copyright Act, as fair dealing for research, pri vate study, criticism, or review. Further distribution or uses may infringe copyright.

Java and UML Class

- ```
  public class Frame { // version 0
       // represent a ‘window’
       /* body of class definition goes here */
  }
  ```





UML class notation
  ```

Encapsulation

- Class:
  - Access control for attributes and methods
    - E.g., public or private
  - Access is not the same as visibility
  - “Design by contract”
    - Public interface represents a contract between the developer who implements the class and the developer who uses the class

Java Class

- ```
  public class Frame { // version 1
      // private implementation

      private datatype variablename;

      // public interface

      public Frame( arguments ) {
          // implementation of constructor
      }

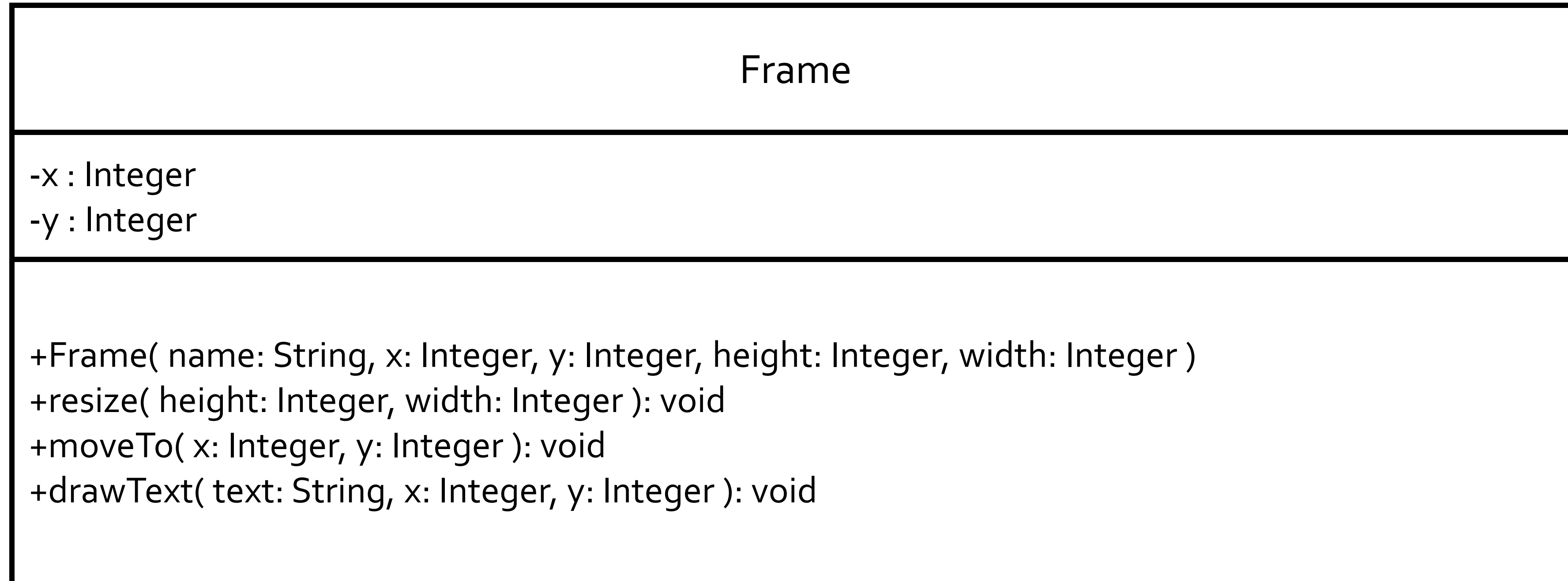
      public returntype methodname( arguments ) {
          // implementation of method
      }
  }
  ```
  ```

Java Class

- ```
  public class Frame { // version 2
      private int x;
      private int y;
      …
      public Frame ( String name,
          int x, int y, int height, int width ) { … }

      public void resize(
          int newHeight, int newWidth ) { … }

      public void moveTo(
          int newX, int newY ) { … }

      public void drawText( String text,
          int x, int y ) { … }
  }
  ```

- ```
  Frame f = new Frame( "Untitled", 0, 0, 480, 640 );
  ```

