Computer Science II
Handout 5

Member modifiers

- The *members* of a class are the collection of its *variables* and *methods*
- Declaring each of these is done with several keyword modifiers

```
public
private
static
final
```

Member modifiers – public

- Public members can be accessed by any class, either from within the class where they are defined, or without

```
public class Simple {
      public int amount = 0;
      public void doNothing() { }
}


public class SimpleDemo {
      public static void main(String[] args) {
            Simple s = new Simple();
            s.amount = 5;   // Value within ‘s’ is now 5
            s.doNothing();  // Method is called within ‘s’
      }
}
```

Member modifiers – private

- Private members can *only* be directly accessed from within the class where they are defined

```
public class Simple {
      private int amount = 0;
      private void doNothing() { }
}


public class SimpleDemo {
      public static void main(String[] args) {
            Simple s = new Simple();
            s.amount = 5;    // Compilation error
            s.doNothing();   // Compilation error
      }
}
```

Member modifiers – static

- Static members do *not* belong directly to any instance of the class

```java
public class Simple {
      public static int amount = 0;
      public static void doNothing() { }
}


public class SimpleDemo {
      public static void main(String[] args) {
            Simple s = new Simple();
            s.amount = 5;        // Value within ‘Simple’ is now 5
            // Valid code, but bad! Why?

            Simple.doNothing();  // Method called within ‘Simple’
      }
}
```

Member modifiers – final

- Final members can *not* be modified after they are assigned a value

```java
public class Simple {
      private static final int AMOUNT = 0; // ALL_CAPS by convention
      private final void doNothing() { }
}


public class SimpleDemo {
      public static void main(String[] args) {
            //Simple s = new Simple();
            Simple.AMOUNT = 5;   // Compilation error
            Simple.doNothing();  // We will re-visit this later
      }
}
```

Member modifiers

- A member may be either:
  - public or private
  - static or not
  - final or not

- A member may be any combination of modifiers, taking (at most) one from each line above:

```
public static final      int            x;
private final            double         value;
public                   void           doStuff() {
private static           boolean        check() {
```

- Modifiers always come immediately before the return/data type

Member modifiers – access modifiers

- The **`public`** keyword is one we have most commonly used, and gives full access to all classes

- The **`private`** keyword limits access to variables *and* methods
  - Useful for helper, support, or internal methods
  - Private methods can *only* be called from within the same class (not necessarily the same instance)

Member modifiers – in UMLs

+ public member

- private member

+ public static member

- private static member

- You can generally ignore final modifiers in your UML diagram, although the ALL_CAPS convention may give a hint

