Università
della
Svizzera
italiana

**Facoltà
di scienze
informatiche**

Programming Fundamentals 2 - Smalltalk **2004 - 2005**

Prof. Michele Lanza Paolo Bonzini, Mircea Lungu and Romain Robbes

Hands on 2 : collections, refactoring, streams 16/03/2005 @ 10:30

This hands-on session will tell you more about the collection classes. They are a very powerful abstraction and you should learn to use their methods.

It will also tell you about refactoring, which simply means reorganizing code so that it has a better structure and can be reused more easily. The VisualWorks environment includes very good tools for refactoring, that help you keeping your code tidy.

Finally, you'll have a short introduction to streams, another powerful abstraction in Smalltalk. Basically streams provide a way to create collections, especially strings like the `printString`. They are often used and in your project you'll use something akin to a stream to provide web content.

All these exercises will extend the `FMPlayer` and `FMPlayerPool` classes, that you should get from Moodle. Use a Workspace so that you can easily look at the values of the variables and inspect them. In the Workspace, remember to refer to the classes using the namespace, like `ProFund.FMPlayer`.

Exercise 1 - Collections

To get more practice with collection, we will add more methods like `potentialAttackers`.

1. First, see if you grasped `select:` and `reject:` Define a `potentialGoalkeepers` method, which returns the players with a dexterity higher than or equal to 75. Do not look at the code for `potentialAttackers`: instead, compare the two methods after you are finished.

   Sooner or later, it will happen that all substitutions have been made and a red card is showed to the goalkeeper. Surely you don't want to have a goalkeeper with a dexterity of 30, so we need a `replacementGoalkeepers` method. It should use `reject:` to filter away players whose dexterity is lower than 30.

2. To show how `collect:` is used, we will need a `tshirtNumber` field in `FMPlayer`. Add it together with the getter, `tshirtNumber`, and the setter, `tshirtNumber:`.

   Don't forget to automatically generate the T-shirt number of the players!

   Now, add a `tshirtNumbers` method to `FMPlayerPool` that collects all the T-shirt numbers and return them. `collect:` is like `map` in Scheme: it takes a one-argument block, and passes to the block all the objects in the collection, one at a time. Then it returns a new collection, made from the values that the block returned.

3. There are various kinds of collection. `Set` is a collection that stores unique values. You can convert a collection to a set with the `asSet` method that the system provides to you: the resulting object will have duplicates removed.

   Building a `Set` is often done to check if all the elements in a collection are unique. How can you do it?

   Hint : If duplicates are removed, the `Set`'s `size` will be lower than the original collection's...

   Use this trick to create a `numbersOkForGame` method in `FMPlayerPool`, that checks if the player in the pool have unique numbers.

4. There are also sorted collections. For now, we will use them on collections of number or strings—in general, objects that respond to messages like `<`, `<=` etc. If you have a collection of such objects, you can easily sort them using `asSortedCollection`.

   Hint : Puzzled? Try using `asSortedCollection` on an array of numbers in a workspace. See what happens if you try to sort together numbers and strings.

   Now, add a `numbersForGame` method. It will return the T-shirt numbers in sorted order if they are unique. Otherwise it will answer nil.

5. Finally, you may want to add methods such as `select:`, or `collect:`, to the `FMPlayerPool` directly! This way, the class will behave more like a collection.

   These methods will not do anything interesting: they will simply *delegate* their work to the `players` instance variable. You may want `at:` and `size`, for example. Try adding a `collect:` method to `FMPlayerPool`, and use it in `tshirtNumbers`.

Exercise 2 - Refactoring

As you saw in the last part of exercise 1, changes are seldom local to a single method. For example, after adding `collect:` to the player pool class, it was natural to use it elsewhere.

1. One of the most useful refactorings in VisualWorks is ”Extract method”. It takes a part of a method, and moves it to a newly created method.

