INFO1113 Object-Oriented Programming

Week 5A: Class Inheritance
Reusing variables, methods and classes

Slides developed by Tyson Thomas

Copyright Warning

Topics

Inheritance

Inheritance is a significant concept of **OOP**. Allowing reusability and changes to inherited methods between different types in a **hierarchy**.

What does inheritance offer?

- Attribute and method reusability
- Defining sub-class methods
- Overriding inherited methods
- Type information

Inheritance

How does it work?

We will be introducing a new keyword today called **`extends`**, this keyword allows the class to inherit from another class.

Syntax:

```
    [public] class ClassName extends SuperClassName
```

Refer to Chapter 8.1, page 624-629 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Inheritance

How does it work?

We will be introducing a new keyword today called **`extends`**, this keyword allows the class to inherit from another class.

Syntax:

```
    [public] class ClassName extends SuperClassName
```

Class definition, we specify the access modifier

Inheritance

How does it work?

We will be introducing a new keyword today called **`extends`**, this keyword allows the class to inherit from another class.

Syntax:

```
[public] class ClassName extends SuperClassName
```

Class definition, we specify the access modifier

ClassName (What you are going to name the class)

Refer to Chapter 8.1, page 624-629 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Inheritance

How does it work?

We will be introducing a new keyword today called **`extends`**, this keyword allows the class to inherit from another class.

Syntax:

```
[public] class ClassName extends SuperClassName
```

[public]: Class definition, we specify the access modifier

ClassName: **ClassName** (What you are going to name the class)

extends: We are inheriting from the following class. It is seen as an **extension** of the super class.

Refer to Chapter 8.1, page 624-629 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Inheritance

How does it work?

We will be introducing a new keyword today called **`extends`**, this keyword allows the class to inherit from another class.

Syntax:

```
[public] class ClassName extends SuperClassName
```

Class definition, we specify the access modifier

ClassName (What you are going to name the class)

We are inheriting from the following class. It is seen as an **extension** of the super class.

The class we are inheriting from. It will inherit any **protected** or **public** methods or attributes

Refer to Chapter 8.1, page 624-629 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

How it looks

Part of our class declaration line allows for us to define what class we want to **extend** from

```
public class Dog extends Animal
```

Once defined, **Dog** type can also be used as a **Animal** type as it is just an extension of such type.

Encapsulation

We have used the **public** and **private** access modifier but we will now use the **protected** access modifier.

What does **protected** mean?

Like **private** it will not be accessible to other classes but now with the exception **inherited classes.**

- Is only accessible within the class
- Attributes and methods will be accessible by all subclass
- Allows single definition of an attribute instead of multiple

Refer to Chapter 8.1, page 632 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;


    public double volume() {
        return height*width*depth;
    }

}
```

```
public class GlassBottle extends Bottle
{







    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Inheritance

So let’s take a look how inheritance works between two classes.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public double volume() {
        return height*width*depth;
    }

}
```

Subclass will have access to any **protected** and **public** methods.

```
public class GlassBottle extends Bottle
{

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Inheritance

So let’s take a look how inheritance works between two classes.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;


    public double volume() {
        return height*width*depth;
    }

}
```

Protected like **private** but allows subclass to inherit the property.

```
public class GlassBottle extends Bottle
{






    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;


    public double volume() {
        return height*width*depth;
    }

}
```

```
public class GlassBottle extends Bottle
{

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled
    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

All properties from the **super** class are **inherited** by the **subclass.** As if they were defined in the class itself.

Inheritance

So let’s take a look how inheritance works between two classes.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;


    public double volume() {
        return height*width*depth;
    }

}
```

```
public class GlassBottle extends Bottle
{

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled
    private boolean shattered = false;

    public void shatter() {
        System.out.println(“We lost
        “ + litresFilled + “Litres”);
        litresFilled = 0;
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Able to refer to the attributes within the **subtypes** own methods.

What about constructors?

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```java
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {

    }

    public double volume() {
        return height*width*depth;
    }

}
```

```java
public class GlassBottle extends Bottle {



    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
```

```
public class GlassBottle extends Bottle {



    private boolean shattered = false;

    public void shatter() {
        shattered = true;
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {

    GlassBottle b = new GlassBottle();
    System.out.println(b.isBroken());
    System.out.println(b.name);
}
```

Refer to Chapter 8.1, page 637 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {

    }

    public double volume() {
        return height*width*depth;
    }

}
```

```
public class GlassBottle extends Bottle {



    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

However! Nothing was initialised, so all we get are default values

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
        this.depth = 10d;
        this.litresFilled = 0;
    }

    public double volume() {
        return height*width*depth;
    }
}
```

```
public class GlassBottle extends Bottle {

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Providing some values we can inspect the previous code segment

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
```

```
public class GlassBottle extends Bottle {



    private boolean shattered = false;

    public void shatter() {
        shattered = true;
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.isBroken());
    System.out.println(b.name);
}
```

```
> java MyProgram
false
Basic Bottle
<program end>
```

