CAB302 Software Development

Practical 2

Object Orientation and API Documentation

This week's practical exercises aim to get you familiar with developing object-oriented class hierarchies in Java and documenting them using Javadoc in the Eclipse integrated development environment.

Exercise 1: Footy scores

This exercise will give you experience at developing a simple (concrete) *class* in Java. In Australian Rules Football, teams earns *points* by kicking *goals*, worth six points each, and *behinds*, worth one point each. Your job is to develop a class called `FootyScore` which keeps track of the score earned by a particular team. This class needs to:

- declare a private data structure sufficient to keep track of the team's score;
- declare a parameterless public method called `getPoints` that returns the team's total score in points;
- declare a parameterless public method called `kickGoal` that records the fact that the team has kicked a goal;
- declare a parameterless public method called `kickBehind` that records the fact that the team has kicked a behind;
- declare a parameterless public method called `sayScore` that returns a character string representing the way Australian Football League commentators traditionally say AFL scores, as three numbers, consisting of the number of goals kicked, the number of behinds kicked, and the total number of points earned, in that order; and
- declare a predicate (boolean-valued function) called `inFrontOf` which accepts a `FootyScore` object as its parameter and returns true if and only if (iff) this team's score exceeds that of the team provided as an argument.

To help you assess your solution the file `FootyTester.java` contains a 'main' program that simulates a (rather low scoring) AFL match between traditional rivals Collingwood and Richmond. It prints the breathless commentary of a typically rabid AFL announcer. If your solution works correctly, the test program will print the following commentary on the console. Note that your `sayScore` method should return the three numbers separated by a comma and a space.

```
At the end of the first quarter, it's Collingwood, 0, 2, 2 and Richmond 1, 0, 6.
Richmond lead by 4 points!
At the end of the second quarter, it's Collingwood, 2, 3, 15 and Richmond 3, 0, 18.
Richmond lead by 3 points!
At the end of the third quarter, it's Collingwood, 3, 3, 21 and Richmond 3, 2, 20.
Collingwood lead by 1 point!
```

(We leave the final score in doubt to avoid offending any Collingwood or Richmond supporters in the class.)

Exercise 2: Documenting your footy score class

Use the Javadoc tool to generate a Hyper-Text Markup Language file that describes your `FootyScore` class. Ensure that the purpose of the class, its methods, and their parameters are all clearly described in the HTML documentation by including appropriate comments and annotations in the Java source code. (Enclosed with these instructions we've included an example showing what the result should look like. To see it, open the relevant `index.html` file in your favourite HTML browser.)

Exercise 3: A superhero class hierarchy

This exercise will give you experience at interpreting Application Programming Interface documentation and developing non-trivial class hierarchies, including a Java *interface*, *abstract class* and *concrete classes*.

Comic book superheroes remain in vogue thanks to the financial success of the recent Marvel movies. In this exercise you will develop a type hierarchy that models the characteristics of comic book superheroes. (But for testing purposes we will use 'Golden Age' superheroes from the 1940s and 1950s as these are the ones most familiar to your elderly lecturers.)

Enclosed with these instructions you will find the API documentation for a 'superhero' class hierarchy, along with two Java files:

- Enumerated type `SuperPower.java` which lists the superpowers available to Golden Age superheroes.
- A 'main' program `HeroTester.java` which will work fully only when you have completed the exercise. You will probably want to comment out some of the tests during code development. We suggest you begin with just the

tests related to mortal superheroes uncommented. Once you have got this class working, uncomment some of the other tests.

Included in this documentation is the role of each class and the intended purpose of its members, as well as brief hints about intended usage.

Your task is to study the API documentation and, based on this, to implement the missing classes, `Hero.java`, `SuperHero.java`, `Human.java`, `EnhancedHuman.java` and `SuperHuman.java`. These classes form a hierarchy as shown overleaf. Make sure you read *all* the API documentation and clearly understand how the various classes and their methods relate to one another *before* you start writing code. Also, try to make good use of Eclipse's "quick fix" features to help you develop the code quickly. For instance, when creating a subclass you can get Eclipse to automatically declare all the inherited methods that must be implemented.

(**Comment for nitpickers**: The particular choices of superpowers used in test program `HeroTester.java` are somewhat debatable. For instance, Captain Marvel is usually described as 'swift as Mercury' but we have not given him the power of 'super speed' because he seems to be nowhere near as fast as characters who specialise in speed, such as The Flash. Similarly, we have not ascribed the power of flight to Wonder Woman because in her Golden Age incarnation she needed her invisible plane to fly. More recently, however, she has acquired the ability to 'ride the winds' as if flying. In summary, the superpowers used in `HeroTester.java` are for illustrative purposes only. No correspondence will be entered into.)

Exercise 4: Adding secret agents as heroes

So far the `Hero` interface doesn't appear to be especially helpful. However, it serves an important purpose because it gives us flexibility in extending the type hierarchy. For instance, we can add a (concrete) `SecretAgent` class below it, without affecting the superhero classes in any way.

Not only do secret agents have an alias (when on a mission James Bond is known as '007' and Maxwell Smart as 'Agent 86'), but they are frequently associated with gadgets (like Max's shoephone). Therefore, your `SecretAgent` class should inherit the features of the `Hero` class and add an attribute which stores the gadget typically associated with this agent (as a text string). Once again, consult the API documentation provided for the precise specification. Use the `AgentTester.java` program to test your class.

