Class Hierarchy

Discussion D

Constructor

```
public class X {
 private int capacity;
 public X() { capacity = 16;}
 public X(int i) {capacity = i;}
 public int getCapacity() {return capacity;}
}

public class Y extends X {
 private double loadFactor;
 public Y(double d) { this.loadFactor = d;}
 public getLoad() {return loadFactor;}
}
```

Usage

```
public static void main(String [] args) {
  X x = new X(32);                      //1
  Y y = new Y();                        //2
  Y y2 = new Y(0.25);                   //3
  x = (X) y;                            //4
  x.getCapacity();                      //5
  x.getLoad();                          //6
  y = (Y) x;                            //7
}
```

```
X x = new X(32);                    //1
```

- Allocate a new object of class **`X`**
- Invoke **`X(int capacity)`** constructor
  - Invoke **`super()`**
    - **`Object()`**
  - capacity == 32
  - return

```
Y y = new Y();                    //2
```

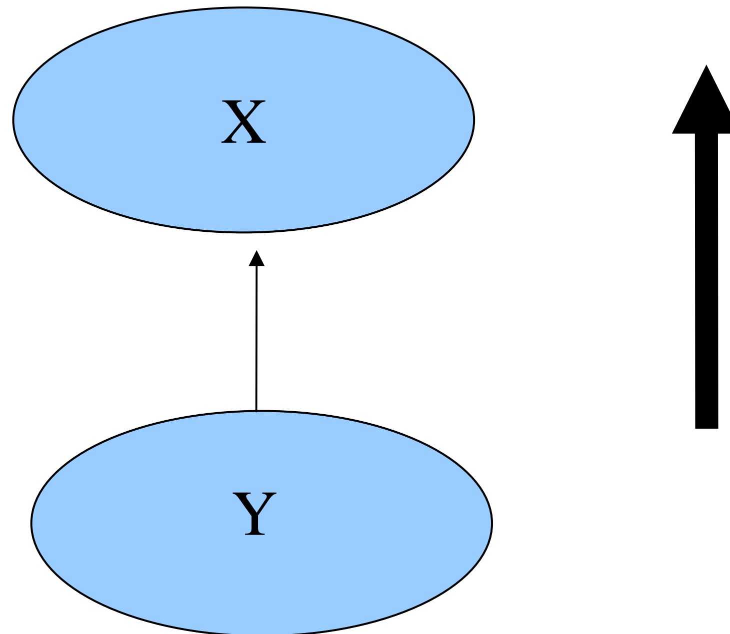
- No **`Y()`** constructor defined
  - Constructors are not inherited
- Uses compiler generated
  - **`Y() { super(); }`**
- Invokes **`X()`**
  - Invokes **`Object ()`**
- capacity == 16
- loadFactor == 0.0

```
Y y2 = new Y(0.25);                //3
```

- Allocates a new object of class **`Y`**
- ***Invokes*** ***`Y(double loadFactor)`***
  - Invokes **`X()`**
    - Invokes **`Object()`**
    - capacity == 16
  - loadFactor == 0.25

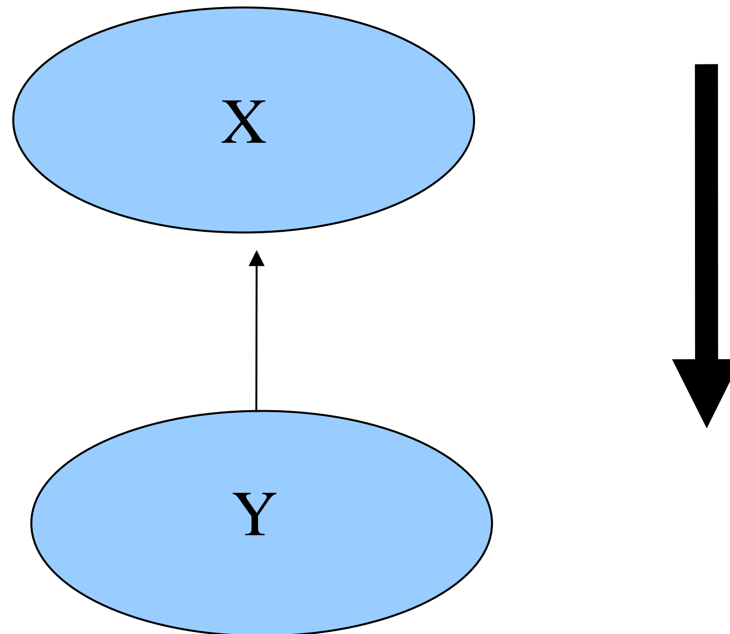
```
x = (X) y;                              //4
```

- Is unnecessary
- Improves readability



```
y = (Y) x;                              //7
```