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
```

```
public class GlassBottle extends Bottle {



    private boolean shattered = false;

    public void shatter() {
        shattered = true;
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.isBroken());
    System.out.println(b.name());
}
```

```
> java MyProgram
false
Basic Bottle
<program end>
```

We can see that even though we seemingly used the **GlassBottle** constructor.

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
        this.depth = 10d;
        this.litresFilled = 0;
    }


    public double volume() {
        return height*width*depth;
    }
}
```

```
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

What if we were to define a constructor in the subclass?

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```java
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
```

```java
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

```java
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.volume());
    System.out.println(b.name);
}
```

```
> java MyProgram
1000.0
Glass Bottle
<program end>
```

We can see that we called the **GlassBottle** constructor and it set the **name** to **Glass Bottle**.

Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
```

```
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.volume());
    System.out.println(b.name);
}
```

```
> java MyProgram
1000.0
Glass Bottle
<program end>
```

Hang on! If we called GlassBottle() how is volume returning 1000.0?

Let’s try something

Inheritance

```java
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
        this.height = height;
        this.depth = depth;
        this.litresFilled = 0;
    }


    public double volume() {
        return height*width*depth;
    }
}
```

```java
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Inheritance

```java
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
        this.height = height;
        this.depth = depth;
        this.litresFilled = 0;
    }

    public double volume() {
        return height*width*depth;
    }
}
```

What if we were to add a constructor with parameters?

```java
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Inheritance

```java
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

```java
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods.

```java
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.volume());
    System.out.println(b.name);
}
```

How would the GlassBottle constructor be able to invoke the super constructor?

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class
Bottle cannot be applied to given types;
            public GlassBottle() {
                        ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Inheritance

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

```
public class GlassBottle extends Bottle {

    public GlassBottle() {
        super("", 0, 0, 0);
        this.name = "Glass Bottle";

    }

    private boolean shattered = false;
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```
public static void main(String[] args) {
    GlassBo
    System.
    System.out.println(b.name);
}
```

We are able to use the **super** keyword to invoke the **parent** constructor.

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class
Bottle cannot be applied to given types;
            public GlassBottle() {
                        ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Inheritance

```java
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

Refers to **Bottle** constructor

```java
public class GlassBottle extends Bottle {

    public GlassBottle() {
        super("", 0, 0, 0);
        this.name = "Glass Bottle";

    }

    private boolean shattered = false;
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```java
public stati
    GlassBo
    System.
    System.out.println(b.name);
}
```

We are able to use the **super** keyword to invoke the **parent** constructor.

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class
Bottle cannot be applied to given types;
            public GlassBottle() {
                        ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Inheritance

```java
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

Refers to **Bottle** constructor

```java
public class GlassBottle extends Bottle {

    public GlassBottle(String name, double
        width,double height,double depth){
        super(name, width, height, depth);

    }

    private boolean shattered = false;
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```java
public static void main(String[] args) {
    GlassBo
    System.
    System.out.println(b.name);
}
```

We could match the constructor of the parent type.

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class
Bottle cannot be applied to given types;
            public GlassBottle() {
                        ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Relationship

There are two types of relationships we will look at when it comes to inheritance.

- **Is-a** relationship (Extension)
- **Has-a** relationship (Composition)

In regards to class inheritance we are considering the **Is-a** relationship how a class is an **extension** of another class but is also the other class.

Relationship

We have to be very **certain** with inheritance that any class that inherits from another **is a** type of that class. There should be clear reasoning that the types satisfy the relationship.

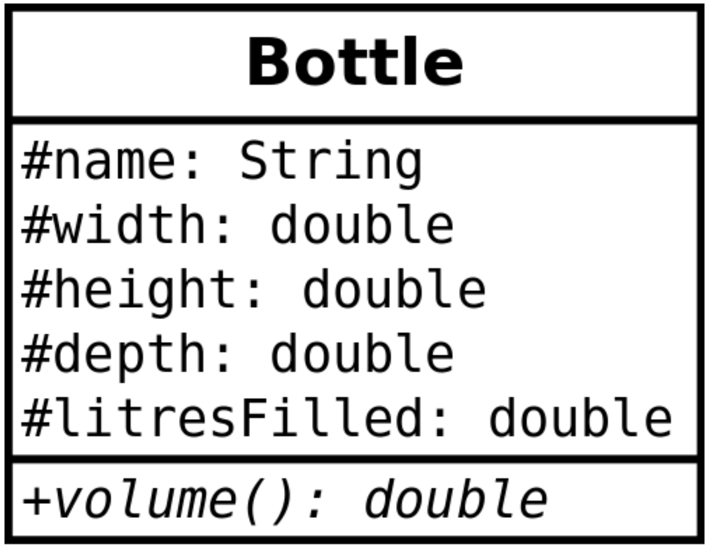
There needs to be clear reasoning to extending the super class.