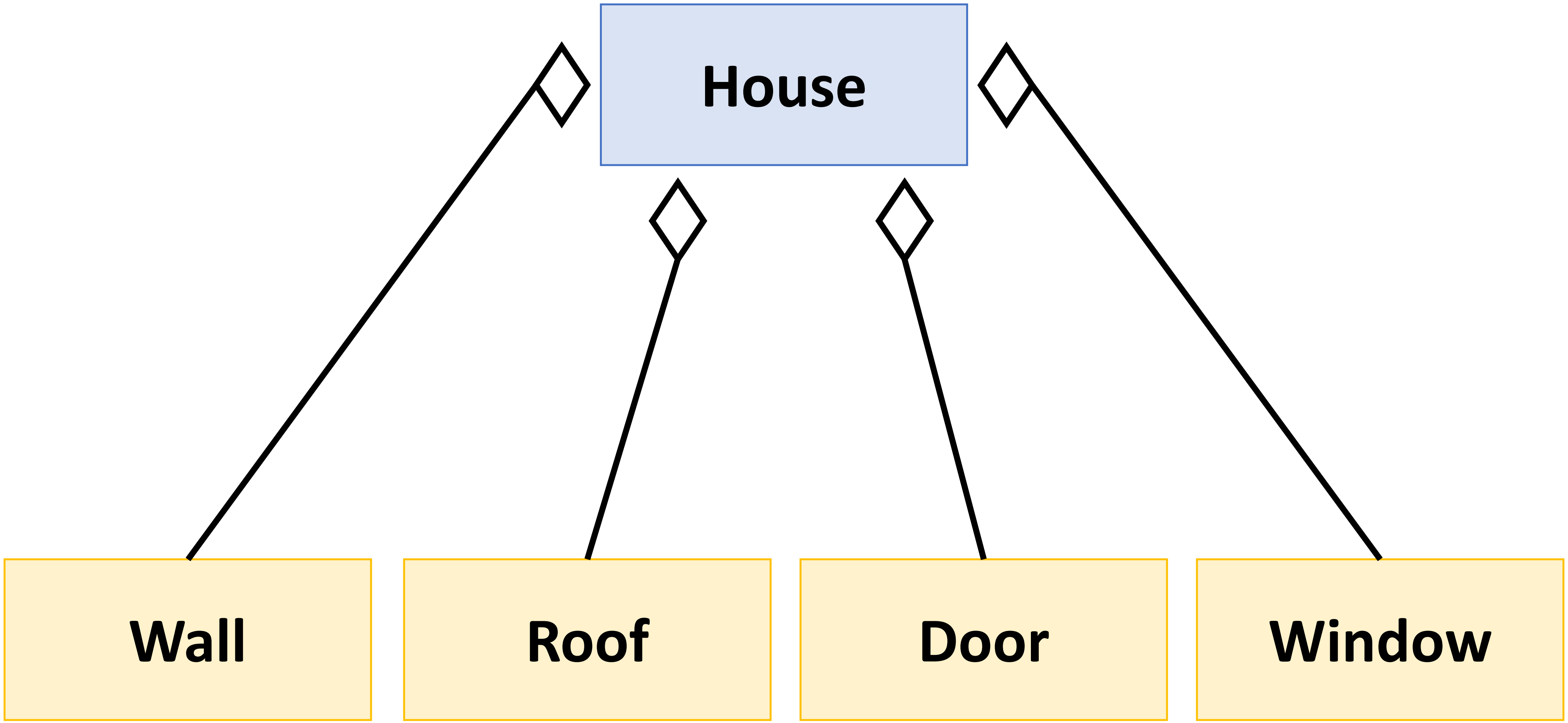
Aggregation of classes

- In the real world, objects are often a collection of other objects
  - A house is made up of walls, a roof, doors, windows, etc.
  - A car is made up of a body, a windshield, an engine, tires, etc.

- An *aggregate* is “a whole formed by combining several elements”

- This relationship between classes can be modeled in UML diagrams with a solid line ending in a diamond
  - Usually described with the phrase “has a”
  - The diamond indicates the *owning* class

