Outline

- Problem
- Solution Analysis and Design
  - IS_A Relationships and Inheritance
  - Design Evolution
  - Abstracting the Hierarchy
The Problem

- Ralph owns the Trinidad Fruit Stand
- He sells fruit on the street
- He wants to use a computer to track his produce
- Sale items include:
  - Apples
  - Bananas
  - Lemons
- Ralph uses his old records to design a data structure

<table>
<thead>
<tr>
<th>Apple</th>
<th>Banana</th>
<th>Lemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>variety</td>
<td>origin</td>
<td>country</td>
</tr>
<tr>
<td>cost</td>
<td>weight</td>
<td>weight</td>
</tr>
<tr>
<td>weight</td>
<td>isOrganic</td>
<td>cost</td>
</tr>
<tr>
<td>getCost()</td>
<td>getCost()</td>
<td>getCost()</td>
</tr>
<tr>
<td>print()</td>
<td>print()</td>
<td>print()</td>
</tr>
<tr>
<td>core()</td>
<td>peel()</td>
<td>squeeze()</td>
</tr>
</tbody>
</table>
The Problem

- Ralph then made a FruitI interface
  - Using common methods

```java
public interface FruitI {
    public int getC0st(); // classes must implement!
    public void print();
}
```

- Using this FruitI interface he made an Apple0 class

```java
class Apple0 implements FruitI {
    private int weight;
    private String variety;
    private final int CENTS_PER_OZ;

    public Apple0(int weight, int costPerOz, String variety) {
        this.weight = weight;
        this.CENTS_PER_OZ = costPerOz;
        this.variety = variety;
    }

    public void print() {
        System.out.println("variety + " "Apple");
    }

    public int getC0st() { // Apple0 class implements a FruitI method
        return this.weight * CENTS_PER_OZ;
    }
    // ... core() and other code not shown ...
}
```
The Problem

- Ralph buys and sells his produce using weight and cost
  - He buys in bulk, his cost is in cents per ounce
    - His cost does not change after he buys it
  - He sells fruit based on its weight
  - When he wants to know the cost
    - He wants to know the total cost for the fruit
    - This means \texttt{getCost()} must produce the cost based on weight
The Problem

- Making an interface solves one problem..
- It treats different fruits as “just some fruit”
- Since each fruit class will use the *FruitI* interface
  - This supports polymorphism
- This approach has issues…
  - After writing *Apple0* and *Banana0* Ralph notices something…
    - There is a lot of code duplication…
    - This would be common in all fruits!
  - The interface helped specify common methods
    - But not common data fields…

- Ralph wants our help with his Trinidad.java program!
Solution Analysis and Design
IS_A Relationships and Inheritance

• Inheritance creates an IS_A relationships from different interfaces

• What does inheritance mean???
  • In a family, an inheritance is something that is passed down
  • Such as a gene for blue eye…
  • OO inheritance is basically the same thing…
  • We can create classes that have an inheritance relationship
• Terminology:
  • Base (or super) class: the more general class
  • Inherited from
  • Subclass: extends a superclass
  • It inherits
  • The extends keyword says we are creating an inheritance
  • Some call this a parent-child relationship
• The Java class libraries are a great example of inheritance
  • The Object class is the class that every class inherits from.
  • You do not have to state this...
Solution Analysis and Design
Design Evolution

- If we revise the fields a bit…
  - We can see common attributes of all fruits
  - We can see common methods too…
- We can write a *FruitC* class
  - A generalization that can hold common attributes
  - For fields and methods that apply to one kind of fruit…
    - We will make *Apple*, *Banana*, and *Lemon* classes
- We will connect the individual fruit classes to the *FruitC* class
  - We will use inheritance
  - This will make an *Apple* object that also *IS_A FruitC*
- We will do two versions of this:
  - Version 1: factor out common properties
  - Version 2: make it more *abstract* by doing a bit of “tuning”
Solution Analysis and Design
Design Evolution

- Version 1: Factoring out the superclass
  - Lets start with the *Apple0* class
    - Factor out the common characteristics into *FruitC*
      - *FruitC* will handle everything common to all fruits
    - The remaining stuff goes into *Apple1* class...
  - But what gets moved?
    - What is common in all fruits?
      - *Cost per ounce*
      - *Weight*
      - *Origin* (renamed from country)
      - *IsOrganic* (not really used for Apples/Lemons, but could be)
        - Use false as a default value
    - Everything else stays...
  - Repeat for Lemons and Bananas...
Solution Analysis and Design

