Composition

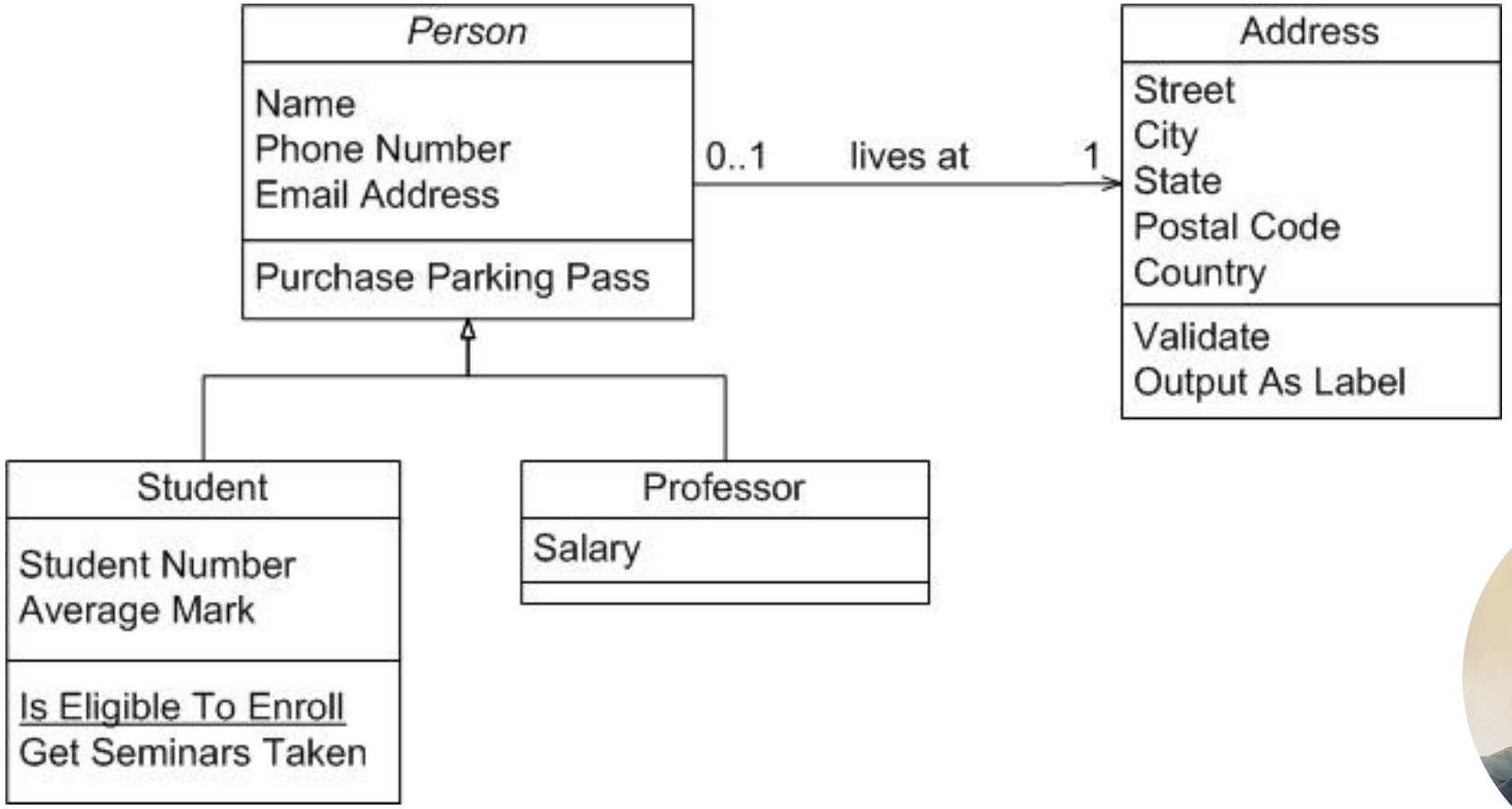
Summary

- You will know about polymorphism limits
- You will learn about compositions as an alternative to polymorphism
- You will know to apply composition