Some instances where it makes sense:

- Super class is **Cat** and subclasses are **Panther, Lion, Tiger**
- Super class is **Controller** and subclasses are **Gamepad, Joystick, Powerglove**
- Super class is **Media** and subclasses are **DVD, Book, Image**

UML Generalization

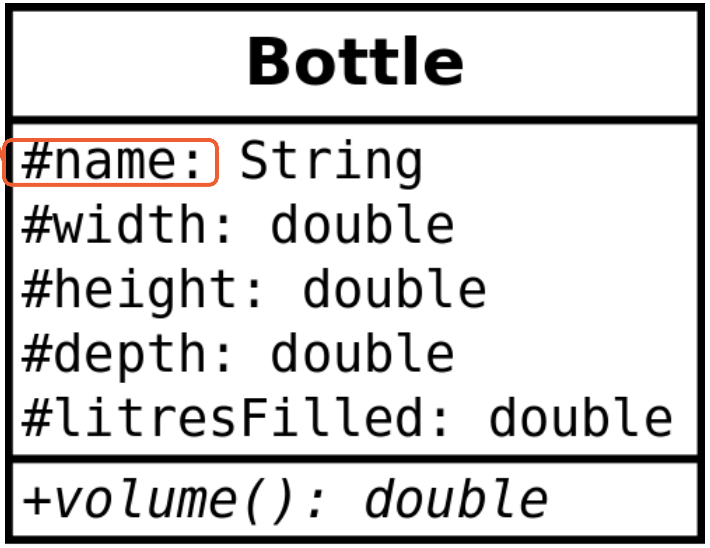
Let’s examine the following UML Diagram.



UML Generalization

Let's examine the following UML Diagram.

Protected is defined using the # symbol and will be a variable that is inherited.

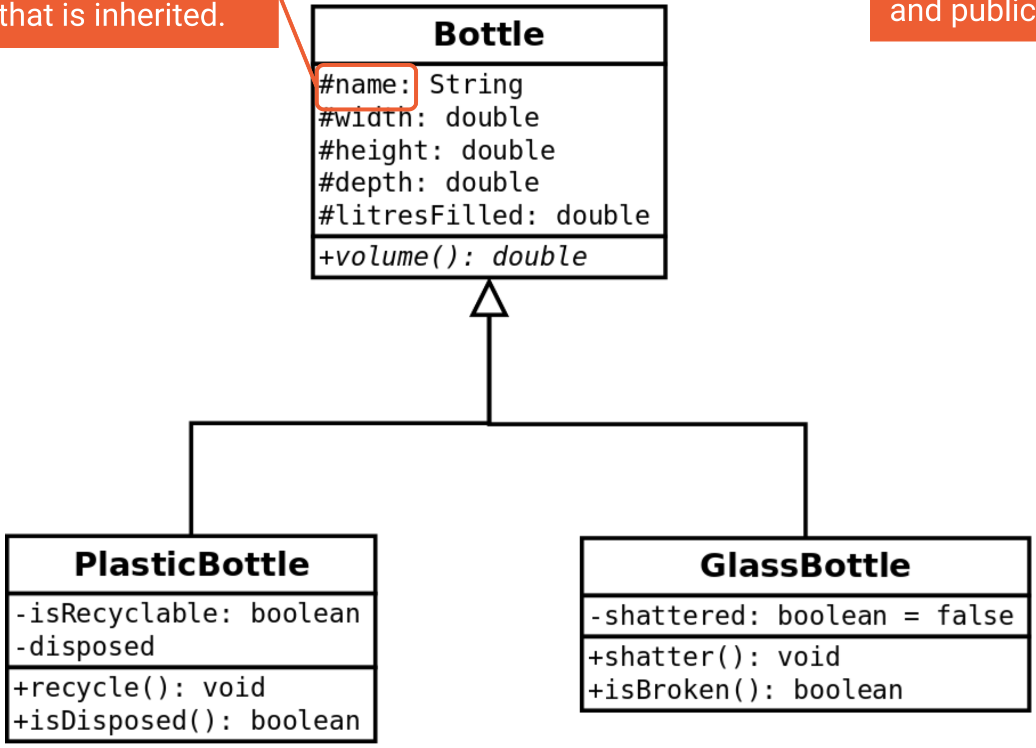


UML Generalization

Let's examine the following UML Diagram.

