Week 4 Lab

Enumerations, Inheritance, Collections, and I/O

Objectives

In this week's lab, you will:

- use proper Java coding standards
- implement an `Enum` type
- implement an abstract base class and concrete subclasses
- select and use a collection class
- implement console I/O
- design and implement your own solutions to satisfy requirements

You must first have completed all tasks from the previous week's lab.

You must draw a UML class diagram showing the system that you plan to implement *by the end of the lab*. Note that this system will incorporate work you have done in multiple tasks, so **read the entire lab sheet** carefully before commencing.

There are more decisions left to you when designing the system this week, including:

- deciding how to test the new Minion class hierarchy
- deciding which collection you will use to store the skills of your minions (so you will need to have done the set reading)

This diagram should be quick and simple. It only needs to show the classes you plan to create and the relationships (*associations and dependencies in UML*) between them. You don't need to show fields or methods. You are strongly encouraged to draw this diagram by hand — you don't need to use a UML diagramming tool.

You must get your design in UML approved by your demonstrator before you start coding.

Java Coding Standards

From this week onwards your marker will check that you have used proper Java coding standards before marking your work. These include conventions for naming classes, variables, and more. Make sure you have read the standards linked to from Moodle, and adhere to them in your code.

Task 1. Add an `enum` type for skills

This week, we need to make an enhancement to the system. We will need to keep track of our minions' skills so that we can make sure that our doomsday devices are operated by trained personnel.

There are some choices to be made about how to represent these skills. We could simply use Strings, such as "Psychology" or "Ballistics", but that has risks: it is very easy to make a typo, especially in words that are long or hard to spell. The Java compiler can't check that we have spelled consistently, and that means potential crashes or odd behaviour at runtime.

The Java language provides a safer mechanism for doing this using an `enum` type (short for "enumeration"). An `enum` specifies a list of named constants. Any variable of that type must have one of these constants as its value. This will mean that the compiler will ensure that your enums are correctly spelled: if you have declared a skill in your `enum` called `PSYCHOLOGY` then you won't be able to compile a method in which you've called it `PSYCOLOGY` – try it and see! Create an `enum` type called `MinionSkill` which has values that represent the various different skills (e.g. `SCUBA`, `PSYCHOLOGY`, `ROCKETRY`—yours may differ).

Give `Minion` an attribute that is a collection of type `MinionSkill`. You saw Java collections last week – consult the Week 3 readings list and decide for yourself which kind of collection will be most appropriate. You will need to be able to do the following things with your collection:

- add `MinionSkills` to the collection (note that you should not store duplicate skills – attempting to add a skill that is already in the collection should leave the collection unchanged)
- count how many `MinionSkills` the collection contains
- check (reasonably efficiently) whether a particular `MinionSkill` is stored in the collection

Add methods to your `Minion` class with the following signatures:

```
public void addSkill(MinionSkill skill)
```

and

```
public bool hasSkill(MinionSkill skill)
```

The method `addSkill(skill)` should add the given skill to the Minion's collection, and `hasSkill(skill)` should return `true` if the given skill appears in the Minion's collection and `false` if it does not. Do not add any other methods to Minion – at this stage, the analysts don't believe that we will be needing any other operations on skills.

Test your code before proceeding to make sure that your skills system works: add code to your `printStatus` method that adds skills to minions and then checks that correct values are returned when checking for them.

Task 2. Implement a Minion class hierarchy

It has become apparent that the system needs to represent different kinds of minions, with different properties. Supervillains have two kinds of employees: researchers, who devise new equipment and traps for the lair, and Support staff, who are responsible for day-to-day operations. Skills and training will work the same way for researchers and support staff, except for one important difference: they get paid at different rates.

All minions need to have the attributes and methods of the `Minion` created in earlier labs. A new method is needed too, that all `Minions` must have, but which will behave differently for different kinds of `Minion`. This method must have the signature:

```
public int monthlyPay();
```

Refactor your code so that `Minion` is an *abstract* class. All attributes and methods shared by all kinds of minion should be declared and defined there. The fact that all subclasses of `Minion` must have a `monthlyPay()` method must also be declared.

Create two concrete subclasses of `Minion`, one to represent researchers and one to represent support staff. In each subclass, implement a version of `monthlyPay()` that computes and returns the minion's monthly pay (in dollars) according to the following rules:

- Researchers earn a flat rate of $5000 per month if they have fewer than three skills, or $10000 per month if they have three or more skills
- Support staff earn a base rate of $3000 per month plus $500 per skill that they have

This means that if Natasha Fatale is a research minion who has mastered the skills `ESPIONAGE` and `PSYCHIATRY`, then `natasha.monthlyPay()` should return 5000. If she then adds another skill, such as `CRYPTOGRAPHY`, then `natasha.monthlyPay()` should return 10000. If Boris Badenov is a support minion who knows `ESPIONAGE` and `PSYCHIATRY`, then `boris.monthlyPay()` should return 4000. If he learns `CRYPTOGRAPHY`, then `boris.monthlyPay()` should return 4500.