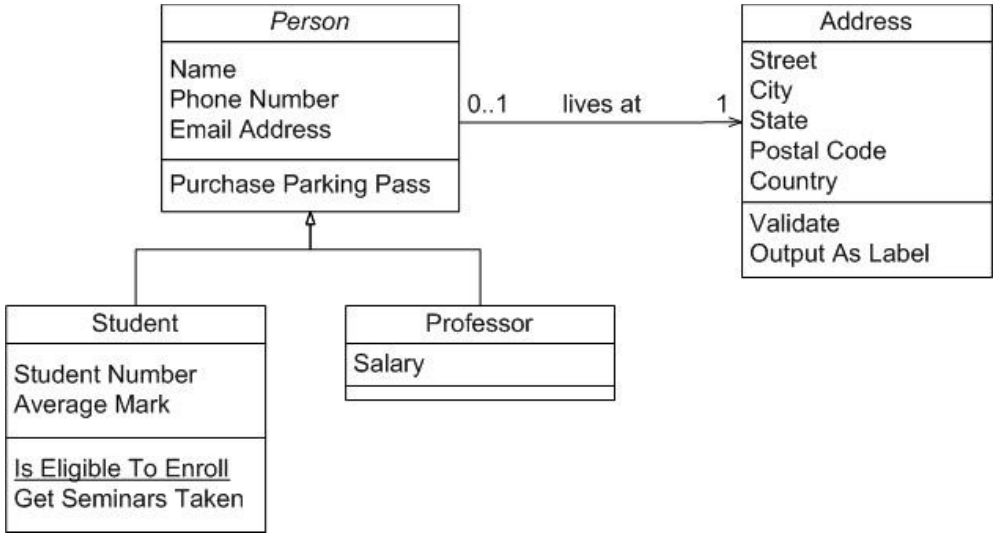
Polymorphism is not the only instrument we can apply

