Lab Exercise 7: Generics, Lists and Stacks
CS 2334

October 6, 2016

Introduction

In this lab, you will experiment with the use of generics. Generics allow you to abstract over types. More specifically, they allow types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. You will create a card game that makes use of a generic Deck class to create three decks of cards of two different types: one type is based on the Characters on the card, and the other type is based on the Fate shown on the card. Although we will have two distinct types of Decks, both will need a standard set of operations (including shuffling and drawing). By using Java generics to implement a generic Deck class, we only need to implement these methods once and the implementation will work for many different types of card decks.

As part of the Deck implementation, we will also make use of the Stack class from the Java Collections Framework to store stacks of used and unused cards.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Create classes using generics in Java
2. Use generic classes to solve larger problems
3. Use the Stack class to store and retrieve objects

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Import the existing lab7 implementation into your eclipse workspace.

   (a) Download the lab7 implementation:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab7/lab7.zip`

   (b) In Eclipse, select *File/Import*

   (c) Select *General/Existing projects into workspace*. Click *Next*

   (d) Select *Select archive file*. Browse to the lab7.zip file. Click *Finish*

The Fate Card Game

The game has three decks of cards: an Emperor's Deck, a Plebeian's Deck, and a Fate Deck. To play the game:

- The player draws one card from the Fate Deck.
  - If the player drew a Riches Card, they will receive the Emperor's Deck, and the opponent the Plebeian's Deck.
  - If the player drew a Revolution Card, they will receive the Plebeian's Deck, and the opponent the Emperor's Deck.
- Each player then draws the top card from their deck and plays it face up.
  - If both cards are a Patrician, the outcome is a tie. Both players then draw the next card on top of their respective decks, repeating until the cards are different.
  - If an Emperor and a Plebeian are played against each other, the Plebeian wins and the owner of the Plebeian card wins 4 points.
  - If an Emperor is played against a Patrician, the Emperor wins and the owner of the Emperor card wins 1 point.

  - If a Plebeian is played against a Patrician, the Patrician wins and the owner of the Patrician card wins 1 point.

- If there are more games to be played, all decks are shuffled and the player restarts by drawing a Fate card.

The full implementation of our program will first create one Emperor, one Plebeian, and one Fate Deck, and then shuffle all three. Your program will then play our game 50 times, reporting the result of each play to the console (you are free to change this number inside of the Driver class to a larger number, if desired).

Class Design

Below is the UML representation of the set of classes that make up the implementation for this lab. Note that we have introduced a couple of new bits of notation:

- We are explicitly representing our generic classes with the generic type undefined **AND** our generic class with the generic type bound to some other type (e.g., `Card<T>` vs `Card<Character>`).

- Although we will explicitly implement the generic class (`Card<T>`), we don't provide an explicit implementation of the bound class (`Card<Character>`) – the compiler does this for us! Hence, the bound class is labeled as "`<<implicit>>`".