UML Class



– Private
+ Public

Decomposition

- Association relationship:
  - “Some” relationship between classes
    - E.g., between Book and Patron

UML Association

- Read class diagram using “objects”
  - A Food *object* goes well with a Juice *object*
  - A Food *object* is associated with o or more Juice *objects*
  - A Juice *object* is associated with o or more Food *objects*

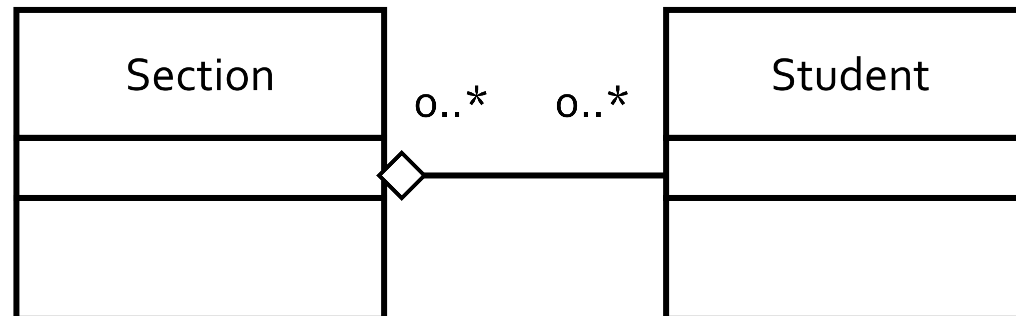
Decomposition

- Aggregation relationship:
  - Weak “has-a” relationship
  - Whole “has-a” part
  - A part may belong to (be shared with) other wholes
    - E.g., a Section and a Student

Java and UML Aggregation

- Dynamic number of aggregated objects:

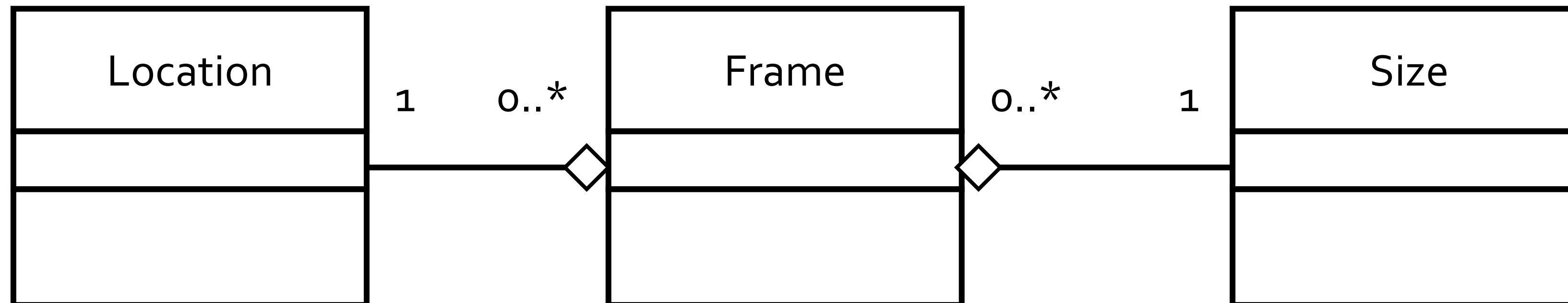
```
public class Section {
    private ArrayList<Student> roster;
    …

    public Section() {
        roster = new ArrayList<Student>();
        …
    }
    public void add( Student s ) { … }
}
```



Java and UML Aggregation

- Fixed number of aggregated objects:

```
public class Frame {

    private Location defaultLocation; // shared
    private Size defaultSize; // shared
    …
}
```

