1 Problem

Ralph owns the Trinidad Fruit Stand that sells its fruit on the street, and he wants to use a computer to track its produce. Their sale items include apples, bananas, and lemons. Starting from his old records, Ralph began to design his data structures, and made this sketch for some classes, their fields and methods:

<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>

Ralph then developed common methods for a FruitI interface. Below is Ralph’s resulting interface and his initial Apple0 class.

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

class Apple0 implements FruitI {

    private int weight;
    private String variety;

    private final int CENTS_PER_OZ = 12;

    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(this.variety + " Apple");
    }

    public int getCost() {  // Apple0 class implements a FruitI method
        return this.weight * CENTS_PER_OZ;
    }

    // ... core() and other code not shown ...
}
```
Ralph buys and sells his produce using weight and cost. When he buys in bulk, his cost is in cents per ounce, and he sells a piece of fruit based on its weight. His ‘cost’ is the cost per ounce, and that cost does not change after he bought it. When he wants to know the cost, he wants to know his total cost for the fruit; that means `getCost()` must produce the cost for the particular weight of the fruit.

Defining the interface solves the problem of treating different fruits as ‘just some fruit’. Since each class implements the same methods and can be referenced by values of the `FruitI` interface type, the approach supports polymorphism.

While an interface allows Ralph to treat different fruits as ‘just some fruit’ and collect or process multiple kinds of fruit as a collection, this approach has some issues. After writing `Apple` and `Banana`, Ralph realized there would be a lot of duplication with all fruits as separate classes. The `FruitI` interface helped specify the common methods but not the common data fields.

Ralph wants our help with his `TrinidadFruitStand.java` program.

2 Solution Analysis and Design

2.1 IS-A Relationships and Inheritance

Inheritance creates IS-A relationships different from interfaces. What does the word inheritance mean? In a family, an inheritance is something that is ‘passed down’, such as a gene for blue eyes. Object-oriented inheritance relationships are similar; we can create classes that have an inheritance relationship. In OO terminology, we call the more general class a base class or a superclass. When we create another class that inherits from that superclass, we say we are creating a subclass, which extends the superclass. In Java, the keyword `extends` tells the Java compiler we are creating an inheritance relationship.

Some books refer to the IS-A relationship as a parent-child relationship, in which the newborn child inherits characteristics from a parent. The ‘parent’ class passes down attributes and methods (variables and code) to a ‘child’ class. Any subclass instance we create inherits the fields and methods declared and defined in the superclass. This inheritance mechanism saves us from having to duplicate code in multiple, similar classes.

The Java class libraries embody an extensive inheritance hierarchy. The class `Object` is the superclass from which all classes extend. Every class javadoc shows that the class extends from `Object`.

2.2 Design Evolution

If we revise the fields a bit, we can find attributes common to apples, bananas and lemons, and we can do something similar with methods. We will write a `FruitC` class that is a generalization that can hold common attribute values. For the fields and methods that apply only to one kind of fruit, we will create `Apple`, `Banana`, and `Lemon` classes that specialize the `FruitC` class. When

1. We have named this `FruitC` for reasons that will be come clear later. For now, think of the `C` suffix as one that means ‘common’.

2
we connect the **Apple** subclass to **FruitC** using **inheritance**, we will be able to construct an Apple object that also **IS A** FruitC.

We will proceed this way:
- Create version 1, v1, of an inheritance hierarchy by **factoring** common properties from specific classes and constructing the classes with testing along the way.
- Create version 2, v2, to make it more **abstract** by ‘tuning’ the hierarchy to ensure it enforces the right behaviors.

The accompanying **Code.zip** file contains directories for each version of the source code, including the interface sketch.

### 2.2.1 Factoring out a superclass

We start with the **Apple0** class and **factor out** the common characteristics into the **FruitC** superclass that our new **Apple1** class **extends.** FruitC handles the fields and methods that should be common to all fruits. The remaining specifics of **Apple0** move into the **Apple1** subclass.

Before implementing the hierarchy, we need to modify the initial data attributes based on an analysis of the similarities and differences between Apple, Banana and Lemon attributes.

Since the cost per ounce and weight attributes are common; we can put these into the new, FruitC superclass. Similarly, we can rename **country** to **origin** and add the field to **FruitC**. Although **isOrganic** is not in Apple or Lemon, each of these classes could have a value for that attribute. If we choose a **false default value**, then we can put **isOrganic** into the **FruitC** superclass. That leaves the attribute **variety** as a field specific to **Apple**, and the other kinds of fruits have no additional specific attribute values.

Our hierarchy is now taking shape. The **FruitC** class has fields to store the kinds of values common to the different kinds of fruit; information can be stored more consistently, in a location that is reusable by more than one subclass.

What remains is to decide on the methods that **FruitC** should have. Since Java is our implementation language, we need a way to **construct** all the parts of our objects, and we can **overload** the toString() method from the **Object** class. To design for Java, we replace **print()** with a **toString()** that will allow clients to get a printable representation of the instance.

<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>variety</td>
<td></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>getCost()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>core()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peel()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>squeeze()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.2 Constructing the classes

Now we create the superclass and **stub out** its fields and methods. The **FruitC** constructor needs 4 parameters: weight, costPerOz, origin, isOrganic. However, since isOrganic has a default
value, we can add a 3 parameter constructor with weight, costPerOz, and origin. The 3 parameter constructor can call the 4 parameter constructor and pass the default value to it. Java syntax for one constructor to call another is this(...).

