Advanced Programming

Lab Sheet 10 – Programming to Interface

Introduction

Interfaces are a powerful feature of Object-Oriented programming languages, but it can sometimes be difficult to get your head round what makes them useful when first learning about them. One common scenario where interfaces shine is when you wish to allow other developers to implement their own classes in the future, which you'll call from inside your code, similar to a plugin engine. In this lab, you will write some classes that implement an interface I have already written for you, to make it easy to integrate your work into an existing program I have made.

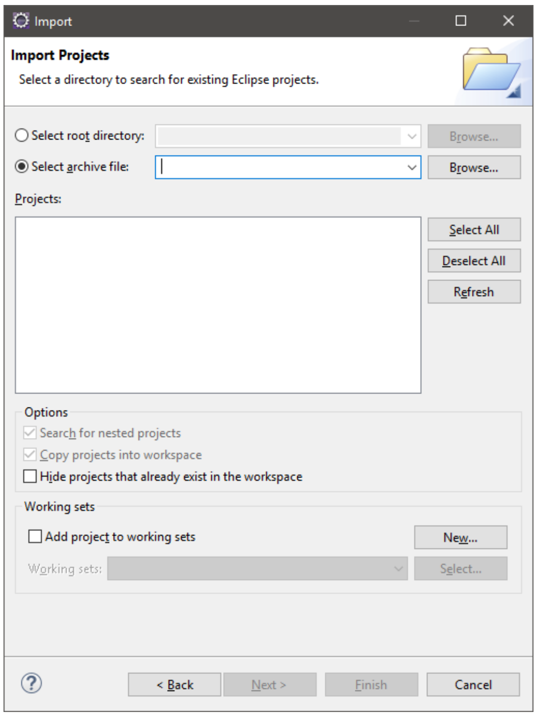
The term "Programming to (an) interface" refers to the practice of developers working together on larger projects agreeing the interfaces that their code will provide in advance, so that they each know what methods they will be able to call, and for what. This lab serves as a simple example of this practice, with the interface to which your code will adhere being pre-defined.

You will be working on a program that simulates a simplified version of the card game *blackjack*, allowing for large numbers of games to be played rapidly and for the efficacy of different player strategies to be evaluated. Blackjack is a simple card game, in which players draw cards in an attempt to obtain a hand that scores as close as possible to 21, without going over (termed "going bust"). Players are initially given a hand of two cards, and can choose to *hit* (draw another) or *stick* (keep what they have), repeatedly until they either choose to stick, or go bust. Cards are valued at the number depicted upon them, with Jacks, Queens and Kings valued at 10. Ace cards can be valued at either 1 or 11, at the player's discretion. The winner of each game is the player with the highest score without going bust.

I have created an initial version of the program for you. In this initial version, all players adopt a simple "stick at 17 or higher" strategy, which is commonly required of casino dealers. You will create and try out several different strategies, which requires you to write classes implementing a specific interface and to make a minor tweak to the existing code to have a player use your new class.

Importing the Code

The initial version of the program is available to download on Moodle as an eclipse project, exported as a .zip file. You will need to import the project into eclipse to start work. You can import it into your eclipse workspace by selecting File → Import, and then selecting "Existing Projects Into Workspace". You will see a dialog similar to that depicted in Fig. 1, below.



Import Projects

Select a directory to search for existing Eclipse projects.

○ Select root directory:

● Select archive file:

Projects:

Options

☑ Search for nested projects

☑ Copy projects into workspace

☐ Hide projects that already exist in the workspace

Working sets

☐ Add project to working sets

Working sets:

Figure 1: Eclipse Import Projects Dialog

On this dialog, you should make sure "Select archive file" is selected, then browse to the zip you downloaded from Moodle. Make sure that the "BlackJackSim" project is ticked, before clicking the "Finish" button at the bottom of the dialog. You will see the project in the left-hand panel in eclipse, along with your previous projects.

The BlackJackSim project has a number of classes split across two packages. The *blackjacksim* package contains the classes that simulate the games and collect the results. The *blackjacksim.strategy* package contains the classes that simulates the player strategies. All of the classes have JavaDoc and inline comments to help you to understand the existing code.