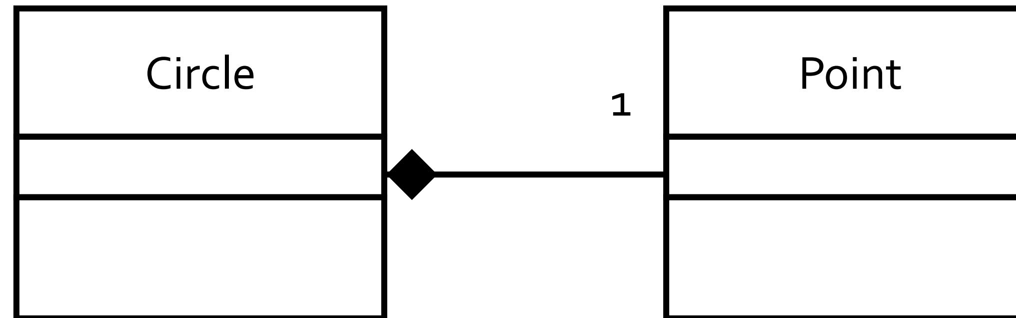


Decomposition

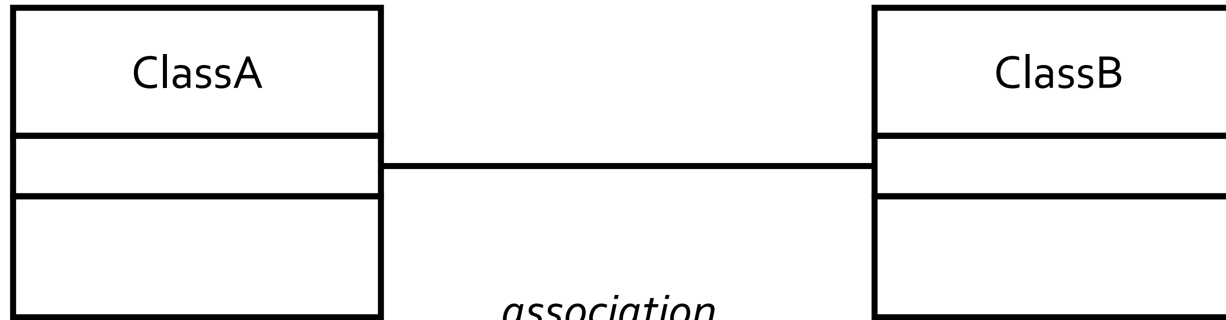
- Composition relationship:
  - Strong “has-a” relationship
  - Exclusive containment of parts
  - Related object lifetimes
    - The whole cannot exist without having the parts; if the whole is destroyed, the parts should also be destroyed
  - Often access the parts through the whole

UML Composition

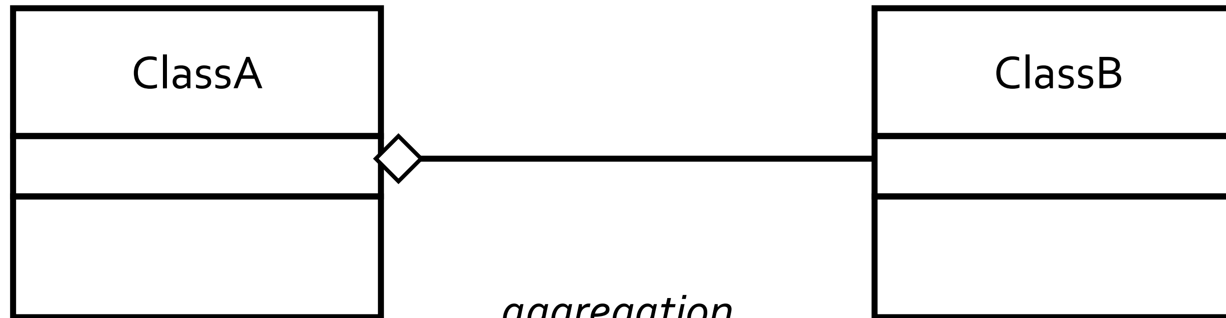
- Contained *objects* are exclusive to the container