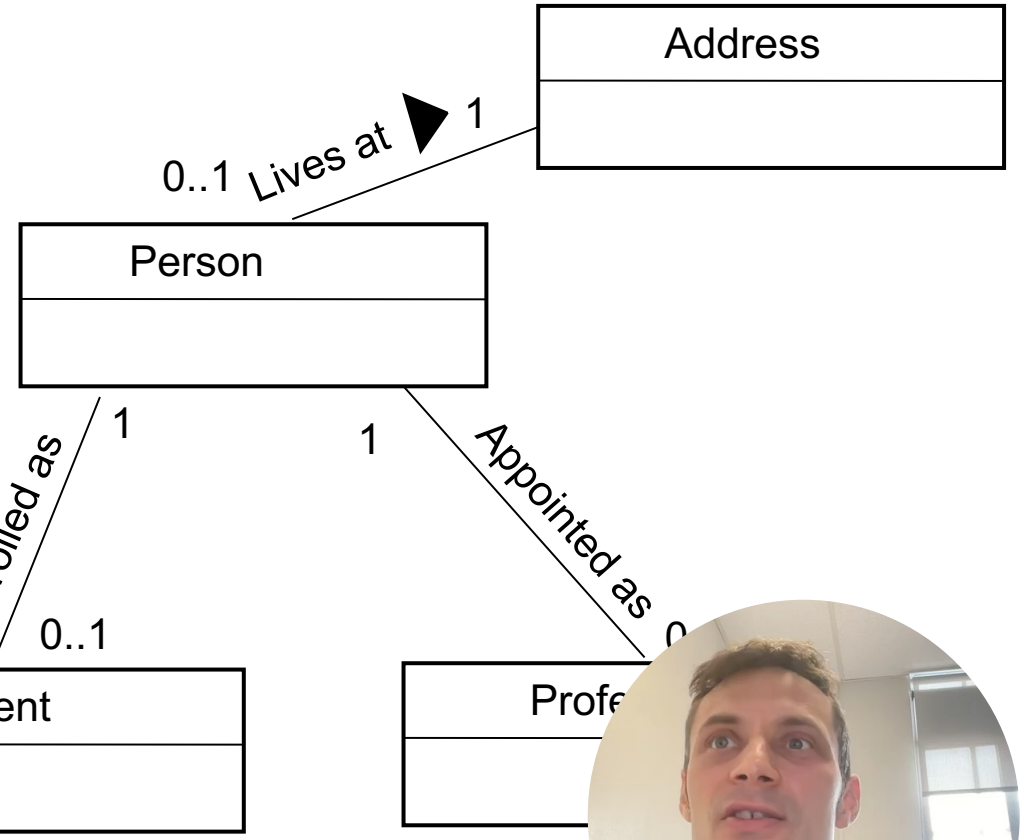
Composition is sometimes better

Text book example



Generalization Exclusivity vs Composition





http://agilemodeling.com/artifacts/classDiagram.htm

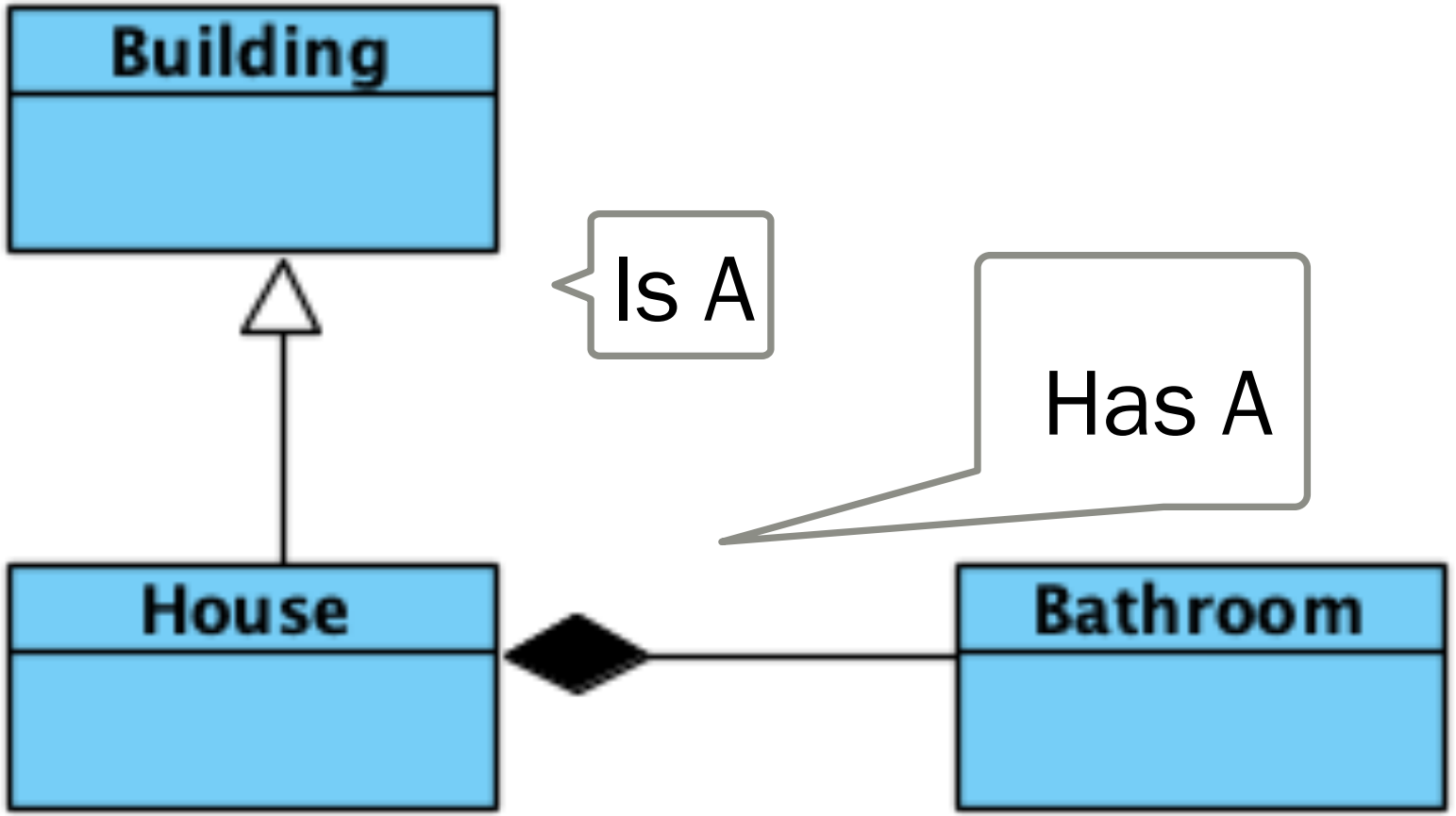
We have this lecture on composition

Class composition and inheritance

- Reminder:

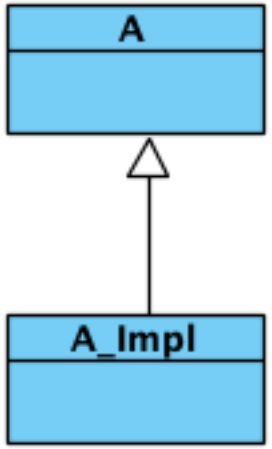
- Two ways to define one class in terms of the other
  - *Composition: one class composed of other classes*
  - *Inheritance: one class is a subclass of another class*