Protected is defined using the # symbol and will be a variable that is inherited.

When other classes inherit from the superclass they will get the protected and public attributes, methods



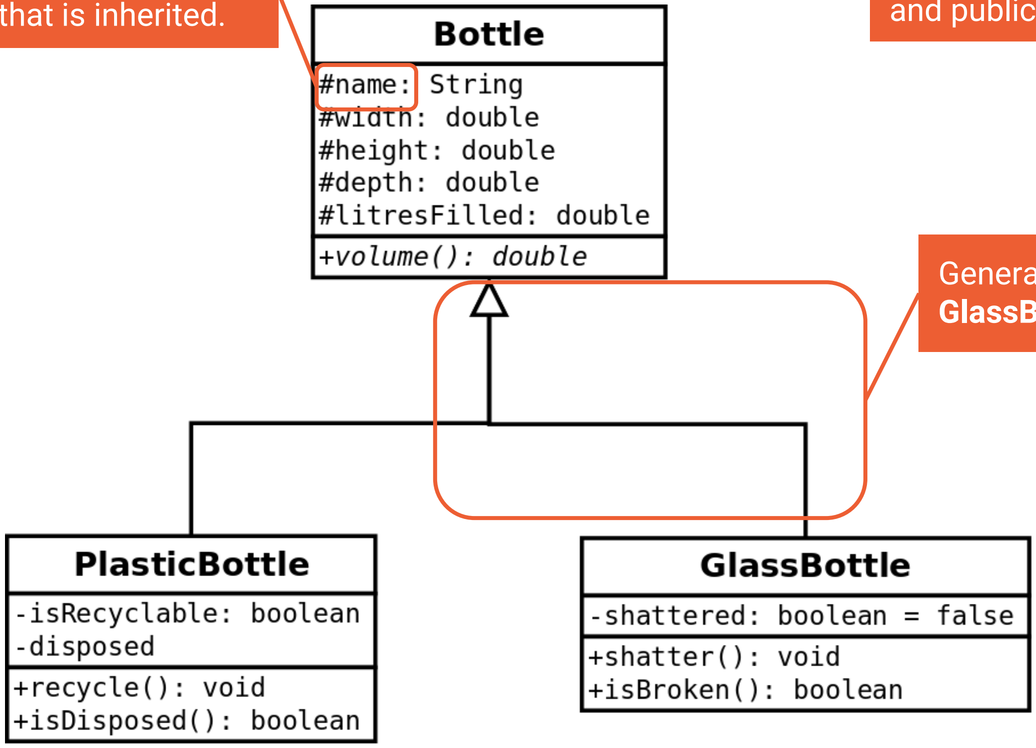
Refer to Chapter 8.1, page 635 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

UML Generalization

Let's examine the following UML Diagram.

Protected is defined using the # symbol and will be a variable that is inherited.

When other classes inherit from the superclass they will get the protected and public fields



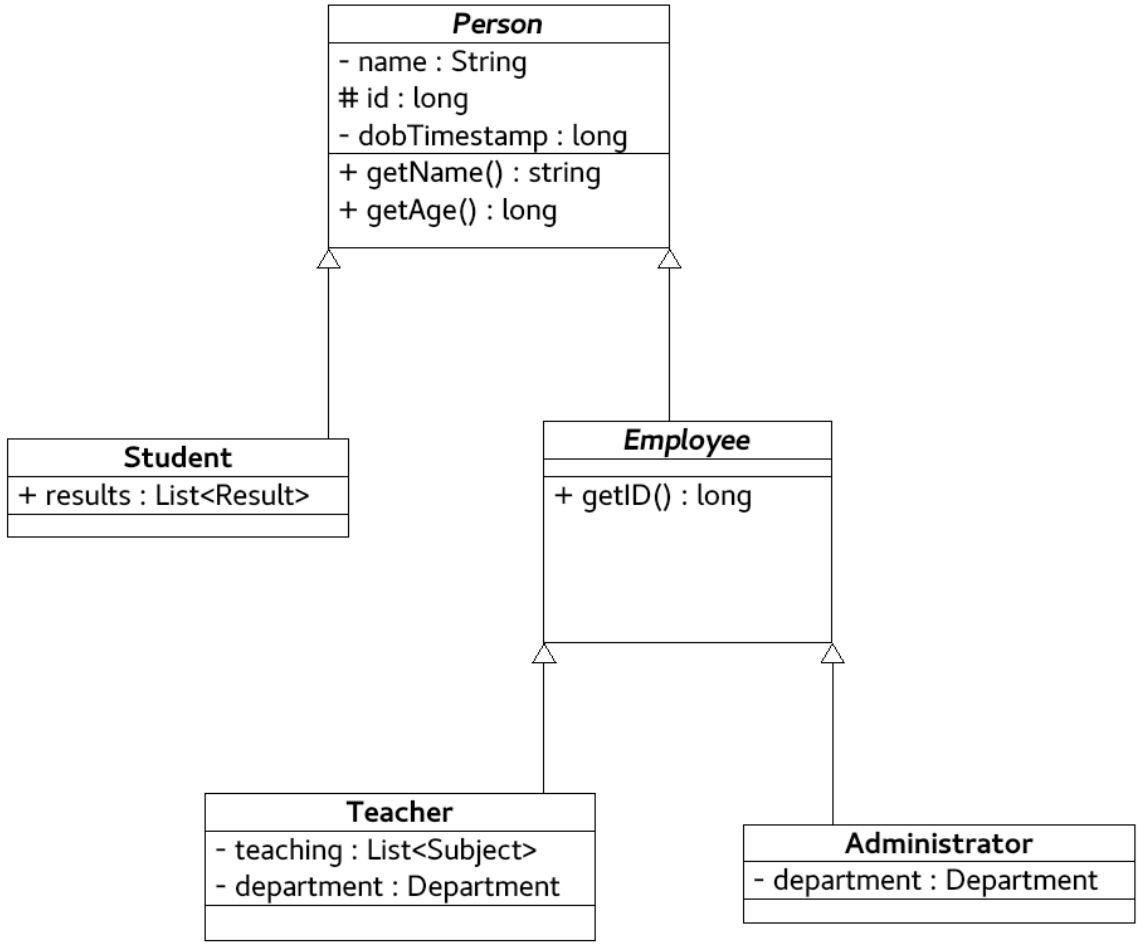
Generalization link, shows that **GlassBottle** is a subclass of **Bottle.**