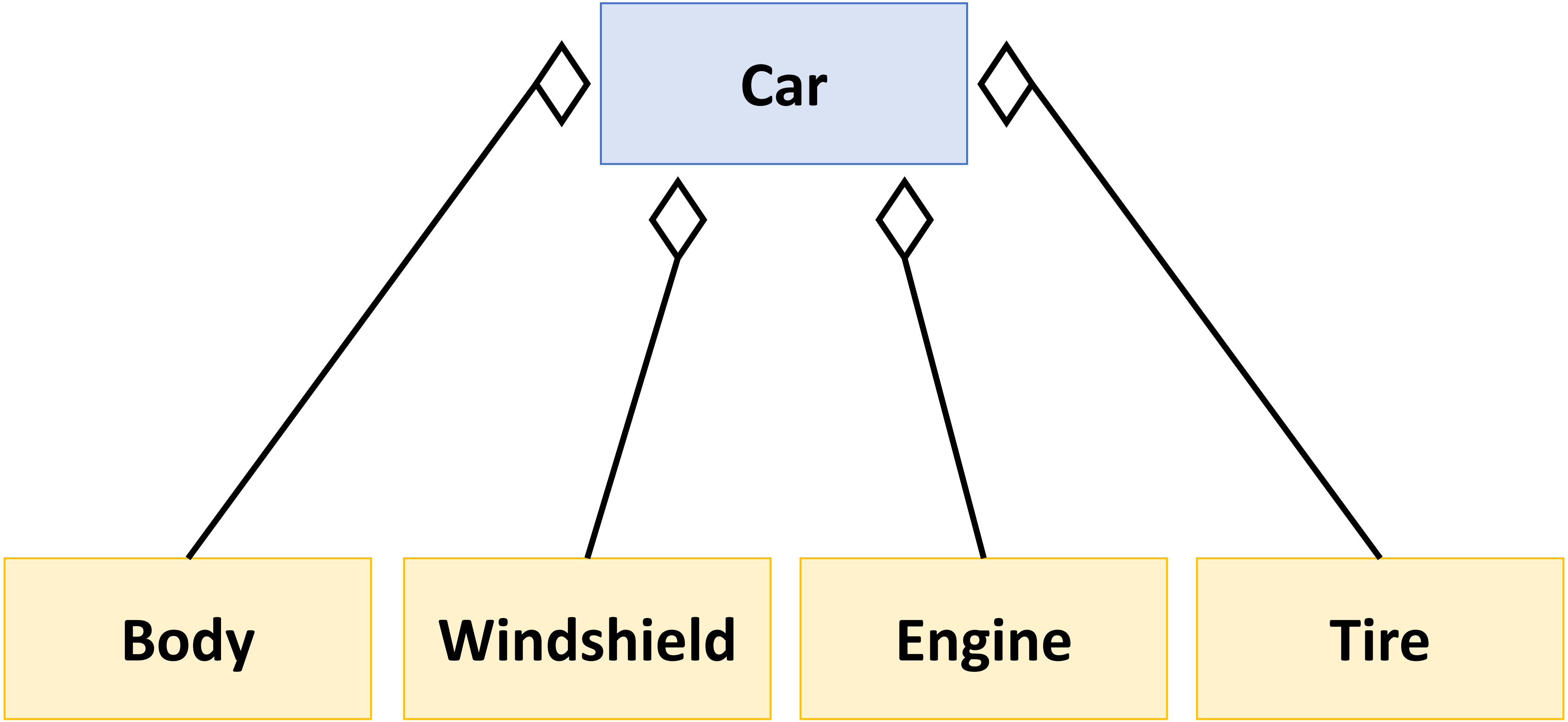
Aggregation of classes – UML



Aggregation of classes – UML

What about for a car that is made up of a body, a windshield, an engine, and tires?

Aggregation of classes – UML



Car

Body

Windshield

Engine

Tire

Aggregation of classes – UML

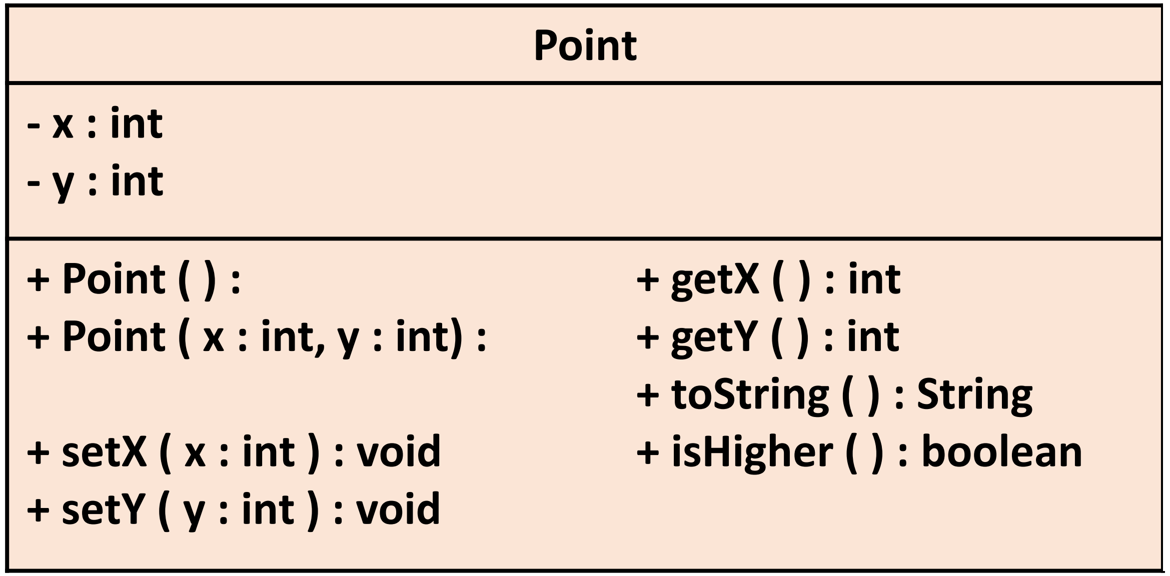
What about for a client bank account that includes a chequing account, savings account, mortgage, and credit line?