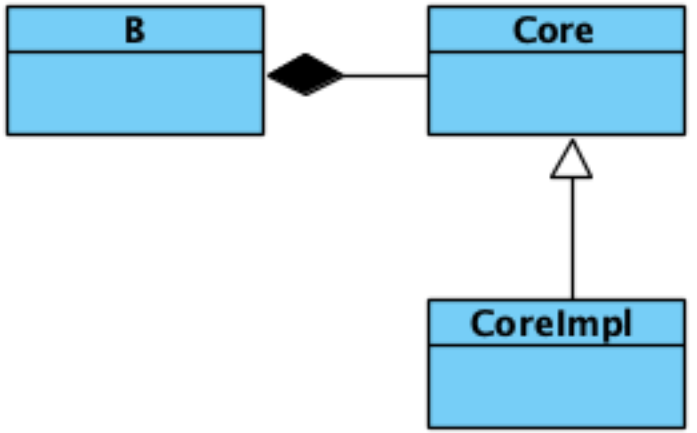
Composition vs inheritance



Composition vs inheritance

- Achieve polymorphic behavior by inheritance
- Achieve code reuse by their composition





Composition vs inheritance

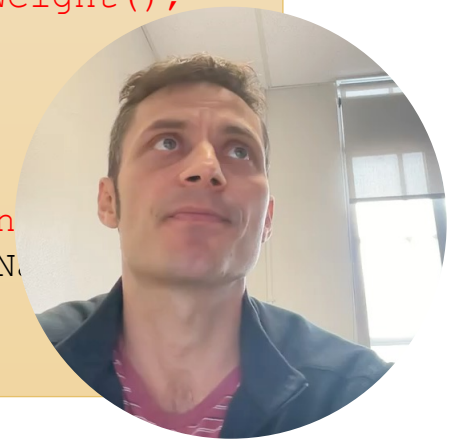
- Achieve polymorphic behavior by inheritance
- Achieve code reuse by their composition
  - Object composition requires that the objects being composed have well-defined interfaces.
  - This style of reuse is called black-box reuse, because no internal details of objects are visible.