Refer to Chapter 8.1, page 635 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Can inheritance be misused?

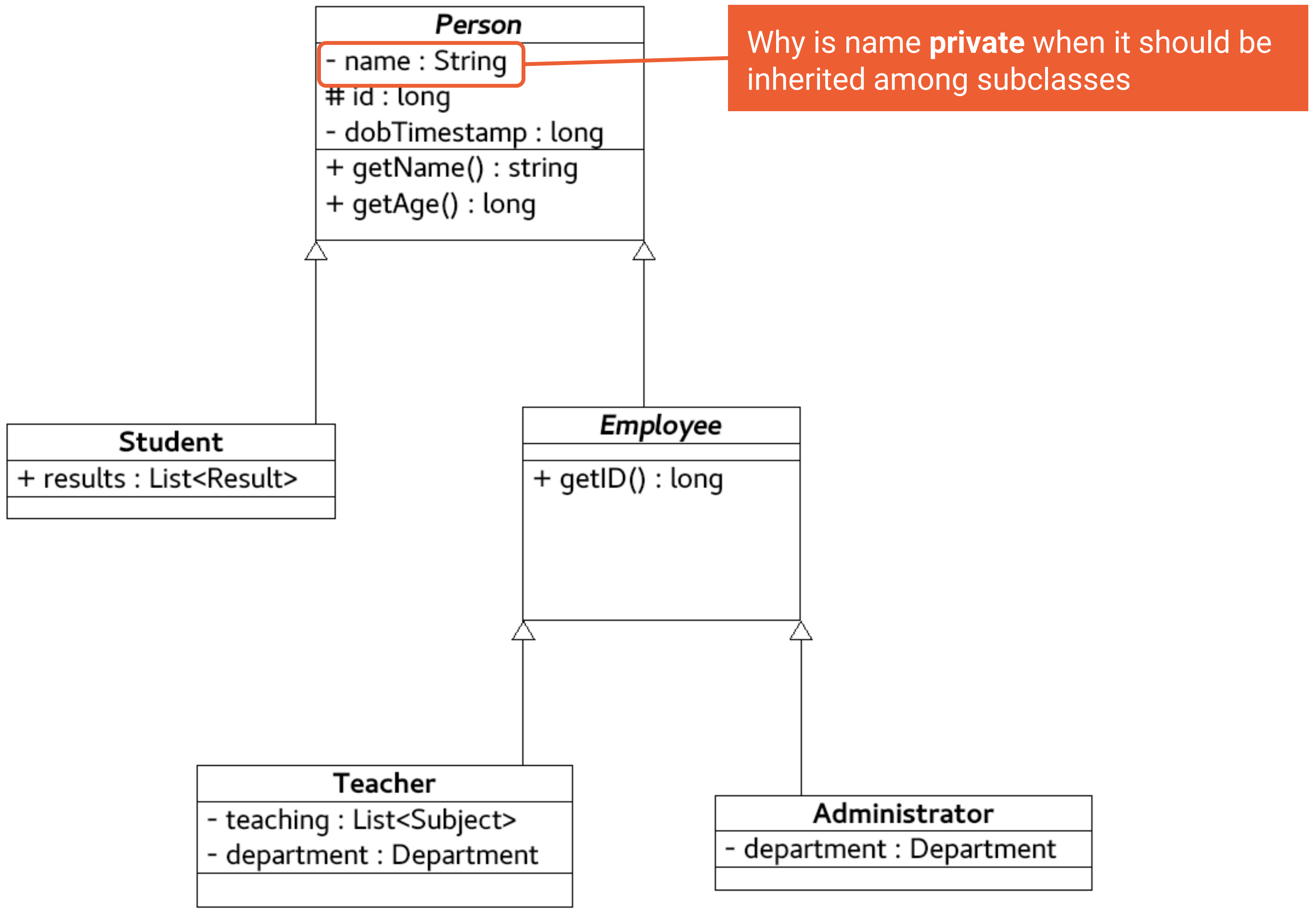
Where inheritance fails