Task 1: Understanding The Existing Code

The main entry point to the program is the BlackJackSim class's main method. The BlackJackSim class has two constants defined at the top of the class that allow you to tweak the program's behaviour, as depicted below.

```java
private static final int GAMES_TO_RUN = 1000000;
private static final int UPDATES_EVERY = 100000;
```

The *GAMES_ TO_ RUN* constant defines how many games should be simulated when you run the program. If you're working on a slow machine, you might want to decrease this value to 500k to speed things up. Realistically, though, the program should only take a few seconds to run on modern hardware.

The *UPDATES_ EVERY* constant defines how often the program should print out a summary of player performance. If you are working on a very fast machine, it might be worth increasing this value so that you get a chance to read the output before more is printed.

The BlackJackSim code in the main method uses an integer array to track the wins of each player, along with a separate integer to track the number of drawn games. The ordering of the integers in the array corresponds to the order of the players in an array defined in the Game class, as depicted below.

```java
static final Player[] PLAYERS = {
    new Player(1,new StickAtSeventeenStrategy()),
    new Player(2,new StickAtSeventeenStrategy()),
};
```

This *PLAYERS* constant array defines the players involved in the simulated games. You can tweak the contents of this array to change the number of players in the games, to adjust their player ID in the program's output, and adjust the strategy that each player will adopt. As you can see in the snippet above, by default both players use the same strategy.

The PlayerStrategy interface, which is in the *blackjacksim.strategy* package, is the interface that the player strategy classes must implement. The Player class has several instance variable and method parameters typed as PlayerStrategy, which means that the existing program's code doesn't need to change if another strategy is substituted for the StickAtSeventeenStrategy. This is the principal benefit of polymorphism.

The Card and Deck classes, as the names suggest, model the deck of cards used in the game. There are a few tricks in these classes you won't have seen before: randomising an ArrayList, using a pointer to implement a queue-like data structure so that draw() will work. There are even some enums and a switch to refresh your memory from last year.

Take some time to read through the existing code and build some understanding. The main bulk of the work in this lab isn't likely to be too time consuming, so there's no rush to get to it.

Task 2: Simulating Different Player Strategies

You should implement each of the following player strategies and evaluate its performance against the standard StickAtSeventeen strategy. Do any of them perform favourably?

- Stick at sixteen
- Stick at eighteen
- Hit soft seventeen (Stick at Seventeen, unless one of the existing cards is an ace)
- Always stick
- Stick at three cards

Implementing each of these strategies will require you to create a new class that implements the PlayerStrategy interface. None will be particularly difficult, because they're all fairly similar to the existing StickAtSeventeen class.

You can evaluate each of the implemented strategies by doing a simulation run of a two player game, with one player running the existing StickAtSeventeen strategy and one player running your new creation. To do this, you will need to modify the PLAYERS array in the Game class, but no other code will need changing.

Task 3: Simulating Games With More Players

You've seen how the strategies fared when pitted one-on-one against a standard "stick at seventeen" strategy, but how do they fare against each other? Try running some three-way simulations to see how the various different combinations of strategy do. Can you identify a ranking from strongest to weakest? Does the ranking change if the games are four-way?

Extension Tasks

There are *a lot* of different ways that you can take this program, and I'd be interested to see what you can come up with given the tasks themselves were fairly light. Some suggestions include:

- In real games of blackjack, players can see the first card drawn to other players, and this influences their choice of strategy. If a player has a hand of sixteen and knows the other player has a two, they may choose to stick and hope the other player goes bust. You could modify the simulation so players have this knowledge, and develop strategies that use it.
- You could track the financial performance of player strategies if they were playing in a casino and gambling. How much money would each of them lose over time? If a player hits a winning streak and always quits at a set threshold, how many would finish ahead?
- Did you see the method that notifies the strategy when the deck is shuffled? You could implement a basic form of card counting, tracking how many high and low cards have been seen since the deck was shuffled, and adjust the strategy based on the adjusted likelihood of going bust. In real casinos, they deal from several decks of cards shuffled together to make this less powerful, there's a nice documentary on it here if you'd like some bedtime viewing.