Composition

- **Combining simple** types to make more complex ones.
- Let us make few twists here:

```
public class Dog {
        public float getWeight();
        public float getHeight();
}
public class GermanDog extends Dog {
        public Date getBreedSince();
}
public class Doberman extends GermanDog {
        public String getNativeName();
}
```

```
public class Dog {
        public float getWeight();
        public float getHeight();
}
public class GermanDog {
        private Dog dog;
        public Date getBreedSince();
        public float getWeight() {
                return dog.getWeight();
        }

}
public class Doberman {
        private GermanDog german
        public String getNativeN
}
```



Composition

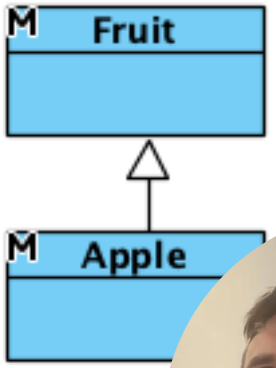
- No common interface or a parent, freedom of combination
- Similar to inheritance but it is all about relationship between classes!
- Two orthogonal directions
  - *inheritance*
  - *composition*
- Which one is better?
  - It depends -> best practice patterns for distinct situations
  - The best is to combine them to solve your problem!

* Running example from Bill Venners in JavaWorld, Oct[illegible]98

Composition vs Inheritance example

- Class extension
  - *class Apple is related to class Fruit by inheritance, because Apple extends Fruit*
  - *in this example, Fruit is the superclass and Apple is the subclass*

```
class Fruit {

        //...
}
class Apple extends Fruit {

         //...
}
```

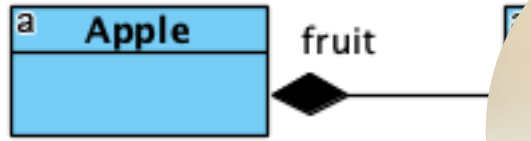


Inheritance re

Composition vs Inheritance example

- Class composition
  - using instance variables that are references to other objects
  - class *Apple* is related to class *Fruit* by composition, because *Apple* has an instance variable that holds a reference to a *Fruit* object.
  - In this example, *Apple* is the *frontend class* and *Fruit* is the *backend class*.
  - In a composition relationship, the frontend class holds a reference in one of its instance variables to a backend class.

```
class Fruit {

    //...
}
class Apple {

   private Fruit fruit = new Fruit();
   //...
}
```

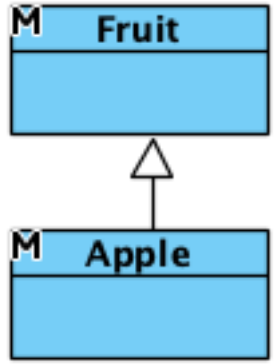


Composition rel

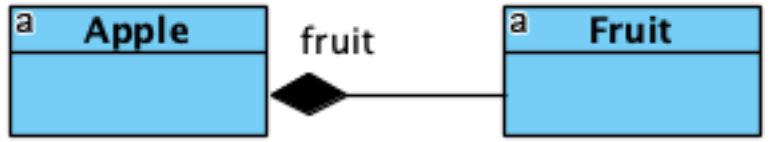
Compare!

- Apple + Fruit - how do we connect them?

Think of a wrapper



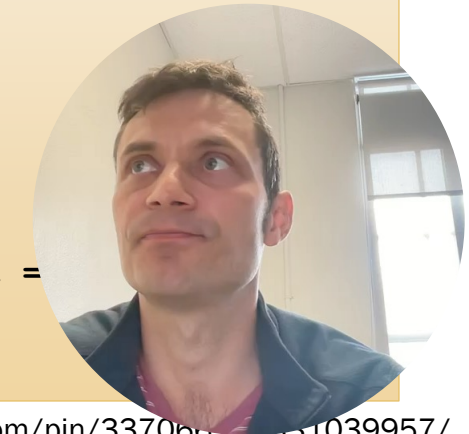
Inheritance connection



Composition relationship

```
class Fruit {

        //...
}
class Apple extends Fruit {

         //...
}
```

```
class Fruit {

    //...
}
class Apple {

   private Fruit fruit =
   //...
}
```



https://www.pinterest.com/pin/337068[illegible]1039957/

Polymorphism & what could be the issue

What are the properties that are (not) perfect?