- A Circle object has a Point object that is exclusive to it (however, other objects may contain Point objects, just not this one)



whole *part*

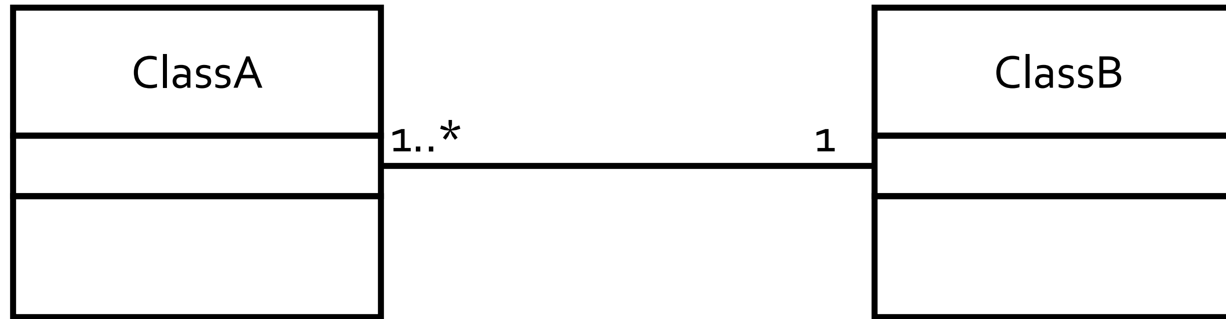


ClassA and ClassB can created/destroyed independently

composition

If ClassA instance is deleted, all of its ClassB instances get deleted too

Navigability

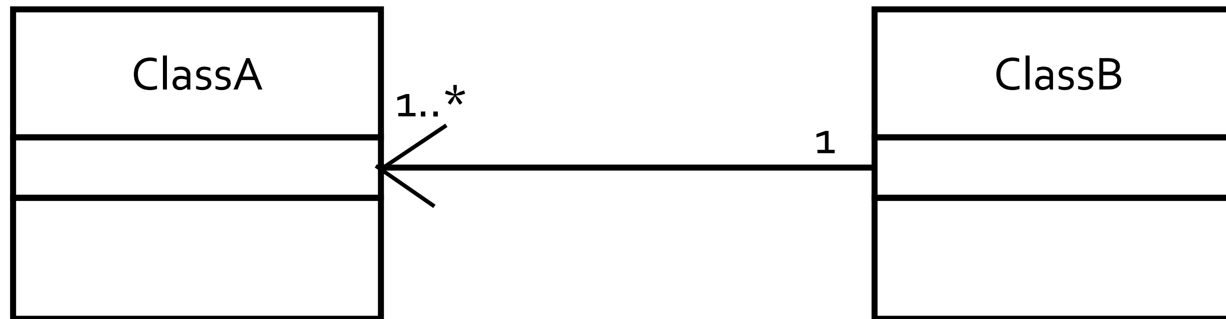


Missing navigability

Usually implies:

A instances have references to one B

B instances have references to one or more A

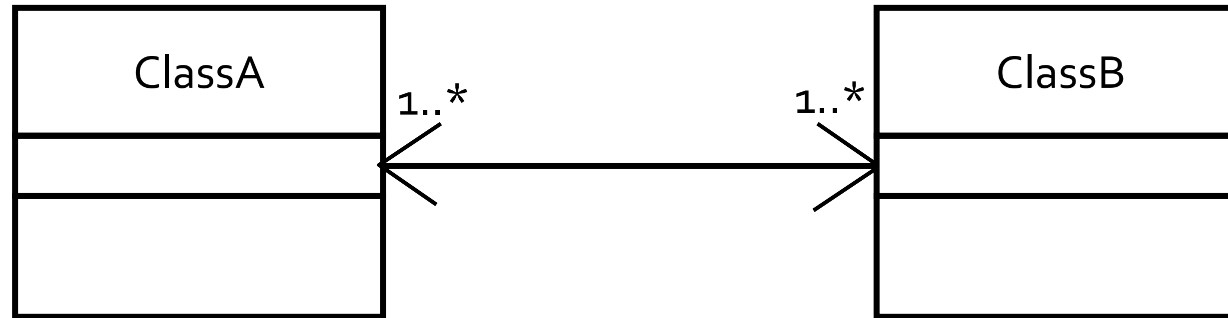


Navigability

← Arrow

B instances have references to one or more A

A instances DO NOT have reference(s) to B



Explicit two-way navigability

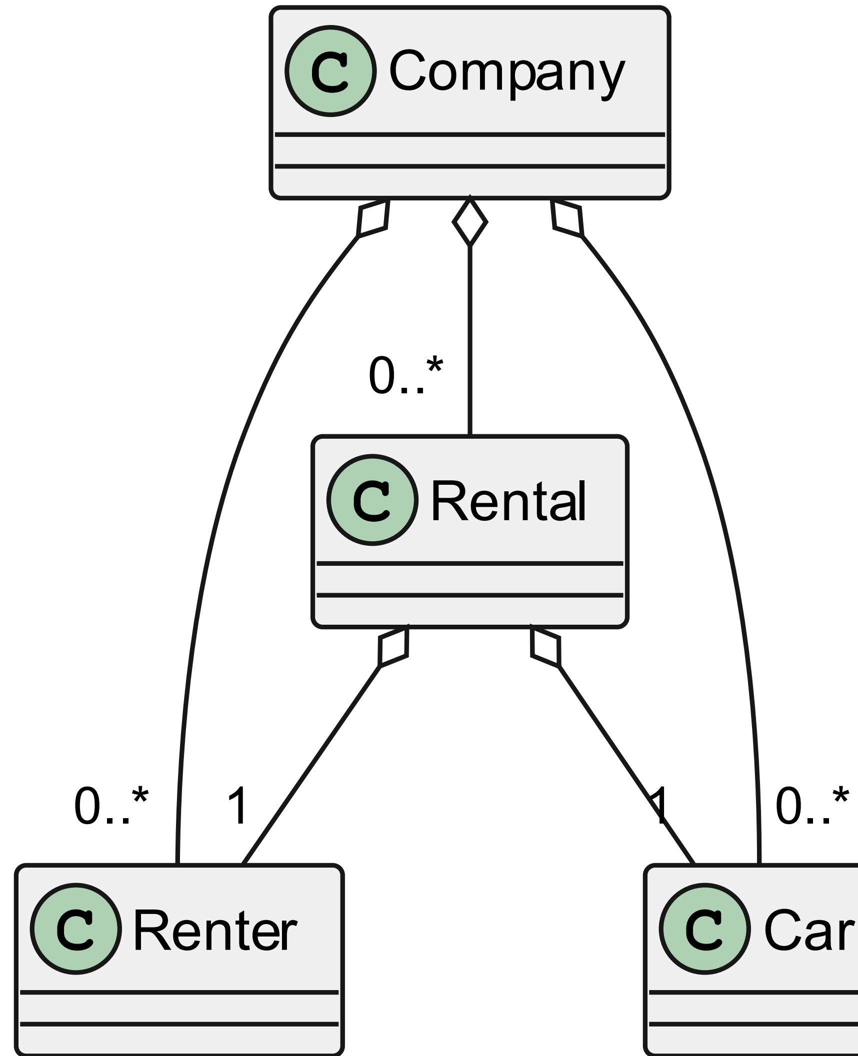
(rare)

