

3.1 Designing Methods: Unions of Classes

In the previous lab you have designed the class hierarchy that represents the following kinds of pets:

- **cats** where we record whether it is a short-hair cat or a long-hair cat
- **dogs** where we record the breed (e.g. Husky, Labrador, etc., or Mutt — describing an unknown breed)
- **gerbils** where we need to know whether it is a male or female

still keeping track of the name of the animal and of its owner.

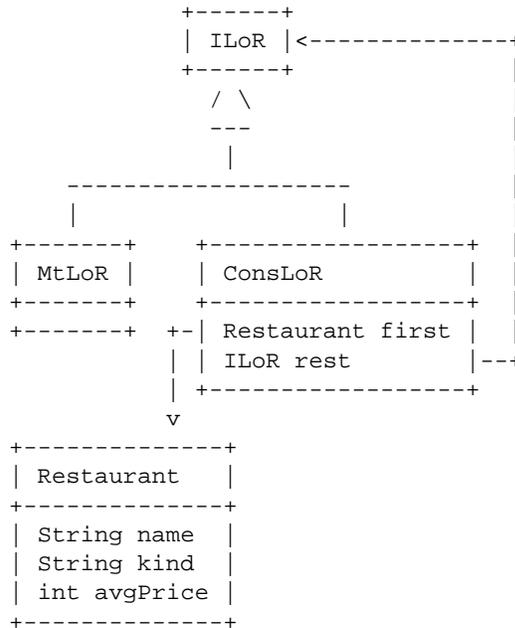
1. Design the method `isAcceptable` that determines whether the pet is acceptable for a child that is allergic to long haired cats, gets along only with Labrador and Husky dogs, and should not have a female gerbil pets.
2. Design the method `isOwner` that determines whether this animal's owner has the given name.
3. Design the method `sameOwner` that determines whether two pets have the same owner.

3.2 Designing Methods: Self-Referential Class Hierarchies

We will work with a list of restaurants. The code in the file `restaurant-list.java` defines the class hierarchy represented by the class diagram on the following page:

1. Design the method `count` that counts the number of restaurants in a list of restaurants.
2. Design the method `averageDinner` that computes the average of the average prices for all restaurants in a list of restaurants. For the `rlist3` in the file `restaurant-list.java` the method should produce \$16.
3. Design the method `cheapList` that produces a list of all restaurants that have the average dinner price below the given limit.

4. Design the method `sort` that produces a list of all restaurants sorted by their average dinner prices.



3.3 Using the draw Library

Download the program *draw-face.java*. Run it.

The program illustrates the use of the draw library that allows you to draw shapes on a Canvas. The first three lines specify that we will be using three libraries (programs that define classes for us to use). The `colors` library defines a union of six colors (black, white, red, yellow, blue, and green) through the interface `IColor`. The `geometry` library defines a single class `Posn` that has no methods besides the constructor. The draw library does the work – allows you to define a Canvas of the given size and to draw shapes on the Canvas.

Use this program and the online documentation as a guide for the homework assignment.

3.4 Designing Recursive Methods with Accumulators

Our goal is to understand how we can use accumulators to keep track of the knowledge we have acquired as we progress towards the program solution.

We are given a list of `Strings` and want to combine all of them into one `String`. We also want to find the length of the longest `String` in this list.

Here is an example:

Initial list:

```
"Mary "  
"had "  
"a "  
"lamb. "
```

Expected result:

```
"Mary had a lamb. "
```

Our solution that follows the DESIGN RECIPE is easy (see *string-acc.java*):

```
// to represent a list of Strings  
interface ILoS{  
    // combine all Strings in this list into one  
    String combine();  
  
    // find the length of the longest word in this list  
    int maxLength();  
}  
  
// to represent an empty list of Strings  
class MtLoS implements ILoS{  
    MtLoS(){  
  
        // combine all Strings in this list into one  
        String combine(){  
            return "";  
        }  
  
        // find the length of the longest word in this list  
        int maxLength(){  
            return 0;  
        }  
}  
  
// to represent a nonempty list of Strings  
class ConsLoS implements ILoS{  
    String first;  
    ILoS rest;  
  
    ConsLoS(String first, ILoS rest){  
        this.first = first;  
        this.rest = rest;  
    }  
}
```

```

// combine all Strings in this list into one
String combine(){
    return this.first.concat(this.rest.combine());
}

// find the length of the longest word in this list
int maxLength(){
    return Math.max(this.first.length(),
                    this.rest.maxLength());
}
}

// to represent examples for lists of strings
class Examples{
    Examples(){}

    ILoS mary = new ConsLoS("Mary ",
                            new ConsLoS("had ",
                                          new ConsLoS("a ",
                                                        new ConsLoS("little ",
                                                                      new ConsLoS("lamb.", new MtLoS())))));

    boolean testCombine =
        check this.mary.combine()
        expect "Mary had a little lamb.";

    boolean testMaxLength =
        check this.mary.maxLength() expect 7;
}