Dynamic binding & polymorphism

- Inheritance relationship
  - Advantage of ***dynamic binding*** and ***polymorphism.***
  - ***Dynamic binding*** *means the JVM will decide at runtime which method implementation to invoke based on the class of the object.*
  - ***Polymorphism*** *means you can use a variable of a superclass type to hold a reference to an object whose class is the superclass or any of its subclasses.*
- Dynamic binding and polymorphism can help make **code easier to change**.

Example:
Dynamic binding & polymorphism

- Our code fragment uses a variable of a superclass type, e.g, a `Fruit`;
  - later we create a **brand new subclass** `Banana`,
  - the code fragment **will work without change** with instances of the new subclass.

- If Banana **overrides** any of Fruit's **methods** invoked by the code fragment, **dynamic binding** will ensure that Banana's implementation of those methods gets executed.
  - *This will be true even though class Banana didn't exist when the code frag written and compiled.*

Dynamic binding & polymorphism Example:

- *Observation:*

- Inheritance helps to **make code easier to change**
  - involves adding a new subclass.
- **However**, this is not the only kind of change you may need to make.

- In an inheritance, superclasses are often said to be "fragile,"
  - **one little change to a superclass** can ripple out and require changes in r places in the application's code.

Changing the superclass interface

- If the superclass is well-designed, with a clean separation of interface and implementation in the object-oriented style, any changes to the superclass's implementation shouldn't ripple at all.

- 1. Changes to the superclass's interface, however, can ripple out and break any code that uses the superclass or any of its subclasses.
- 2. What's more, a change in the superclass interface can break the code that defines any of its subclasses.

- *Well designed Interfaces rarely change*
  - ***Suggestion:*** *Do not put a method that could soon change to interfac*

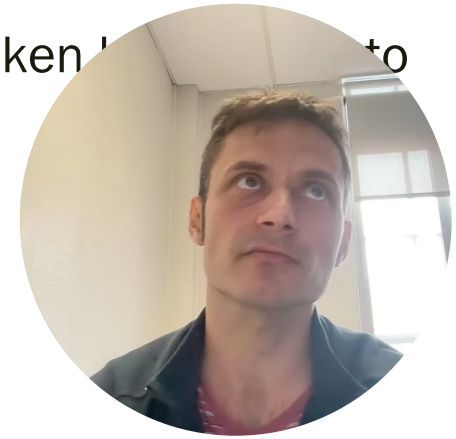
Example:
Changing the superclass interface

- We change the return type of a public method in class Fruit (a part of Fruit's interface),
  - it can break the code that invokes that method on any Fruit type reference or subclass.
  - In addition, it breaks the code that defines any subclass of Fruit that overrides the method.

- Such subclasses won't compile until we change the return value of the overridden method to match the changed method in superclass Fruit.

- Inheritance is also sometimes said to provide "weak encapsulation"
  - if we have code that directly uses a subclass, e.g., Apple, that code can be broken by changing to a superclass, e.g., as Fruit.

  - Apple apple = new Apple(); // now change Fruit



Example:
Changing the superclass interface

- ..

One of the ways to look at inheritance is that it allows subclass code to ***reuse*** superclass code.

- *If Apple **doesn't override a method** defined in its superclass Fruit, Apple **is** in a sense **reusing** Fruit's implementation of the method.*

- *Apple only "weakly encapsulates" the Fruit code it is reusing, because changes to Fruit's interface can break the code that uses Apple.*

Perhaps this is not the best usage of inheritance

The composition alternative

- Inheritance relationship makes it hard to change the interface of a superclass,
  - **looking at an alternative approach?**
  - provided by composition

- It turns out that when your goal is **code reuse**, then **composition** provides an approach that **yields** easier-to-change code to certain situations.