A instances have references to one or more B

B instances have references to one or more A

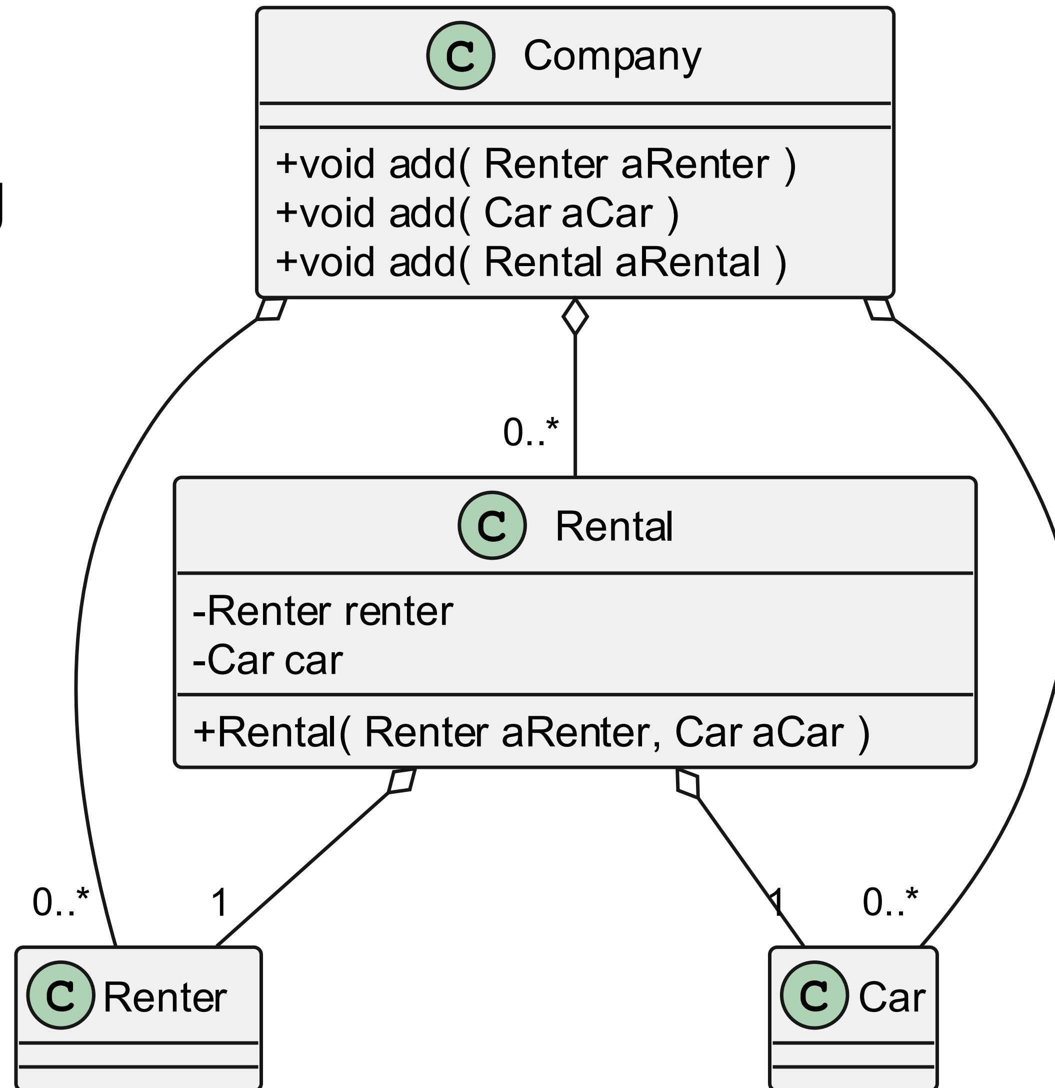
Exercise

- Analyze a UML class model for a car rental company that keeps track of cars, renters, and renters renting cars.


Company
0..*
Rental
0..*
1
1
0..*
Renter
Car

Exercise

- Implement the corresponding Java code.



Generalization

Generalization

- Look for commonalities:
  - Common attributes
    - E.g., all vehicles have…
  - Common methods (behaviour)
    - E.g., all vehicles can…
- Generalize:
  - Find what is common and factor it out into a more general “base” abstraction

Generalization

- Implementation inheritance:
  - Generalize about method signatures, method implementations, and/or attributes
    - I.e., classes having these in common