Yes! Take a look at the following UML diagram.



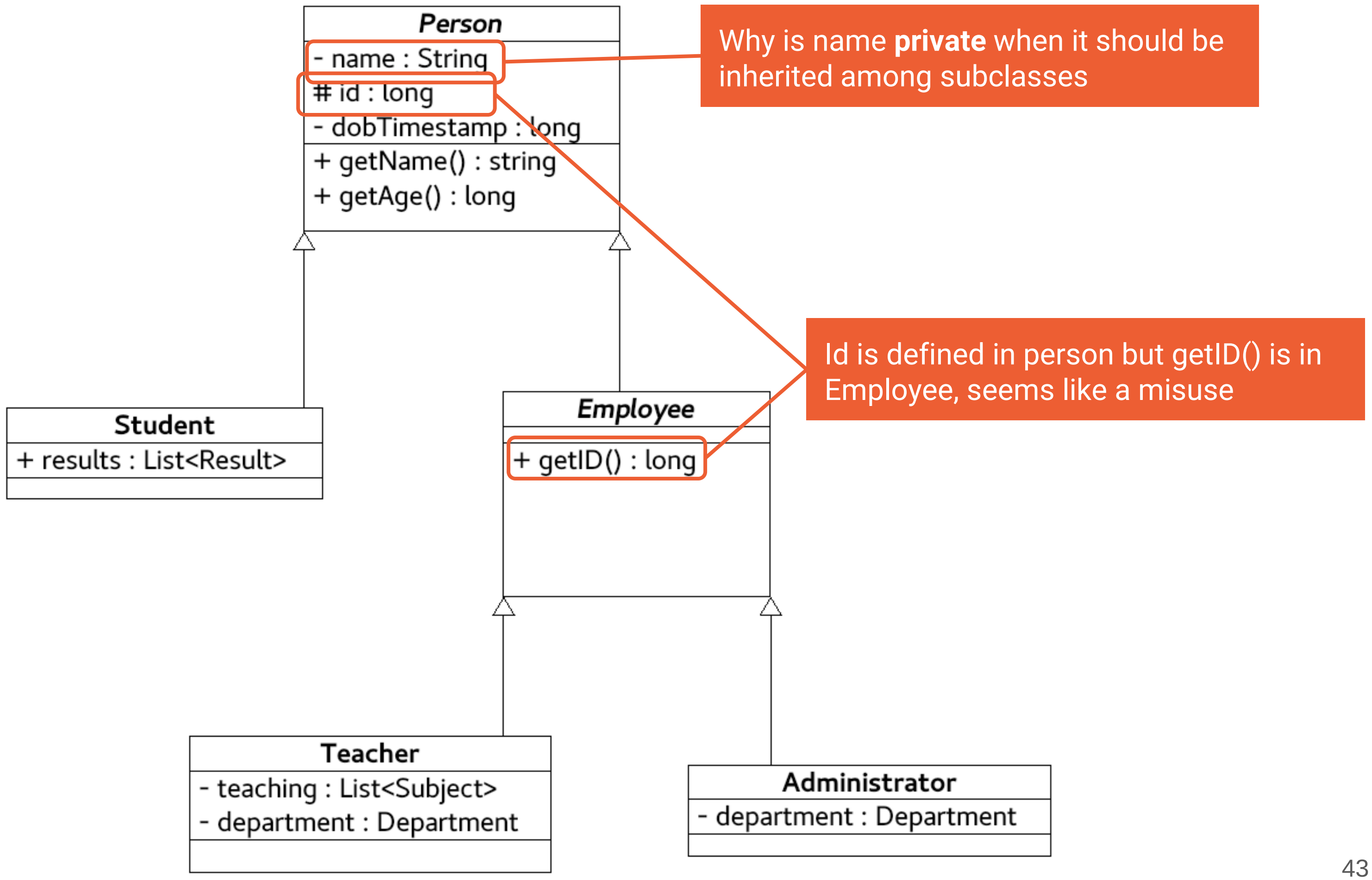
Where inheritance fails

Yes! Take a look at the following UML diagram.



