

# CS 213 – Software Methodology

*Sesh Venugopal*

Inheritance vs Delegation

# Why Inheritance?

- The design aspect of inheritance is to model the “IS A” relationship between objects
- Examples:
  - Car **is a** MotorVehicle (*every car is a motor vehicle*)
  - Motorcycle **is a** MotorVehicle (*every motorcycle is a motor vehicle*)
  - ColoredPoint **is a** Point (*every colored point is a point*)
  - Zebra **is a(n)** animal (*every zebra is an animal*)
- Inheritance then allows class on the right hand side of the **is a** to “hand down” its code to the class on the left hand side
- The RHS class (e.g. MotorVehicle) is the superclass (base class) and LHS class (e.g. Car) is the subclass

# Why Inheritance?

- Handing down code results in reuse: there is only one copy to manage instead of two or more
- Creating an instance of Car does not automatically also create an instance of MotorVehicle. Inheritance does not mean that a subclass object (Car) has a superclass object (MotorVehicle) contained inside it. (A Car does not contain a MotorVehicle)
- Which of these “**IS A**” relationships are accurate?:
  - Square is a rectangle
  - Cube is a square
  - Student is a Person
  - Employee is a Person

## Square is a Rectangle. But does inheritance work?

```
public class Rectangle {  
    int x, y, w, h;  
  
    . . .  
    public int area() {  
        . . .  
    }  
    public int perimeter() {  
        . . .  
    }  
    public void setSize(int w,  
                        int h) {  
        . . .  
    }  
}
```

```
public class Square extends  
    Rectangle {  
    . . .  
    public void setSize(int w,  
                        int h) {  
        // disallow stretching when  
        // w is not equal to h  
        . . .  
    }  
}
```

# Square is a Rectangle. But does inheritance work?

There are two important reasons why the `Square extends Rectangle` DESIGN idea **does not** work:

- The `Square` class **does not provide any new functionality**
- The inherited method `setSize` is **overridden in a way that *restricts the set size behavior***

1. For inheritance to be used correctly, the subclass must provide **ALL** the functionality (methods) of the superclass, AND more .

2. Also, if the subclass overrides an inherited method of the superclass, the overriding method must use ALL of the inherited method's implementation, and ADD more.

# Rectangle – Square Design Alternatives

**Alternative 1: Code only a Rectangle class, and have it tell whether it is a square or not**

```
public class Rectangle {
    public static final int DEFAULT_X = 100;
    public static final int DEFAULT_Y = 100;

    private int x=DEFAULT_X, y=DEFAULT_Y, w, h;
    private boolean isSquare = false;

    public Rectangle(int width, int height) {
        w = width; h = height;
    }

    // to be used if square is needed
    public Rectangle(int side) {
        w = h = side; isSquare = true;
    }

    public boolean isSquare() {
        return isSquare;
    }

    public void setSize(int w, int h) {
        if (isSquare && w != h) {
            throw new IllegalArgumentException(
                "w must be equal to h for square");
        }
        this.w = w; this.h = h;
    }
}
```

What if client “forgets” to use this constructor, and uses the other constructor correctly (by setting  $w = h$ ), intending a square? Rectangle would not know of the intention, leading to problems in usage.

# Rectangle – Square Design Alternatives

**Alternative 2: Code a Rectangle class, and use delegation (composition) to have Square point to it**

```
public class Rectangle {
    public static final
        int DEFAULT_X = 100;
    public static final
        int DEFAULT_Y = 100;

    private int x=DEFAULT_X,
        y=DEFAULT_Y, w, h;

    public Rectangle(int width,
        int height) {
        w = width; h = height;
    }

    public void setSize(int w, int h) {
        this.w = w; this.h = h;
    }
    ...
}
```

Square is composed of  
Rectangle instance,  
i.e. Square has a  
Rectangle

```
public class Square {
    private Rectangle rect;

    public Square(int side) {
        rect = new Rectangle(side,side);
    }

    public void setSize(int side) {
        rect.setSize(side,side);
    }
    . . .
}
```

Rectangle-specific functionality is  
delegated to Rectangle object

Delegation/composition is a viable alternative to inheritance for code reuse

# Employee is a Person. Student is a Person. Issues in Implementing with Inheritance

*This example from “Object-Oriented Design using Java” by Dale Skrien*

```
public class Person {  
    String name, address;  
    . . .  
}
```

```
public class Student  
    extends Person {  
    float gpa;  
    . . .  
}
```

```
public class Employee  
    extends Person {  
    float salary;  
    . . .  
}
```

Scenario 1: A student graduates and becomes an employee of the university

**Solution A: Replace Student object for this person with Employee object**

- Data from Student object (e.g. transcripts) may need to be preserved, but there is no place for this in Employee object



# Employee is a Person. Student is a Person.

## Issues in Implementing with Inheritance

*This example from “Object-Oriented Design using Java” by Dale Skrien*

```
public class Person {  
    String name, address;  
    . . .  
}
```

```
public class Student  
    extends Person {  
    float gpa;  
    . . .  
}
```

```
public class Employee  
    extends Person {  
    float salary;  
    . . .  
}
```

Scenario 1: A student graduates and becomes an employee of the university

**Solution B: Keep *inactive* Student object for this person, and create an *active* Employee object**

- All Person-level data is duplicated in both objects (wasted space)
- Whenever a change is made to Person-level data in one (e.g. address), it must also be made in the other (drawback: tracking for synchronization)

# Employee is a Person. Student is a Person.

## Issues in Implementing with Inheritance

*This example from “Object-Oriented Design using Java” by Dale Skrien*

```
public class Person {  
    String name, address;  
    . . .  
}
```

```
public class Student  
    extends Person {  
    float gpa;  
    . . .  
}
```

```
public class Employee  
    extends Person {  
    float salary;  
    . . .  
}
```

Scenario 2: A Student is also an employee at the same time

**Solution:** Keep *active* Student object for this person, as well as *active* Employee object

- All Person-level data is duplicated in both objects (wasted space)
- Whenever a change is made to Person-level data in one (e.g. address), it must also be made in the other (drawback: tracking for synchronization)

# Employee is a Person. Student is a Person. Issues in Implementing with Inheritance

---

## OBSERVATION:

Employee and Student  
are temporary ROLES  
played by Person

In situations like this, inheritance is not a good design.

Instead, composition/delegation is a better design alternative.

# Employee is a Person. Student is a Person.

## ROLES: Composition/Delegation

```
public class Person {  
    private String name;  
    private String address;  
    public String getAddress() {  
        return address;  
    }  
    . . .  
}
```

```
public class Student {  
    Student refers to  
    private Person me; a Person instance,  
                        i.e. Student has a Person  
  
    private Transcript myTranscript;  
    public String getAddress() {  
        return me.getAddress();  
    }  
    public float getGPA() {  
        . . .  
    }  
    . . .  
}
```

*Person-specific functionality is delegated to Person object*

**Employee** can similarly refer to **Person**, delegating **Person**-specific tasks to the referenced **Person** object

If a student graduates and then becomes an employee, both inactive **Student** object and active **Employee** object can refer to *same* **Person** object. Thus, multiple roles played by the same person at the same, or different times, can be handled well by referencing/delegation.

# Delegation for Roles

**If class B (e.g. Student) models a temporary role played by class A (e.g. Person), then B should not be a subclass of A.**

**Instead, B should reference A and use delegation to do A-specific stuff.**