Aggregation of classes – UML

BankAcct

Aggregation of classes – UML

What about for a Circle that has a Point to indicate its centre?



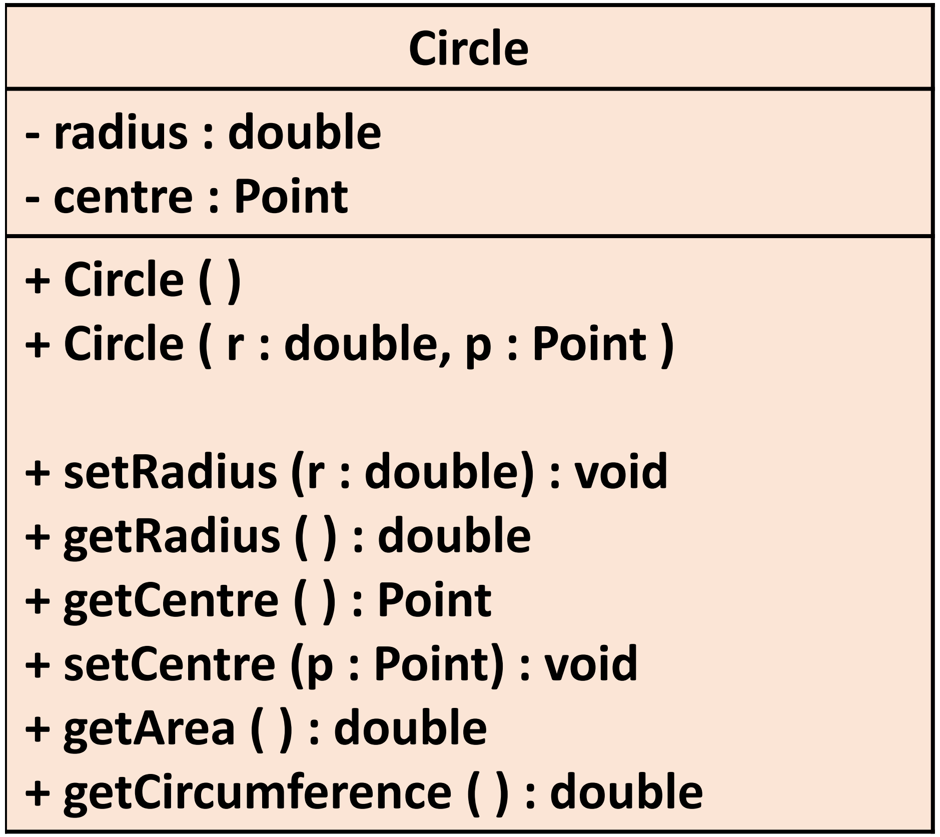
Point

- x : int
- y : int

+ Point ( ) :
+ Point (x : int, y : int) :

+ setX (x : int) : void
+ setY (y : int) : void

+ getX ( ) : int
+ getY ( ) : int
+ toString ( ) : String
+ isHigher ( ) : boolean



Circle

- radius : double
- centre : Point

+ Circle ( )
+ Circle (r : double, p : Point)

+ setRadius (r : double) : void
+ getRadius ( ) : double
+ getCentre ( ) : Point
+ setCentre (p : Point) : void
+ getArea ( ) : double
+ getCircumference ( ) : double

Aggregation of classes – in code

```
public class Circle {
    private Point centre;
    private double radius;

    public Circle() { }

    // etc...

}
```