Where inheritance fails

Yes! Take a look at the following UML diagram.

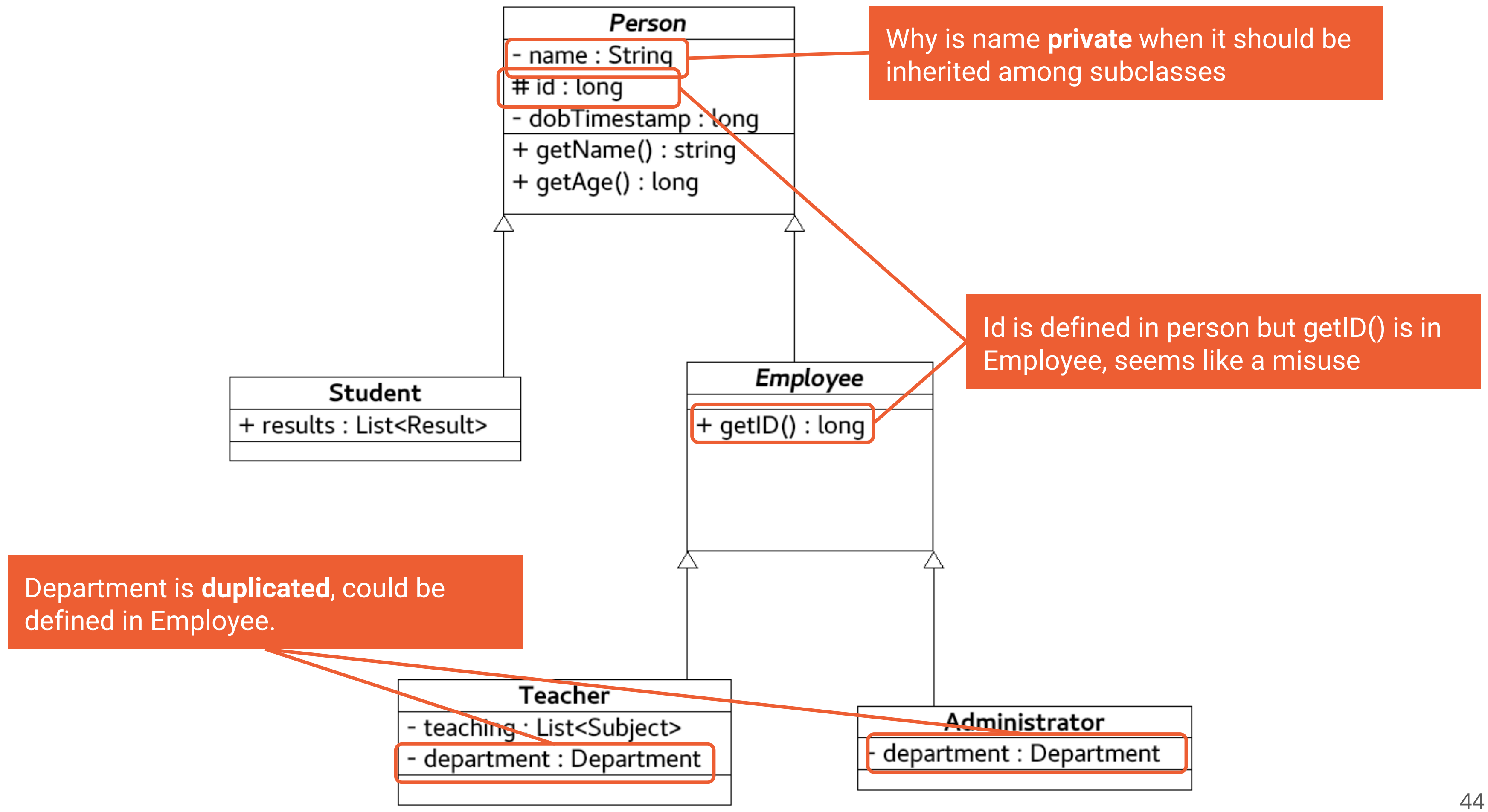


Why is name **private** when it should be inherited among subclasses

Id is defined in person but getID() is in Employee, seems like a misuse

Where inheritance fails

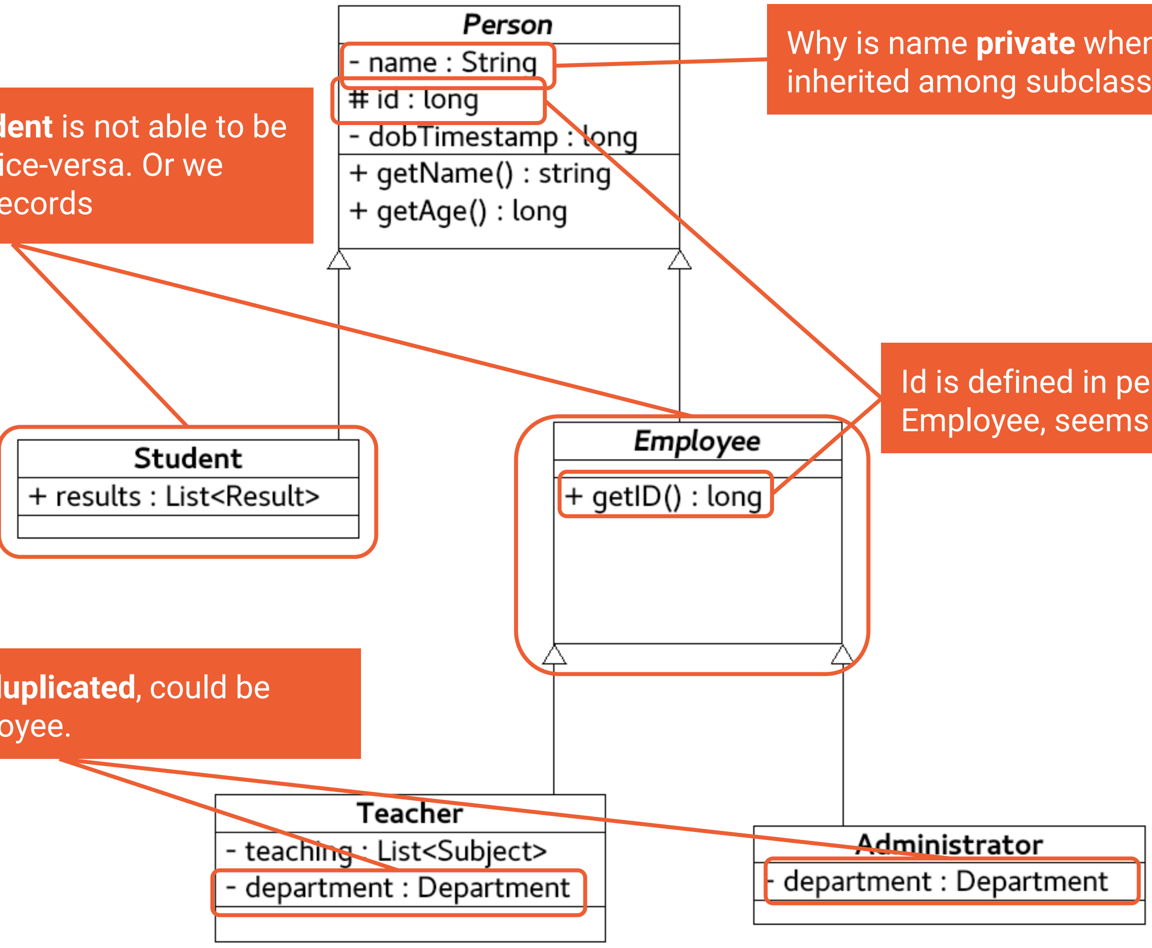
Yes! Take a look at the following UML diagram.



Person
- name : String
# id : long
- dobTimestamp : long
+ getName() : string
+ getAge() : long
Why is name **private** when it should be inherited among subclasses
Id is defined in person but getID() is in Employee, seems like a misuse
Student
+ results : List<Result>
Employee
+ getID() : long
Department is **duplicated**, could be defined in Employee.
Teacher
- teaching : List<Subject>
- department : Department
Administrator
- department : Department

Where inheritance fails

Yes! Take a look at the following UML diagram.



Why is name **private** when it should be inherited among subclasses

By this logic, a **Student** is not able to be an **Employee** and vice-versa. Or we need to store two records

Id is defined in person but getID() is in Employee, seems like a misuse

Department is **duplicated**, could be defined in Employee.

Super class and subclass

Some other factors to consider:

- Superclass does not know about its subclasses
- **Private** is not inherited, only **protected** and **public**
- Ensure when you use inheritance you are certain it will satisfy an **is-a** relationship
- You can only inherit from **1 class**.
- Within **UML** inheritance is shown as a **Generalization**.
- You **cannot** use subclass properties through a superclass binding.
- Subclasses cannot be constructed using a superclass constructor (Subclass a = new Superclass();)

See you next time!