Note:

Note: When you add your concrete subclasses of `Minion`, you will get error messages such as "`Implicit super constructor Minion() is undefined for default constructor. Must define an explicit constructor`". This is because when Java creates an object, it implicitly calls all of its superclass constructors. This is called *constructor chaining*.[1] If a subclass has no explicitly defined constructor, it implicitly has the Java default constructor, which has no arguments. It thus will try to call the argumentless constructor for the superclass — which for `Minion` does not exist. This means that you have to give subclasses explicit constructors with the same signatures as those in their superclasses that you want to use. This is admittedly a bit of a pain. Thankfully, your IDE should provide a mechanism to generate these for you automatically. In Eclipse, go to the `Source` menu and select `Generate Constructors from Superclass...` in the location where you want to do so.

Modify the code in `LairLocation` that creates minions and assigns them to locations so that it creates researchers and support staff. Give different skills to different minions and check that all methods are returning correct values.

Task 3. Display total monthly payroll in each LairLocation

Add a method to `LairLocation` that calculates and returns the total monthly payroll of all minions assigned to that location.

Modify the `displayLairLocation`[2]`(...)` method so that it displays the total monthly payroll required immediately after listing the minions assigned to it, using the method described above. The output for a service should look something like:

```
Welcome to the Supervillain's Lair Management System.

Shark Tank: Full of sharks with lasers on their heads
Assigned Minions:
12345678 Mini Me
12345679 Chum Berley
Total payroll: $8500

Boardroom: Where we plot world domination
Assigned Minions:
12345678 Mini Me
12345680 Donna Matrix
Total payroll: $15000

Janitorial: Cleans up the mess
```

[1] `https://docs.oracle.com/javase/tutorial/java/IandI/super.html`
[2] Or whatever you called it when you refactored it last week.

```
Assigned Minions:
12345666 Domestos McBleach
12103464 Pyne O'Kleen
Total payroll: $7000

Good-bye. Thank you for using the Supervillain's Lair Management System.
```

Task 4. Add console I/O

Many programs that you will write, including the assignments in this subject, will have a user interface of some kind that allows humans to interact with them. These days, most such programs use some kind of graphical user interface, which is often web-based.

However, in certain circumstances a *console-based* user interface are used. You will see a console-based user interface if you open the command prompt (or PowerShell) in Windows, or a terminal on a Mac or a Linux machine.

A console user interface allows you type something (e.g. a command), hit enter, and get a response from the computer — typically it would respond with some text output of its own.

Console-based user interfaces are usually less attractive and harder to learn than GUIs, but they:

- can be as fast or faster to use than a GUI for experienced users,
- can be used over a slow or heavily loaded network connection,
- can be easily *scripted* with another computer program taking the place of the human typing the input, and importantly,
- are much easier to write than a GUI or web interface.

There are several ways to implement a console user interface in Java. Unfortunately, the simplest method, the `Console` class,[3] is incompatible with the IDEs where the output displays in an IDE window when you run your program from within the IDE.

You have seen how to use `println` to produce console output. Below is a Java method you can use to read console input. The argument, `prompt`, is displayed on the line to the left of where you type any input. If you don't want a prompt, pass in an empty string.

```java
private String readString(String prompt) {
    System.out.print(prompt);
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in)
        );
    String s = null;
    try {
        s = in.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return s;
}
```

Modify `LairLocation` so that when each `Minion` object is created, the user is prompted to enter the minion's name and ID. Read this information from the command line and use it when creating the `Minion` object.

[3]https://docs.oracle.com/javase/tutorial/essential/io/cl.html

Feel free to change the type of the data structure in `LairLocation` from `array` at this stage (e.g. to `ArrayList`). You can also use data structures of your choice for the variables that keep track of the `Minion` objects in the method where they are created and assigned.

Getting marked

Once you have completed these tasks, call your lab demonstrator to be marked. Your demonstrator will ask you to do a "walkthrough" of your program, in which you explain what each part does. Note that to receive full marks, **it is not sufficient merely to produce correct output**. Your solution must implement all structures specified above *exactly* — as software engineers we must follow specifications precisely. If you have not done so, your demonstrator will ask you to fix your program so that it does (if you want to get full marks).

If you do not get this finished by the end of this week's lab, you can complete it during the following week and get it marked at the start of your next lab — but make sure that your demonstrator has at least given you feedback on your design before you leave, so that you don't waste time implementing a poor or incorrect design. That will be the last chance to get marks for the exercise. You cannot get marked for an exercise more than one week after the lab for which it was set.

Marking

This lab is worth 2% of your final mark for FIT2099. Marks will be granted as follows:

- 0 marks if you do not attempt the tasks, or if you could only complete them once given the solution
- 1 mark if some tasks are incomplete, if the design or the implementation is flawed, or if you needed to see a significant part of the solution to complete them
- 2 marks if you were able to complete all tasks independently, with good design and correct implementation