- Is correct
- will succeed check cast at run time



```
x.getLoad();                    //6
```

- Substitutability
  - **`y`** extends **`x`**
  - **`y`** can be substituted for **`x`**
- Variable declared of Type **`X`**
- cannot access methods of Class **`Y`**

Polymorphism

```
public class X {
 private int [] data;
 public X(int [] data) { this.data = data;}
 public int [] sort() {...}
 public void print() {}
}

public class Y extends X {
  public Y(boolean direction) {}
 public int [] sort(boolean descending) {...}
 public void print() {}
}
```

Visibility

```
public class X {
    public void u (p(););
    private void p();
}

public class Y extends X {
    private void p();
}

 Y y = new Y();
 y.u();
```

Hierarchy

```
public class X extends Y {}
public class Y extends X {}
```

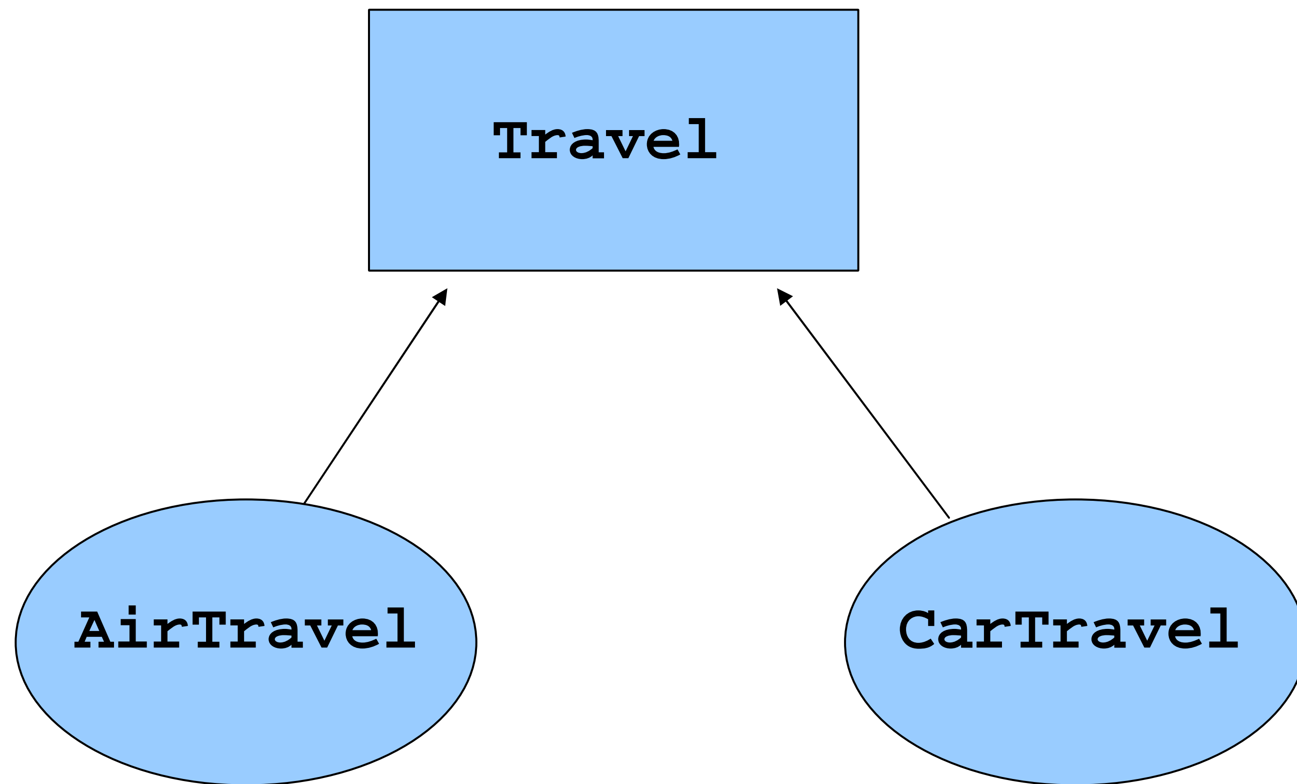
```
public class X {}
public class Y extends X {
public class Z extends X { private Y;}
```

Hierarchy

```
public class X {}
public interface Y {}
public class Z extends X implements Y {}
```

```
public interface X {}
public interface Y extends X {}
public class Z implements Y {}
```

`Travel` Hierarchy



```
public abstract class Travel {

  public Travel (String source,
                 String Destination){}

   public double cost() {}   //cost of travel,fare
   public int time() {}    //travel time for meals
   public int compensation() {}    //dollar amount

}
```