   A natural place to apply this refactoring is `generateRandomPlayers`. The whole body of the loop can be extracted to a new method which we’ll call `addRandomPlayer`. Select with the mouse the body of the loop; in the code pane’s popup menu, pick ”Refactor” and then ”Extract method”.

2. Apply this refactoring again on the last line of `addRandomPlayer`. We’ll call the resulting method `addPlayer:`. VisualWorks finds out automatically that this method needs a parameter (the player it will put in the pool).

3. A more complex refactoring is ”Extract method to component”. Follow these steps one by one to refactor `potentialAttackers`, then do it again on `potentialGoalkeepers`.

   The goal is to move `each attack > 75` into a method of `FMPlayer`, called `isPotentialAttacker`. The steps are:

   a) select `each attack > 75`. Pick ”Extract method to component”.

   b) the player is in `each`, so you want to move the method into `each`.

   c) VisualWorks looks at the source code for your class and figures out that `each` must be an `FMPlayer`. Still, it asks you for confirmation: press Ok.

   d) Finally, VisualWorks asks you for the name of the method it will create. Type `isPotentialAttacker`.

Most other refactorings involve classes in an inheritance hierarchy. They are very useful, because they do work on many classes at once. You will learn more about them by practicing—all refactorings are found in the popup menus of the browser, and most are enabled only once you select a piece of code or an instance variable name.

Exercise 3 - Streams

Before we practice Streams, let’s take a step back and do one more exercise on collections.

Add a `preferredRole` method to the `FMPlayer` class. First it will pick the highest statistic among attack, defense and dexterity. If it is lower than 60, the method will return `#midfield`. Otherwise, the result will be `#attacker` if the highest statistic is attack; it will be `#defense` if the highest statistic is defense; it will be `#goalkeeper` if the highest statistic is dexterity.

Now use a method similar to `tshirtNumbers` to collect the preferred roles of the team. Call it `teamDistribution` and put it in `FMPlayerPool`.

We are only interested in the number of attackers, the number of goalkeepers etc. So, we are interested in the number of duplicates but we don’t want any order to be enforced on the items: in this case, what we want is a `Bag`. To have `teamDistribution` return a `Bag`, send `asBag` to the collection returned by `collect:`.

If you try to use `teamDistribution`, you’ll see that the output is not very tidy: Bags are most useful when you have thousands of duplicates of the same object, so it does not make much sense to print every object N times. An output like

```
Bag(#goalkeeper:2 #attacker:4 #midfield:3 #defense:2)
```

would be better. The output is quite complex, so overriding `printString` would not be very practical; instead we'll override `printOn:`, which prints an object's representation on a Stream.

Go to the Bag class (you can find it with the Hierarchy pane) and type this code:

```
printOn: aStream
    self class printOn: aStream.
    aStream nextPut: $(.
    self sortedCounts do: [:each |
        each value printOn: aStream.
        aStream nextPut: $:.
        each key printOn: aStream.
        aStream nextPutAll: ' ' ].
    aStream nextPut: $)
```

The method is quite complex, but you can see the basic methods:

- You can send `nextPut:` to the Stream if you want to print a single Character on it.
- You can send `nextPutAll:` to the Stream if you want to append a whole String to it. In this case I have used it to print a single space.
- You can use the polymorphic `printOn:` method to print other objects.

Actually, you should never override `printString` like we have done until now. The correct way to change the printed representation of an object is to override `printOn:`, and the `printString` will automatically adjust.

1. Reimplement your `printString` methods as `printOn:`. Do not forget to remove your overridden `printString` implementations.
2. Check that the inspector still prints your objects correctly.
3. Check that the `players` variable of an `FMPlayerPool` prints correctly. In case you had not noticed, previously it printed just "a ProFund.FMPlayer a ProFund.FMPlayer a ProFund.FMPlayer".
4. How can the `printString` automatically adjust to your `printOn:` methods? Look at how it is implemented in Object and try to figure this out.