Example on comparison

Code reuse via inheritance

- **Code reuse via inheritance**
- Apple inherits (reuses) Fruit's implementation of peel()

```
class Fruit {
  // Return int number of pieces of peel that
  // resulted from the peeling activity.
  public int peel() {
      System.out.println("Peeling is appealing.");
      return 1;
   }
}

class Apple extends Fruit { }

class ClientCode {
  public static void main(String[] arg
     Fruit apple = new Apple();
     int pieces = apple.peel();
   }
}
```

Example on comparison

- Now you wish to change the return **int** value of peel() to new class type **`Peel`**
- You will break the code for `ClientCode`
- Your change to Fruit breaks `ClientCode` even though **`ClientCode` uses Apple directly and never explicitly mentions Fruit.**

```
class Peel {
   private int peelCount;
   public Peel(int peelCount) {
      this.peelCount = peelCount;
   }
   public int getPeelCount() {
      return peelCount;
    }
   //...
}
```

```
class Fruit {
  // Return int number of pieces of peel that
  // resulted from the peeling activity.
  public Peel peel() {
      System.out.println("Peeling is appealing.");
      return new Peel(1);
   }
}

class Apple extends Fruit { }

class ClientCode {
  public static void main(String[]
     Apple apple = new Apple();
     int pieces = apple.peel();
   }
}
```

Example on comparison Code reuse via composition

- Composition provides an alternative way for Apple to reuse Fruit's implementation of peel().
- Apple can hold a reference to a Fruit instance and define **its own peel() method** that simply invokes peel() on the Fruit.
  - Called delegation
- The subclass becomes the "frontend class" (**wrapper**), and the superclass becomes the "**backend** class."

```
class Fruit {
  // Return int number of pieces of peel that
  // resulted from the peeling activity.
  public int peel() {
      System.out.println("Peeling is appealing.");
      return 1;
   }
}

class Apple {
   private Fruit fruit = new Fruit();
   public int peel() {
        return fruit.peel();
   }
}

class ClientCode {
  public static void main(String[] a
     Apple apple = new Apple();
     int pieces = apple.peel();
   }
}
```

```
class Peel {
   private int peelCount;
   public Peel(int peelCount) {
      this.peelCount = peelCount;
   }
   public int getPeelCount() {
      return peelCount;
    }
   //...
}
```

Example on comparison

Code reuse via composition

- With **inheritance**, a subclass **automatically** inherits an implementation of any non-private superclass method that it doesn't override.
- With **composition**, the frontend class **explicitly** invoke a corresponding method in the backend class from its own implementation of the method.
- ...

```
class Fruit {
  // Return int number of pieces of peel that
  // resulted from the peeling activity.
  public Peel peel() {
      System.out.println("Peeling is appealing.");
      return new Peel(1);
   }
}

class Apple {
   private Fruit fruit = new Fruit();
   public int peel() {
      Peel peel = fruit.peel();
        return peel.getPeelCount();
   }
}

class ClientCode {
  public static void main(Str
     Apple apple = new Apple(
     int pieces = apple.peel(
   }
}
```

Having hundreds of clie

```
class Peel {
   private int peelCount;
   public Peel(int peelCount) {
      this.peelCount = peelCount;
   }
   public int getPeelCount() {
      return peelCount;
    }
   //...
}
```

Example on comparison

Code reuse via composition

- The **composition** explicit call is sometimes called "forwarding" or "**delegating**" the method invocation to the backend object.
- The **composition** approach to code reuse **provides stronger encapsulation than inheritance**, because a change to a backend class does not need to break any code that relies only on the frontend class.
- Changing the return type of Fruit's peel() method from the previous example doesn't force a change in Apple's interface and therefore needn't break `ClientCode`.

```
class Fruit {
  // Return int number of pieces of peel that
  // resulted from the peeling activity.
  public Peel peel() {
      System.out.println("Peeling is appealing.");
      return new Peel(1);
   }
}

class Apple {
   private Fruit fruit = new Fruit();
   public int peel() {
      Peel peel = fruit.peel();
        return peel.getPeelCount();
   }
}

class ClientCode {
  public static void main(Str
     Apple apple = new Apple(
     int pieces = apple.peel(
   }
}
```