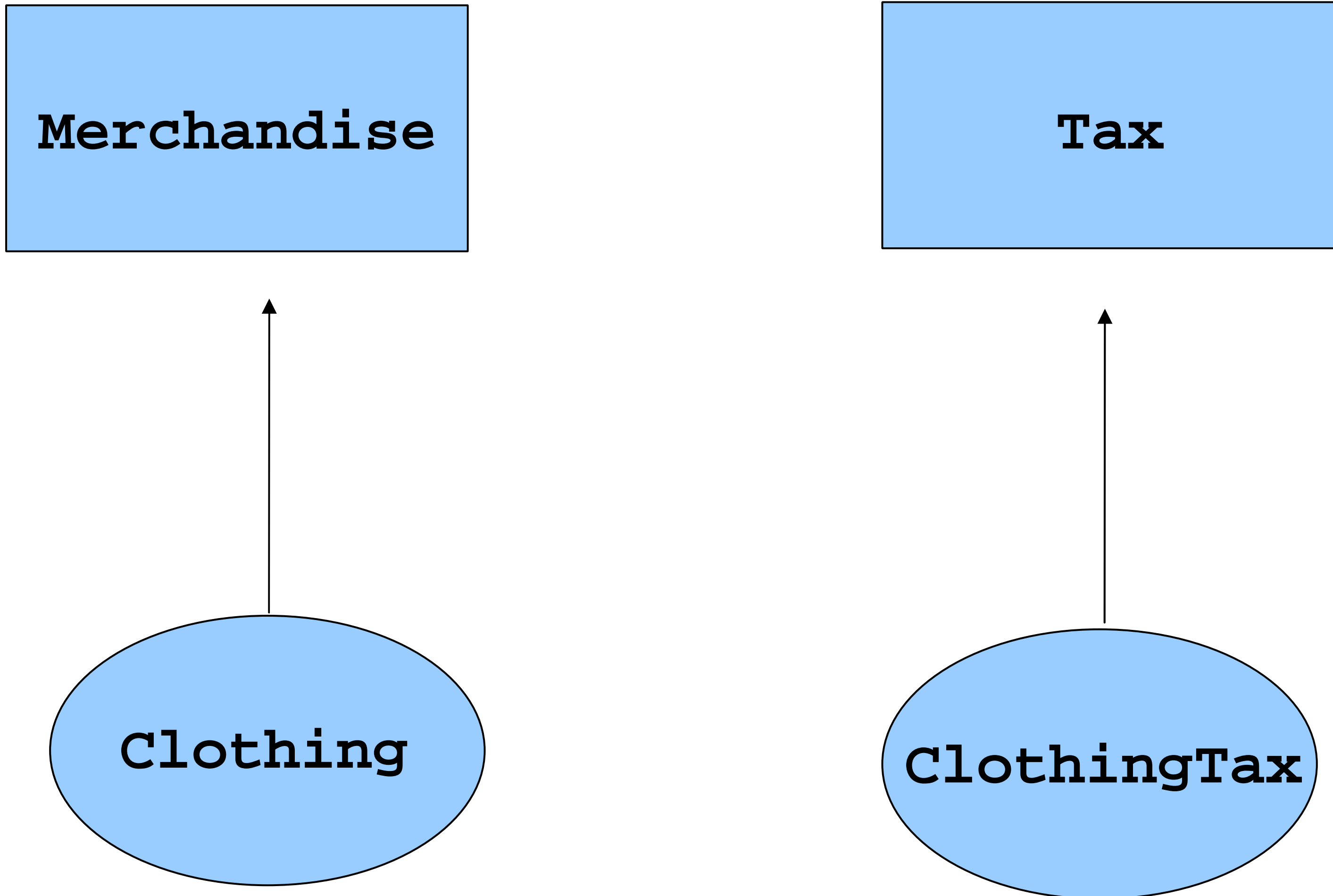
```
public class AirTravel extends Travel {

   public AirTravel(String source, String destination,
                    String airline)
   {
     super (source, destination);
   }


    public AirTravel(String source,  String destination,
                    String airline, String class) {
        super (source, destination);
   }

   public double cost();//verify cost from airline web
   public int time();
   public int compensation () {
     // compute allowable parking, meals etc.
   }
}
```

```
public class CarTravel extends Travel {

   public CarTravel(String source, String destination,
                    String route) {
     super (source, destination);
   }

   public double cost(){
     // compute mileage
     // maybe from yahoo maps directions
     // based on a per mile cost, compute total cost
   }
   public int time();
   public int compensation () {
     // compute allowable parking based on
     // num days spent at destination
   }
}
```

Why `Travel` Interface?