```

Designing the loops with accumulators

We now convert these methods to use the accumulator style. As we did in Scheme, we will design a helper method that takes one more argument, the accumulator. The accumulator at any given point in the computation will hold the “answer” to our question thus far.

The first question to ask is:

Step 1: What is the type of the expected result?

This determines not only the return type for the method, but also the type of the value that will be accumulated. If the type is `AccType`, we add an argument `AccType acc` to the method signature and add `Acc` to its name.

That means we add the following methods to the interface:

```
// combine all Strings in this list into one
String combine();

// combine all Strings in this list into one
// with the given String
String combineAcc(String acc);

// find the length of the longest word in this list
int maxLength();

// find the length of the longest word in this list
// given the max length so far
int maxLengthAcc(int acc);
```

The next step is to change the purpose statements to explain the meaning of the accumulated value.

Step 2: Change the purpose statements for all new methods.

For example, we say:

```
// combine all Strings in this list into one
// adding the Strings from the list to the String
// that contains all the Strings we have found so far
String combineAcc(String acc);

// find the length of the longest word in this list
// if it is longer than the known longest length so far
// otherwise, produce the known longest length
int maxLengthAcc(int acc);
```

Next we need to figure out what are the values each method produces when it is invoked with the empty list. Think of where can you find out. We call this the **base value**.

Step 3: Make a list of the base values for all new methods.

Of course, the base value for the `maxLength` method is 0 . What is the base value for the method `combine`?

We now deal with the following problem. The original question did not mention the accumulator. That means that the method that uses the accumulator is only our helper method. The original method needs to invoke this helper so that it would produce the original result.

There are two steps that make this possible.

Step 4: Invoke the method that uses the accumulator using the base value as the initial accumulator value.

For example we get:

```
// find the length of the longest word in this list
int maxLength(){
    return this.maxLengthAcc(0); }
```

As the things stand now, we need to add this method body to the method definition in both the empty class (MtLoS) and the class that represents a nonempty list (ConsLoS). In the next section we will learn how to do this only once.

Revise the definition of the method `combine` in a similar way.

The next step deals with the class that represents the empty list.

Step 5: Implement the method body in the class MtLoS.

The purpose statement makes it clear that the method just produces the value of the accumulator. This makes sense. If the original method is invoked by an instance of the empty class, it produces the base value, as expected.

Implement all method bodies for the class MtLoS.

Now comes the hardest part of the whole process.

Step 6: Design the update method.

Let us examine what happens inside of the method body in the class ConsLoS. We list again all original method definitions here:

```
// combine all Strings in this list into one
String combine(){
    return this.first.concat(this.rest.combine());
}
```

```
// find the length of the longest word in this list
int maxLength(){
    return Math.max(this.first.length(),
                    this.rest.maxLength());
}
```

In each case, there is some computation that involves `this.first` and the recursive invocation of the original method by the list `this.rest`.

The update method has the following common structure:

```
// update the value of the accumulator using this.first value
AccType updateMethodName(String first, AccType acc){ ... }
```

For the method `maxLength` the `updateMaxLength` method is defined as:

```
// update the value of the longest word so far
int updateMaxLength(String first, int acc){
    return Math.max(this.first.length(), acc);
}
```

Design the update method for the `combineAcc` case. And, yes, *do follow the design recipe*.

We are just about done.

Step 7: Complete the body of the method using the update method.

The method that uses the accumulator is the same for all cases. The instance of the `rest` of the list invokes the method self-referentially, using the updated value of the accumulator.

For the `count` method this will be:

```
// find the length of the longest word in this list
// if it is longer than the known longest length so far
// otherwise, produce the known longest length
int maxLengthAcc(int acc){
    return
        this.rest.maxLengthAcc(this.updateMaxLength(first, acc));
}
```

Finish designing the body of the `combineAcc` method. When done, run the original tests for the main methods as well as all of your other test cases.

Save the work you have done.