APIT, Lab Book 1

Dr. Simon Rogers

Jan 2019

Introduction and aims

This lab book covers the material in the first few weeks of the course. Do the exercises at your own pace. Feel free to skip any that you feel are too easy!

The initial exercises allow you to clone the course git repository (the most up-to-date place to find code, slides, and notes) and give you some practice of compiling and running code from the command line.

Exercises 1: git and the command line

What you will do:

- Setup your own git repository and commit some changes
- Clone the course git repository
- Compile and run some Java code from the command line

Tasks

1. Setting up a new git repository
   - Open a command prompt, navigate to your home space, make a new folder and, from within it, type `git init`
   - Create a file in this folder (e.g. a text file)
   - Add the file to the repository (see slides)
   - Check the Status
   - Commit the changes
   - Make a new branch and within it, add a new file and commit.
   - Switch back to the original (`master`) branch. What happends to the folder?
2. Clone the course git repository
   - Create a folder in your home space in which you want the material to be stored
   - Clone the repo: `git clone https://github.com/sdrogers/APIT.git`
   - The repo will be updated throughout the course, to get the latest version: `git pull origin master`
3. Compiling Java on the command line
   - Open a text editor (e.g. notepad++) and write a short Java programme with a main method that prints `Hello World`. e.g.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

- Save it as `HelloWorld.java`
- Open a command prompt
- Navigate to the folder comtaining this file
- Compile the code `javac HelloWorld.java` (this will create a HelloWorld.class file)
- Run the code `java HelloWorld` (note `java` and not `javac` and no extension after the filename)

Exercises 2: Practice at building classes and class hierarchies

What you will do:

- Make your own interface
- Make your own abstract class
- Refactor classes to implement and extend these
- Play around with polymorphism

Tasks

1. Make your own abstract class and interface
   - In programming last semester, in AE2, you built one or two different Ciphers (mono and Vignere). Download my solution (or get it from `/labs/Ciphers.zip` in the repo) from here (note that this is trimmed down, and doesn't include any file handling): https://github.com/sdrogers/APIT/blob/master/labs/Ciphers.zip. The following code is an example of a `main` method that could test these classes:

```
public static void main(String[] args) {
    String keyword = "RHINO";
    try {
        Mono m = new Mono(keyword);
        String message = "THIS IS A MESSAGE TO ENCODE";
        String output = "";
        for(int i=0;i<message.length();i++) {
            output += m.encode(message.charAt(i));
        }
        System.out.println(output);
    }catch(BadKeywordException e) {
        e.printStackTrace();
    }
}
```

2. Design an interface that includes the basic methods that any cipher should implement. Create the interface and modify the provided classes so that they implement the interface. Just include the `char` versions of the `encode` and `decode` methods.
3. Modify the main method so that it creates an `ArrayList` of the type of the interface. Store two cipher objects (one Mono, one Vignere) in the interface and write a loop to encode the same message with each cipher in the `ArrayList`.
4. Design an `Abstract Class` for the type of Cipher that requires some kind of character array for encoding / decoding (e.g. ones like the Mono and Vignere). Write this class and then modify Mono and Vignere to extend it (they should continue to implement your interface). Your abstract class will have a mixture of abstract and non-abstract methods.
5. Modify `main` again so that the `ArrayList` is now the type of the `Abstract Class`. Satisfy yourself that you can use different methods depending on whether or not the `ArrayList` is the type of the `Interface` or the `Abstract Class`.

6. **Generics** Create your own dictionary object that maintains two ArrayLists of different (generic) types. It should have a `set` method which adds a key and value (i.e. one item to each list) and a `get` method which returns the value for a particular key. It should also have a `toString` method that displays the entire contents of the dictionary in a nice form. *Note: this is basically a copy of Java's* ***`HashMap`****.* Now suppose that you want to use this to store an address book. In your address book you have individuals and businesses and for each one you can store either a phone number or an address. Demonstrate how you can use your dictionary to accomplish this. Note you'll need to define some new interface / class hierarchy so that e.g. individuals are the same type as businesses and phone numbers can be stored as the same type as addresses.

Exercises 3: Iterators and design patterns

1. **Iterators** Make a class that includes an array of some user-defined length and then populates this with random numbers generated with:

```
Random r = new Random();
double value = r.nextDouble() - 0.5; // use this to generate a single value
```

2. Have your class implement `Iterable` so that it has a function `iterator()` that returns an iterator over its elements.
3. Modify your iterator from part 2 so that it only iterates over the **positive** values.
4. **Composite Pattern** You are building a computer operating system. Within your system you have files and directories. Directories can include other directories and/or files. All files and directories are within one top directory. The operating system needs to be able to do three things: a) Get the number of files inside any directory (including the number of files within sub-directories). Directories do not count as files. b) Get the total size of the files within a directory (the size of a directory is equal to the size of its contents). c) Display a directory and all of its contents as in the example below (the number in brackets is the total size for that directory). In this example, there are directories `root`, `pictures`, `music`, `jazz` and `classical` and various files. Directory contents should be indented. Tasks:
   - Sketch out the three components you will need to build (an `interface` and two concrete classes)
   - Create the three components. Remember to ensure that Directories should be able to include other directories.
   - Create another class with a main that demonstrates the system.

```
root (886)
    Settings (10)
    pictures (120)
        portrait (120)
    music (756)
        jazz (335)
            Kind of Blue (201)
            Giant Steps (134)
        classical (421)
            Beethoven, Symphony no 6 (421)
```

5. **Composite and Decorator** In the lectures I went through a shop example.
   - Start by implementing a slight variation on this using the `composite` pattern. The shop has items (leaves) and also sells groups of items (composites). Items have `name` and `price` attributes, composites just have a `name` attribute (their price is computed through looping over their children). Ignore the discount stuff in the lecture example for now. Composites should be able to include composites. All of the components in the system should implement a `compPrice()` method and `toString()` method.

- Now we will implement the discounts but using the `decorator` pattern. Make a `StudentDiscountDecorator` that discounts by 10% and a `StaffDiscountDecorator` that discounts by 50%. Some hints:
  - The highest level interface for the `decorator` you already have from the `composite`.
  - You will need an additional `abstract` class (e.g. `ShopComponentDecorator`) and then concrete `decorator` classes for the two `decorators`
  - It should be possible to `decorate` both leaves and composites!