- Clients of **`Travel`** can compute the necessary information without knowing all the details
- Flexibility is limited to the generality of the **`Travel`** interface for use by the clients
- If the interface needs modification then object oriented programming benefits are lost

Merchandise

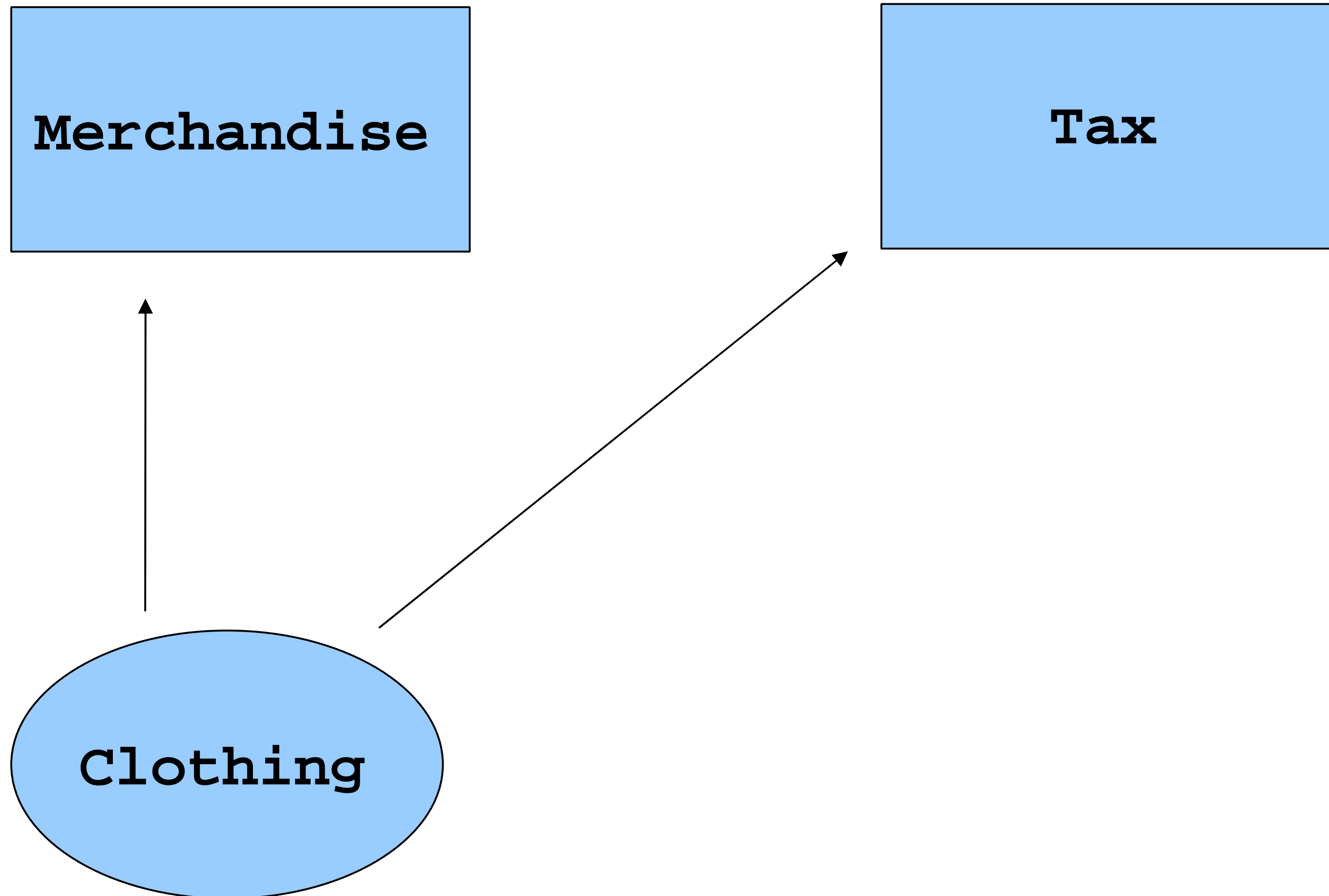


```
public abstract class Merchandise {

 Tax tax;
 public int getCost() {}

 public int getTax(int zipCode) {
   return tax.getTax(zipCode);
 }
}

public class Clothing extends Merchandise {

  public Clothing () {
   tax = new ClothingTax(this);
  }

  public int getCost() {}
}
```

```
public abstract class Tax {
 Merchandise article;
 public Tax();
 public int getTax(int zipCode);
}

public class ClothingTax extends Tax {
 //imagine a static zipcode indexed table for looking up
 //taxation
 public int getTax(int zipCode);
}

public class PharmaTax extends Tax {
 public int getTax(int zipCode);
}
```

We may want to model zip code explicitly using a Location class.

Interfaces



Extension

- Assumed that tax rate was flat for a type
- It may depend on cost of item
  - Clothes > $250 may be taxed differently

Detailed