Methods and Data Abstraction

Matthew (one of your friendly TAs) has discovered a fantastic setting that we can use for our run configuration so it works for any source file. Edit your Eclipse run configuration, in the *Arguments* tab enter:

```
"${selected_resource_loc}"
```

Instead of the src file name. Now Eclipse will use FunJava to *"run"* the file/editor that you have selected.

4.1 Methods for Complex Hierarchies

Grab the `Employees.java` file and add it to your (or an) Eclipse project. Study the class diagram, the class definitions and the examples. Take a piece of paper and draw out the relationships of the examples. We call these classes *mutually recursive*, much like the `XExpr`s from last semester.

Complete the following problems. Make sure you update your templates as you implement various helper methods... things can get awfully complicated!

1. Design the method `countSubs` that computes the total number of subordinates of this `Emp`.
2. Design the method `fullUnit` that computes/collects all the subordinates of this `Emp`. *Hint*: you'll need to add a method to `append` a given `ILoE` to this `ILoE`.
3. Design the method `hasPeon` that determines if this `Emp` has a subordinate of the given name (`String`).

4.2 Abstracting over Data Definitions.

Designing Methods... Again.

As a simplification, imagine that a file on your computer can either be an `ImageFile`, `TextFile`, or `AudioFile`. Each class of data has a name and the owner, but there is additional information for each kind of file.

Download `Files.java` from the lab main page and work through the following problems.

1. Add an example of each of the three classes and add tests for the `size` method we've given you.

Design the following methods:

2. Design the method `downloadTime` that determines how many seconds it takes to *download* this file at a given download rate, in *bytes-per-second*.

3. Design the method `sameOwner` that determines whether the owner of this file is the same as the owner of the given file.

Abstracting Fields and Methods, and Methods

Look at the methods you've written and identify the places where your implementations are similar, and (wait for it...) *abstract*!!.

1. Lift the common fields to an **`abstract`** class `AFile`. Include a constructor in the abstract class and change the constructors in the sub-classes accordingly (remember **`super(...)`**?). After your *refactoring* the tests should run/pass exactly as before.

For each method defined in the three *concrete* classes decide which category it belongs in:

2. *The implementations in the sub-classes are all different*. For this case declare the method as **`abstract`** in the abstract class.

3. *The implementations in the sub-classes are all the same*. For this case you can implement the method completely in the abstract class. Other implementations can be removed.

4. *The implementations in the sub-classes are the same for some, but not all*. For this case you can move the common implementation into the abstract class, and *override* the the methods in the classes that require different implementation(s).

Move the methods that can be *lifted* (*abstracted*) and make sure all tests pass. *Note:* You can only lift the `sameOwner` method if you modify its contract. Do so, and adjust the types of your examples so that your tests have no compile errors.

4.3 A Bit of World Fun

Finally we do some interactive programming. Grab the `Follow.java` file from the lab page. For this problem you'll need to add the `JavaWorld-3.jar` (*Note the "3"*) library to your project and place it in your `EclipseJars` directory. See Lab 3 for directions and other links to set this up.

4.3.1 Simple Game

Your task is to create a little "*game*", where a little `Circle` (or `Star`... or be creative) follows a little target around the screen. When the target is reached motion stops. When the mouse is clicked the target is moved to where the click took place and motion begins again.

You need to design three methods: `toDraw`, `onTick`, and `onMouse`, with the signatures and purpose statements given in a comment. Be sure to *design* the methods, including examples and tests, before you code the method bodies. If methods get too complicated then design helpers where needed. Make your `Scene` 400 by 400.

To help you out, the `JavaWorld` library includes `placeImage` method that accepts a `Posn`, in addition to the one that takes **`int`**s:

```
// Place an Image on this Scene at the given Posn
Scene placeImage(Image i, Posn p);

// Place an Image on this Scene at the given X/Y
Scene placeImage(Image i, int x, int y);
```

We've also given you a `CartPt` class that **`extends`** `Posn`. Feel free to add any helper methods needed to this class, and since every `CartPt` is also a `Posn`, they can still be passed to `placeImage`.

4.3.2 Multiple Followers

Now that you've got the simple game working, make the following (no pun intended) additions.

1. Design an interface (and classes) to represent a list of `CartPt`s. As usual, call them `ILoCP`, `MtLoCP`, and `ConsLoCP`.

2. Design a method called `place` that places a dot (`Circle`, `Star`, or something special) in the given `Scene` for each `CartPt` in this list. *Hint*: think of the given `Scene` as an accumulator.

3. Design a method `moveToward` that returns a new list with each `CartPt` in the list moved toward the given `CartPt`. *Note*: if you were good about the design of your helper methods this is a piece of cake.

4. Modify your `Follow` class (the *world*) to have a list of locations (`CartPts`) instead of just one. Note that there's still only one *target*.

   You'll need to update your `onDraw`, `onTick`, `onMouse` methods, and your template, assuming you change **`this.`**`loc` to be **`this.`**`locs`. Feel free to start with the `MtLoCP`... we'll make it interesting in a second.

5. Finally, design an `onKey` method with the following signature:

   ```
   // Create a random List when a key is released
   Follow onKey(String ke);
   ```

   In your method, test the key-event (`ke`): if it is the string `"release"` create a new `Follow` with a random list of locations, otherwise return this `Follow` unchanged.

   To create a random `ILoCP`, put the following methods in your `Follow` class. Study it so you understand... it's just natural-number recursion, but notice how we create random integers.

   ```
   // Create a random integer
   int rand(){
      return new java.util.Random().nextInt(400);
   }
   // Create a list of random CartPts
   ILoCP random(int len){
      if(len == 0){
         return new MtLoCP();
      }else{
         return new ConsLoCP(new CartPt(this.rand(),
                                        this.rand()),
                                  this.random(len - 1));
      }
   }
   ```