Having hundreds of clie

How composition and inheritance compare? Backend class

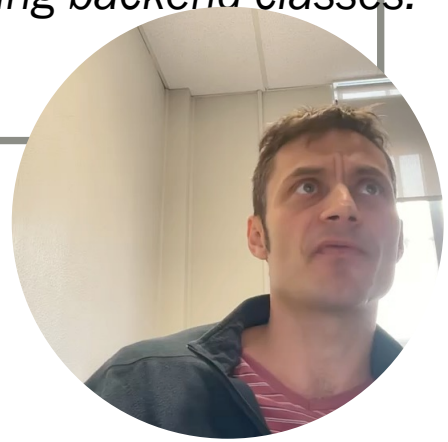
- It is **easier to change the interface of a backend class** (composition) than a superclass (inheritance).
  - *As the previous peel example illustrated, a change to the interface of a backend class necessitates*
    1. *a change to the frontend class implementation, but not necessarily the frontend interface.*
    - *code that depends only on the frontend interface still works,*
    - *since the frontend interface remains the same.*

  - *By contrast in inheritance, a change to a superclass's interface has two effects:*
    1. *It can not only ripple down the inheritance hierarchy to subclasses,*
    2. *It can also ripple out to code that uses just the subclass's interface.*

How composition and inheritance compare? Frontend class

- It is **easier to change the interface of a frontend class** (composition) than a subclass (inheritance).
  - *You can't just change a subclass's interface without making sure the subclass's new interface is compatible with that of its super-types.*
    - *E.g., you **can't add** to a subclass **a method with the same signature but a different return type** as a method inherited from a super-class.*

  - *Composition, allows you to change the interface of the frontend class without affecting backend classes.*



How composition and inheritance compare? Lazy load

- Composition allows you
  - to **delay the creation of backend objects until** they are **needed**,
  - change the backend objects dynamically throughout the lifetime of the frontend object.

- *With **inheritance** there is tight connection,*
  - *You get the image of the superclass in your subclass object image **as soon as the subclass is created**, and it remains part of the subclass object throug the lifetime of the subclass.*

How composition and inheritance compare? Subclasses

- It is easier to add new subclasses (inheritance) than it is to add new frontend classes (composition), because inheritance comes with polymorphism.
  - *If you have code that relies only on a superclass interface, that code can work with a new subclass without change.*
  - *This is not true of composition, unless you use composition with interfaces.*
  - ***Used together, composition and interfaces make a very powerful design***

How composition and inheritance compare? Performance

- The explicit method-invocation forwarding (or **delegation**) approach of composition will often have a **performance cost** as compared to inheritance's single invocation of an inherited superclass method implementation.
  - Virtual method (inheritance) is faster

How composition and inheritance compare? Implementation

- With both composition and inheritance, changing the implementation (not the interface) of any class is easy.
  - The ripple effect of implementation changes remain inside the same class.

Composition

- Defined dynamically at run-time through objects acquiring references to other objects.
- Respect each others' interfaces
- Keep each class encapsulated and focused on one task.
- Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters.
- Much more flexible than Inheritance.
  - Cannot change class extension at runtime, but with composition, you just defi
    "Type" which you want to use that can hold its different implementation.

Choosing composition OR inheritance

- **Make sure inheritance models the *is-a* relationship**
  Prefer inheritance to be used when a subclass *is-a* superclass. In the example above, an Apple likely is-a Fruit, so use inheritance.

- An important question to ask yourself when you think you have an is-a relationship is whether that is-a relationship will be constant throughout the lifetime of the application and, with luck, the lifecycle of the code.

  - *For example, you might think that an Employee is-a Person, when really Employee represents a role that a Person plays part of the time.*
  - *What if the person becomes unemployed?*
  - *What if the person is both an Employee and a Customer?*

  - *Such impermanent is-a relationships* ***should usually be modeled with composit***

Choosing composition OR inheritance

- **Don't use inheritance just to get code reuse**
- If all you really want is to reuse code and there is no is-a relationship in sight, **use composition.**
- **Don't use inheritance just to get at polymorphism**
- If all you really want is polymorphism, but there is no na **is-a relationship**, use **composition** with interfaces.

What is the right solution?

- Design patterns (DP)
  - Best practice to common situations and context.
  - Something we will do in this course now on..

Q/A

…?? ? .. ?