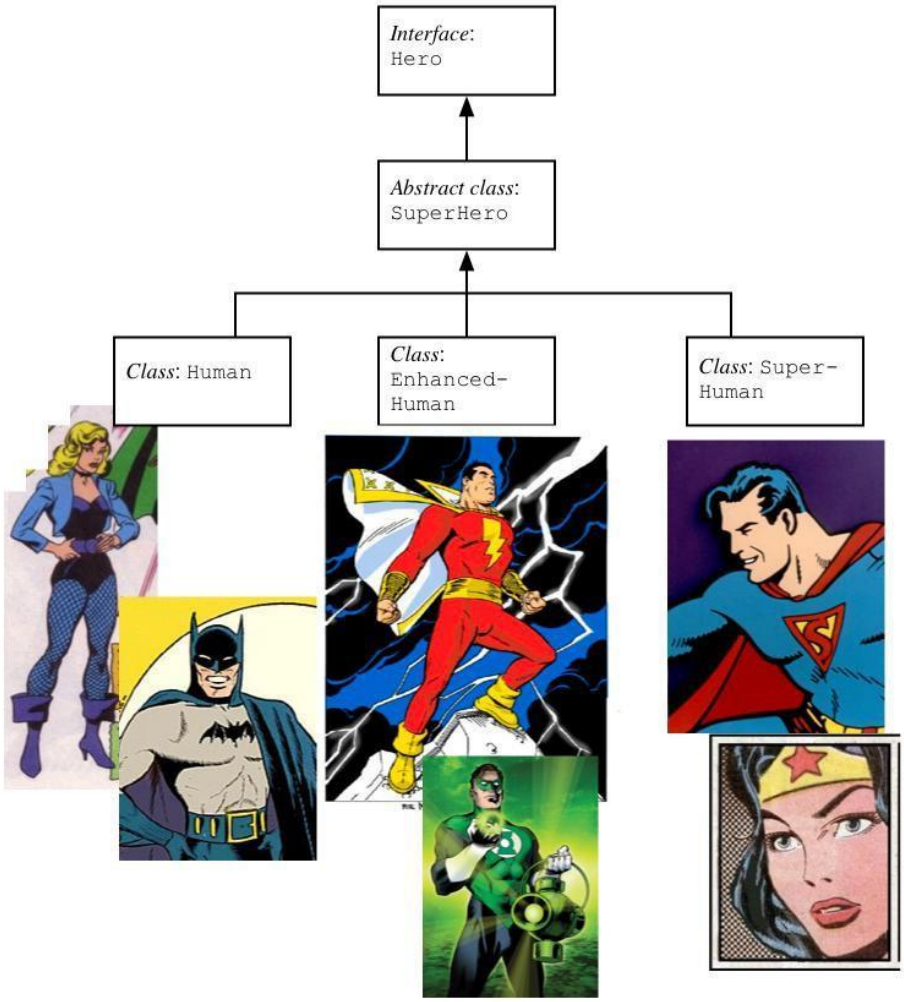

Figure 1: Hero class hierarchy

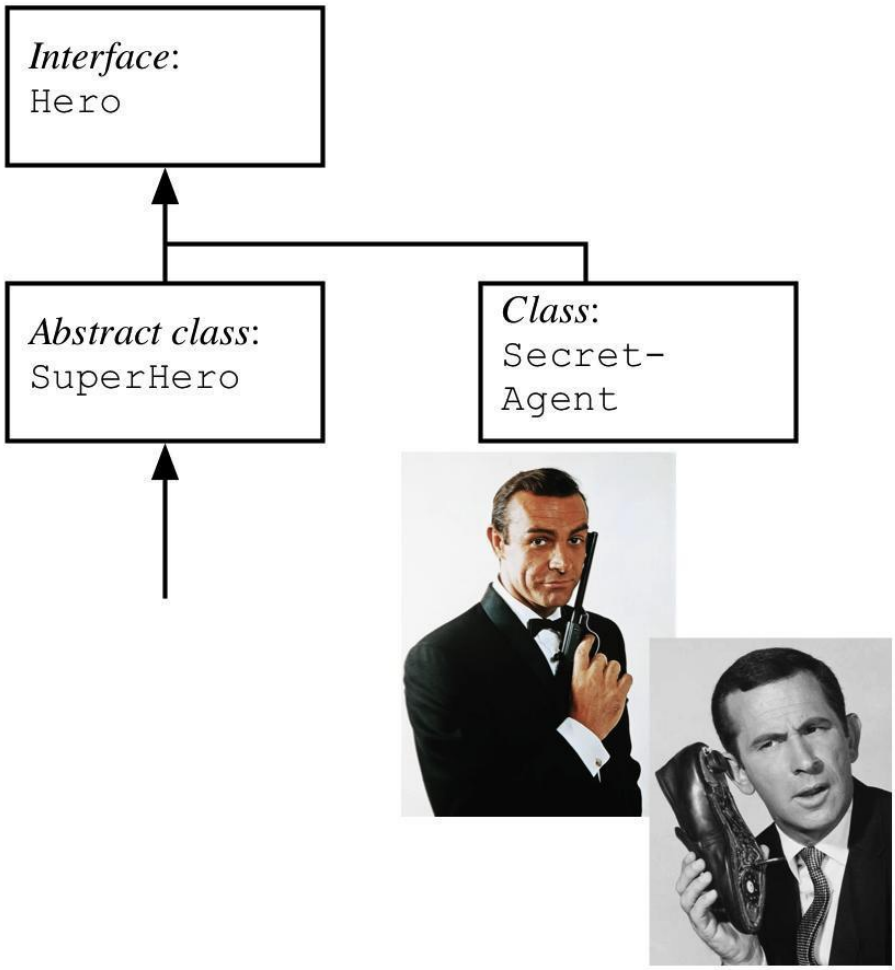

Figure 2: Hero class heirarchy extended with SecretAgent

Exercise 5: Documenting your hero class hierarchy

Finally, use the Javadoc tool to generate your own API documentation for your entire `Hero` class hierarchy, including all of its subclasses. If successful the output should look the same as the one we provided!