Composition

More relationship advice

establishing relationships between classes

- Two fundamental ways to relate classes

- *inheritance*
- *composition*

Inheritance

```
class Fruit {

    //...
}

class Tomato extends Fruit {

    //...
}
```

Composition

using instance variables that are references to other objects.

```
class Fruit {

    //...
}

class Tomato {

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

- class Tomato is related to class Fruit by composition, because Tomato has an instance variable that holds a reference to a Fruit object.
- Tomato is the *front-end class* and Fruit is the *back-end class*.
- In a composition relationship, the front-end class holds a reference in one of its instance variables to a back-end class.

Dynamic binding, polymorphism, and change

- When you establish an inheritance relationship between two classes, you get to take advantage of *dynamic binding* and *polymorphism*
- One of the prime benefits of dynamic binding and polymorphism is that they can help make code easier to change.
- If you have a fragment of code that uses a variable of a superclass type, such as Fruit, you could later create a brand new subclass, such as Banana, and the old code fragment will work without change, with instances of the new subclass.

- If Banana overrides any of Fruit's methods that are 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 fragment was written and compiled.
- Thus, inheritance helps make code easier to change if the needed change involves adding a new subclass. This, however, is not the only kind of change you may need to make.

Changing the superclass interface

- Superclasses are often said to be "fragile," because any change can ripple out and require changes in many other places in the application's code.
- What is actually fragile about a superclass is its 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.
- Changes to the superclass's interface, however, can ripple out and break any code that uses the superclass or any of its subclasses. What's more, a change in the superclass interface can break the code that defines any of its subclasses.

Aaaaaggggghhhhhh

- For example, if you change the return type of a public method in class Fruit (a part of Fruit's interface), you can break the code that invokes that method on any reference of type Fruit or any subclass of Fruit.
- Given this problem, it is worth looking at an alternative approach provided by composition.

Code reuse via inheritance

```
class Fruit {
public int peel() {
System.out.println("Peeling is appealing.");
return 1;
} }

class Tomato extends Fruit { }
```

```
class Example {
public static void main(String[] args) {
Tomato Tomato = new Tomato();
int pieces = Tomato.peel();
} }
```

Everything works fine.

However, you wish to change the return value of peel() to type Peel,…

Not easy – lets introduce Peel first

```
class Peel {
private int peelCount;

public Peel(int peelCount) {
this.peelCount = peelCount; }

public int getPeelCount() { return peelCount; }
//... }
```

Now updating

```
class Fruit {
public Peel peel() {
System.out.println("Peeling is appealing.");
return new Peel(1);
} }
class Tomato extends Fruit { }
```

Tomato works fine – where is the problem!

BOOM!!!!!!!!!!!!!!!!!!

```
class Example {
public static void main(String[] args) {
Tomato Tomato = new Tomato();
int pieces = Tomato.peel();
} }
```

This is broken and won't compile!!!!!!!!!!!!!!!!!!

Code reuse via composition

- Composition provides an alternative way for Tomato to reuse Fruit's implementation of peel().
- Instead of extending Fruit, Tomato can hold a reference to a Fruit instance and define its own peel() method that simply invokes peel() on the Fruit.

```
class Fruit {
public int peel() {
System.out.println("Peeling is appealing.");
return 1;
} }
class Tomato {
private Fruit fruit = new Fruit();
public int peel() { return fruit.peel(); }
}
```

```
class Example {
public static void main(String[] args) {
Tomato Tomato = new Tomato();
int pieces = Tomato.peel();
} }
```

Okay, where is the difference?

- In the composition approach, the subclass becomes the "front-end class," and the superclass becomes the "back-end class."
- With inheritance, a subclass automatically inherits an implementation of any non-private superclass method that it doesn't override.
- With composition, the front-end class must explicitly invoke a corresponding method in the back-end class from its own implementation of the method.

Lets return Peel

As before ….

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

public int getPeelCount() { return peelCount; }
//... }
```

As before .....

```
class Fruit {
public Peel peel() {
System.out.println("Peeling is appealing.");
return new Peel(1);
} }
```

So where is this difference?

Now Tomato needs to change!

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

If I need to update Tomato – what is the point?

Look at Example

```
class Example {
public static void main(String[] args) {
Tomato Tomato = new Tomato();
int pieces = Tomato.peel();
} }
```

It now works! It has not needed to change!

We have protect it (the user/client) from the interface change!!

Comparing composition and inheritance

- It is easier to change the interface of a back-end class (composition) than a superclass (inheritance).
- As the previous example illustrated, a change to the interface of a back-end class necessitates a change to the front-end class implementation, but not necessarily the front-end interface.
- Code that depends only on the front-end interface still works, so long as the front-end interface remains the same.
- By contrast, a change to a superclass's interface can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclass's interface.

Comparing composition and inheritance

- It is easier to change the interface of a front-end class (composition) than a subclass (inheritance).
- Just as superclasses can be fragile, subclasses can be rigid.
- You can't just change a subclass's interface without making sure the subclass's new interface is compatible with that of its superclasses.
- For example, you can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass.
- Composition allows you to change the interface of a front-end class without affecting back-end classes.

Comparing composition and inheritance

- Composition allows you to delay the creation of back-end objects until (and unless) they are needed, as well as changing the back-end objects dynamically throughout the lifetime of the front-end object.
- With inheritance, 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 throughout the lifetime of the subclass.
- It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism.
- If you have a bit of 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.
- With both composition and inheritance, changing the implementation (not the interface) of any class is easy.

Composition versus Inheritance

- **Make sure inheritance models the *is-a* relationship**
  inheritance should be used only when a subclass *is-a* superclass. In the example above, an Tomato is-a Fruit, so I would be inclined to 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.
- 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 Supervisor? Such impermanent is-a relationships should usually be modelled with composition.
- **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 natural is-a relationship, use composition with interfaces.