Design Evolution

- Version 1: Factoring out the superclass
- Our FruitC class is taking shape

<table>
<thead>
<tr>
<th>FruitC</th>
<th>Apple</th>
<th>Banana</th>
<th>Lemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>costPerOz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>origin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isOrganic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>variety</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FruitC(...)</td>
<td>Apple(...)</td>
<td>Banana(...)</td>
<td>Lemon(...)</td>
</tr>
<tr>
<td>toString()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>core()</td>
<td>peel()</td>
<td>squeeze()</td>
</tr>
</tbody>
</table>

- Note: We added a `toString()`
- Will allow us to print our fruit...
Solution Analysis and Design

Design Evolution

- Version 1: Factoring out the superclass
  - Now construct the code for our *FruitC* class
    - Add the common fields
    - Make constructors

```java
public class FruitC {
    ...
    FruitC( int weight, int costPerOz, String origin, boolean isOrganic ) {
        this.weight = weight;
    }
    ...
    FruitC( int weight, int costPerOz, String origin ) {
        this( weight, costPerOz, origin, false ); // how java calls another constructor
    }
    ...
}
```

- Make the common methods...
  - We will see this in the code
Solution Analysis and Design

Design Evolution

- Version 1: Factoring out the superclass
  - Now construct the code for our *Apple* class
    - Add the *Apple* only fields
    - Make *Apple* constructor
    - Make the *Apple* only methods...

```java
public class Apple extends FruitC {
    private String variety;

    public Apple( int weight, int costPerOz, String origin, String variety ) {
        super( weight, costPerOz, origin ); // must call super() first
        this.variety = variety;
    }

    public int getCst() { // overrides getCst() to add surcharge
        return 1 + super.getCst(); // make an ’up call’ to superclass
    }
}
```

- Repeat for *Banana* and *Lemon* classes
Solution Analysis and Design

Design Evolution

• Version 1: Factoring out the superclass
  • To the code!
Solution Analysis and Design

Design Evolution

- Version 1: Factoring out the superclass
- Testing
  - Validate code by compiling it
  - Should be done often
    - Write a method… compile it
    - Change/add a method or field… compile it
- While writing we can use the main of each class as a test program…
  - We should test everything
    - *Constructors*
    - *getWeight()*
    - *getCost()*
    - *getOrganic()*
    - *isOrganic()*
    - *ToString()*

  - **Note:** this is simple testing… we will want to do more as we develop
Solution Analysis and Design
Design Evolution

- Version 1: Factoring out the superclass
- UML
Solution Analysis and Design
Design Evolution

- Version 2: Abstract Classes
  - We have a problem
    - Makes no sense to make an instance of `FruitC`
      - `FruitC fruit = new FruitC( 1, 1, "somewhere" );`
  - We can make this impossible
    - By using an abstract class
      - Being abstract it cannot be instantiated

```java
public abstract class FruitA {
    // ...
}
```
Solution Analysis and Design
Design Evolution

• Version 2: Abstract Classes
  • Abstract classes can contain fields and methods
    • Methods can be implemented or made abstract
      • An abstract method has no body in the abstract class
      • The subclass must implement the abstract method
  • Let’s say we wanted to calculate price differently for each fruit
  • But each fruit must have a method to calculate price
  • We can enforce this with an abstract method

```java
public abstract class FruitA {
    // ... other content including fields, constructors, constants, methods

    /** @return sale price of this fruit */
    public abstract int getPrice(); // there is a ';' instead of a method body
}
```
Solution Analysis and Design
Design Evolution

- Are two fruits equal?
  - We have to make a choice here
  - What type of equality:
    - Type equality?
    - Value equality?
    - Weight equality?
    - “some other” equality?
  - Trinidad fruit stand say they are equal when:
    - Same type of fruit
    - Organic feature
    - Price
  - How do we do this?
Solution Analysis and Design

Design Evolution

- Are two fruits equal? (con’t)
- Implement the equals method for each fruit…
  - Not in the FruitC class

```java
public class Apple2 extends FruitA {
    // ...

    public boolean equals( Object obj ) {
        boolean equal = false;
        if ( obj instanceof Apple2 ) { // good start; same 'kind'...
            Apple2 other = (Apple2) obj; // casting to use specific methods
            if ( ... ) { // make comparisons using 'other' reference
                equal = true;
            }
        }
        return equal ;
    }

    // ...
```
Questions?