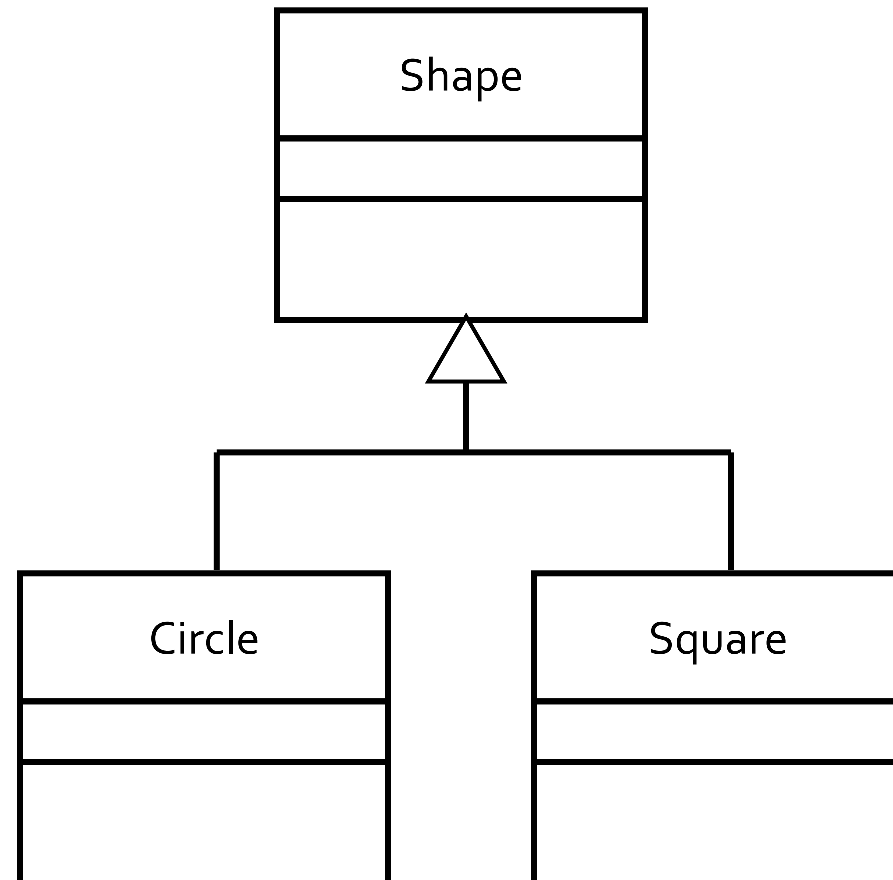
Implementation Inheritance

- General part:
  - A base class (or "superclass") defines the attributes and methods to be shared
- Specific part:
  - A derived class (or "subclass") is endowed with the attributes and methods of its base class
  - A subclass may "extend" a superclass by adding attributes and methods, or overriding an existing method

Java Implementation Inheritance

- ```
  public class Shape { // superclass
      protected Location myLocation;
      public Shape() { … }
      public void setLocation( Location p ) { … }
      public Location getLocation() { … }
  }
  ```

- ```
  public class Circle extends Shape { // subclass
      private int diameter;
      public Circle() { … }
      public void setDiameter( int d ) { … }
      …
  }
  ```

- ```
  public class Square extends Shape { // subclass
      private int side;
      public Square() { … }
      public void setSide( int s ) { … }
  }
  ```
  ```

UML Inheritance

- Implementation inheritance relationship:
  - “Is-a” relationship between classes
  - I.e., subclass “is-a” kind of superclass
  - I.e., subclass “extends” superclass
- E.g., Circle “is-a” kind of Shape



Generalization Principles

- Inappropriate inheritance:
  - Subclass inherits from superclass but “is-a” (is a kind of) relationship *does not* exist
  - If “is-a” test fails:
    - Likely not appropriate
  - If “is-a” test succeeds:
    - *May* or *may not* be appropriate

Generalization Principles

- Liskov substitution principle:
  - An instance of the subclass should be substitutable anywhere a reference to a superclass object is used
  - ```
    Shape s;
    s = new Circle(); // instance of subclass
    …
    Location l = s.getLocation(); // superclass method
    ```
    ```

Inheritance Example

- Suppose:
  - class Dog
    - Provides bark(), fetch()
  - class Cat extends Dog
    - “Hides” bark(), “hides” fetch(), and adds purr()
- Question:
  - Cat “is a” Dog?

Inheritance Example

- Suppose:
  - class Window
    - Provides show(), move(), resize()
  - class FixedSizeWindow extends Window
    - “Hides” resize()
- Question:
  - FixedSizeWindow “is a” Window?

Inheritance Example

- Suppose:
  - class ArrayList
    - Provides add(), get(), remove(), …
  - class ProjectTeam extends ArrayList
- Question:
  - ProjectTeam “is a” ArrayList?

Lecture 3 ends here …