```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
    }
    ...
}
```

The `getWeight` is common and easy to implement, `getCost` will return weight * costPerOz, and `toString` will return "Fruit". The `getOrigin` and `isOrganic` methods are trivial accessors, and there are no mutators because the field values should not change after construction.

Now we create stubs for the subclasses and connect them to the superclass. The Java syntax for calling a superclass constructor is `super(...)`, which must be the first statement inside the constructor.

Another use for `super` is to invoke a method in a superclass. For example, suppose that the supplier adds a surcharge for a shipment of `Apple`, and that surcharge is one cent per Apple. The store might handle this by overriding `FruitC.getCost` and including the surcharge in the result of calling `Apple1.getCost`. The `Apple1.getCost` would call `super.getCost` to get the cost, add the surcharge, and return the sum.

```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 getCost() { // overrides getCost() to add surcharge
        return 1 + super.getCost(); // make an 'up call' to superclass
    }
    ...
}
```
2.2.3 Testing

We compile the individual classes to validate our code syntax. That should happen frequently, as we write every few methods or classes, to ensure that our code continues to obey the type specifications, interfaces, method names, and Java syntax.

We can adapt the main program to test construction of the Apple, Banana, and Lemon instances and operation of their methods.

We should make sure that we test these things:

- Constructors: the superclass has two constructors, and each subclass has at least one of its own constructors.
- getWeight()
- getCost()
- getOrigin()
- isOrganic()
- toString()

While this main program performs some simple tests, such an approach would not be advisable for a full implementation of this application. We would want to develop this program into a more comprehensive test harness for the hierarchy. As the design and implementation proceeds, we would develop test cases in this test program.

Finally we show a class diagram of our hierarchy using the Unified Modeling Language (UML). The solid line with the ‘hollow’ arrowhead points from the specialized classes to the general class. The – and + signs before the fields and methods symbolize private and public respectively.
2.3 Version 2: Abstracting the hierarchy

2.3.1 Abstract Classes

One problem with our work makes no sense: it is possible to construct an instance of FruitC. We do not want FruitC to be a concrete class; we want it to be true that every instance of a FruitC is 'some actual, real kind' of fruit. In terms of code, the statement below is legal but undesirable:

```java
FruitC fruit = new FruitC( 1, 1, "somewhere" );
```

To make it impossible for clients to construct an instance, we make a class abstract, and the compiler will enforce that the class cannot be instantiated. Our concrete, FruitC class becomes an abstract class, FruitA.

```java
public abstract class FruitA {
    // ...
}
```

Making a class abstract is sufficient to prevent its instantiation even when it implements all the methods in its interface\(^2\).

2.3.2 Abstract Methods

If we add a method, getPrice() for example, we learn that the implementation of this method depends on the specific subclass because the customer wants to use a different profit factor for different fruits. In that case, we want to make the method abstract in the superclass. Whenever that happens, we must also make the class itself abstract.

Here is how an abstract class, FruitA, declares 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

}
```

Other than the abstract keyword, the declaration of an abstract method has the same structure as the declaration of an interface method.

2.3.3 Instance Equality: defining equals()

Now that we can subsume one type ‘under’ another, we have to think about what equals means. When is one instance of Fruit equal to another?

We should write the equals method to mean some sort of equivalence, but which should it be?

- type equality?
- value equality?
- weight equality?
- 'some other' equality . . . ?

---

2. Here we mean interface in the general sense, not in the Java syntax sense.
The application requirements may specify different needs for comparing instances for equality and equivalence. There are too many possibilities to be able to enumerate these easily. In the case of the Trinidad Fruit Stand, we might suppose that two fruits are equivalent if they have the same kind of fruit (i.e. the same subclass), organic feature, and overall cost because this is what the store would tell clients and how it prices fruits.

At a minimum for `equals` in an inheritance hierarchy, the code needs to recognize the class of an instance, cast the `Object` argument to that specific class, and compare the states of the instances, as shown in the following code fragment for a concrete subclass.

```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;
    }

    // ...
}
```

We also need equivalent instances to be recognized for hashing purposes. To fulfill the requirements of Java, if two instances are `equals`, then their hash codes should be the same. We need to write the `hashCode()` method so that equivalent instances hash to the same code. Since all the information of interest is in the `FruitA` class, we can write `hashCode` once, and it should cover the needs of the subclasses.

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

    public int hashCode() {
        int val = getCost();
        return val * val + (val << 1) + (isOrganic ? 1 : 0);
    }

    // ...
}
```