Aggregation of classes – example

Create a **`Course`** class that will hold:

- The course name
- The instructor’s last name and first name
- The textbook title and cost

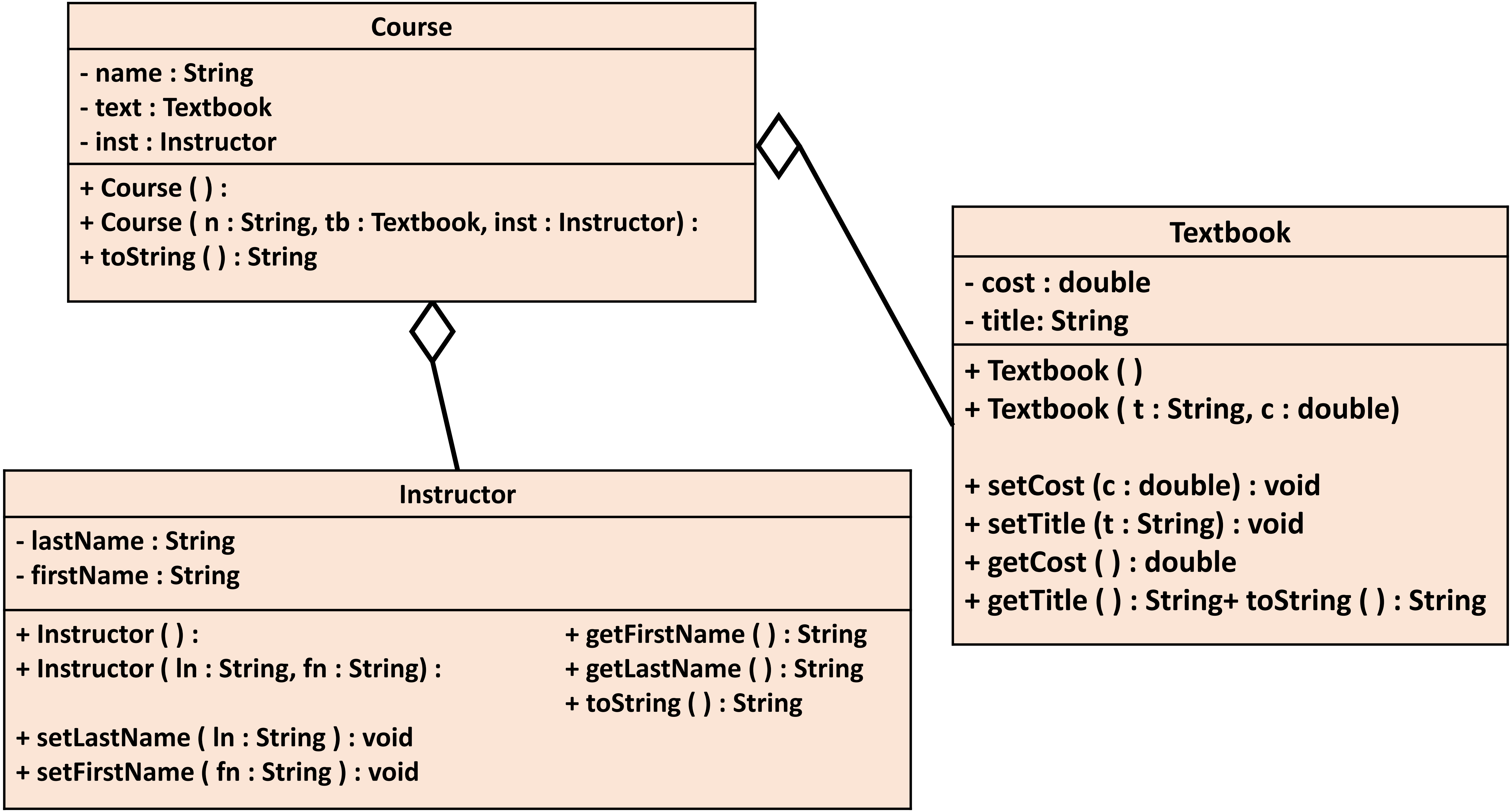
This could work by putting everything in one class but …

… an important principle of the OOP approach is keeping related data and operations *together*

- So **`Course`** should use an **`Instructor`** and a **`Textbook`** class

Aggregation of classes – example

1. Start by separating data/operations into distinct classes
   - We have **`Course`**, **`Textbook`**, and **`Instructor`**
2. Create the UML diagram for each class, then aggregate them appropriately
3. Create and compile the “supporting” classes first
   - This makes designing the “owning” class easier
4. Create and compile the “owning” class last



Course

- name : String
- text : Textbook
- inst : Instructor

+ Course ( ) :

+ Course (n : String, tb : Textbook, inst : Instructor) :

+ toString ( ) : String

Textbook

- cost : double
- title: String

+ Textbook ( )

+ Textbook (t : String, c : double)

+ setCost (c : double) : void

+ setTitle (t : String) : void

+ getCost ( ) : double

+ getTitle ( ) : String+ toString ( ) : String

Instructor

- lastName : String
- firstName : String

+ Instructor ( ) :

+ Instructor (ln : String, fn : String) :

+ setLastName (ln : String) : void

+ setFirstName (fn : String) : void

+ getFirstName ( ) : String

+ getLastName ( ) : String

+ toString ( ) : String

```java
public class Instructor {



        // Constructors
        public Instructor() { }

        public Instructor(String ln, String fn) {
                this.lastName = ln;
                this.firstName = fn;
        }

        // Setters









        // Getters
        public String getLastName() {
                return lastName;
        }

        public String getFirstName() {
                return firstName;
        }


        public String toString() {
                return firstName + “ ” + lastName;
        }
}
```

```
public class Textbook {



        // Constructors
        public Textbook () { }

        public Textbook(String t, double c) {
                this.title = t;
                this.cost = c;
        }

        // Setters








        // Getters
        public String getTitle () {
                return title;
        }

        public double getCost() {
                return cost;
        }

```

```
        public String toString() {
                return title + “ ($” + cost + “)”;
        }
}
```

```
public class Course {
        // Attributes




        public Course() { }

        public Course(                                           ) {





        }

        public String toString() {
                return “Course Name: ” + name + “\nInstructor: ” + inst
                         + “\nTextbook: ” + text;
        }
}
```

```
public class CourseDemo {

   public static void main(String[] args) {
      Instructor myInst = new Instructor(“Porter”, “Jeremy”);
      Textbook myText = new Textbook(“Java”, 125.5);

      Course myCourse = new Course(“CSCI1101”, myText, myInst);

      System.out.println(myCourse);



   }

}
```

Aggregation of classes

What would happen if we instead used this as our constructor?

```
public Course(String n, Textbook tb, Instructor i) {
            name = n;
            inst = new Instructor(i.getLastName(), i.getFirstName());
            text = new Textbook(tb.getTitle(), tb.getCost());
}
```

Aggregation of classes – shallow vs deep copy

- The first constructor uses a *shallow copy*
  - This is simply a copy of the Object *reference*
  - Any changes made are done on a single instance of the Object
  - Advantages: saves memory, simpler to user, easier to code
  - Disadvantages: decreased security, less control over data

- The second (new) constructor uses a *deep copy*
  - This individually copies the relevant attributes into a *new* Object
  - This results in a second instance that may be changed separately from the original
  - Advantages: more control over data (only one reference exists)
  - Disadvantages: uses more memory, may not always be necessary

Aggregation of classes – copying

- Choosing shallow vs deep impacts the **`copy`** method for any class using aggregation

```
public Course copy() {
    Course cc;
    // Shallow copy!!
    cc = new Course(name, text, inst);


    return cc;
}
```

Aggregation of classes – getter methods

- We encounter a similar issue with getter methods

```
public Instructor getInstructor() {
      // Shallow copy!!
      return this.inst;



}
```

Aggregation of classes – walk-away example

Add to the Circle and Point classes so that a Circle can determine whether or not a Point is contained within its boundary.