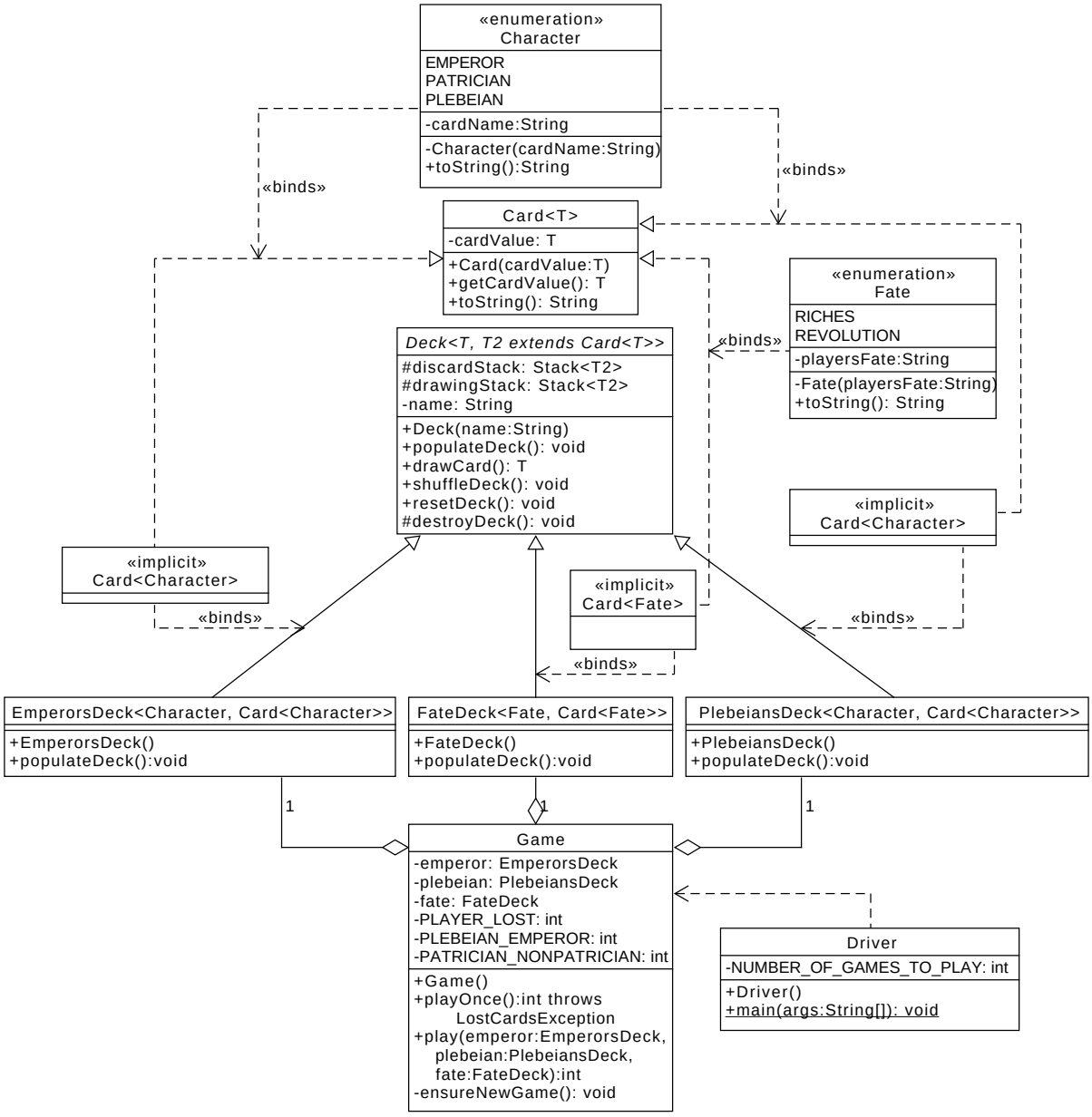
- The lines labeled "`<<binds>>`" tell us explicitly what the binding is for our generic type.

The key classes in our project are:

- **Character** is an enumerated data type that defines a set of characters. Namely, the Emperor, Patrician, and Plebeian.

- **Fate** is an enumerated data type that defines a set out deck assignments . Namely, the Revolution card and the Riches card.

- The **Card** generic class is defined by a generic type, **T**. The two arrows from `Card<Character>` and `Card<Fate>` to **Card** indicate that the generic type of **Card** is bound in two different ways: one with **Character** and the other with **Fate**.


«enumeration»
Character
EMPEROR
PATRICIAN
PLEBEIAN
-cardName:String
-Character(cardName:String)
+toString():String
«binds»
«binds»
Card<T>
-cardValue: T
+Card(cardValue:T)
+getCardValue(): T
+toString(): String
«enumeration»
Fate
RICHES
REVOLUTION
-playersFate:String
-Fate(playersFate:String)
+toString(): String
«binds»
Deck<T, T2 extends Card<T>>
#discardStack: Stack<T2>
#drawingStack: Stack<T2>
-name: String
+Deck(name:String)
+populateDeck(): void
+drawCard(): T
+shuffleDeck(): void
+resetDeck(): void
#destroyDeck(): void
«implicit»
Card<Character>
«implicit»
Card<Character>
«implicit»
Card<Fate>
«binds»
«binds»
«binds»
EmperorsDeck<Character, Card<Character>>
+EmperorsDeck()
+populateDeck():void
FateDeck<Fate, Card<Fate>>
+FateDeck()
+populateDeck():void
PlebeiansDeck<Character, Card<Character>>
+PlebeiansDeck()
+populateDeck():void
1
1
1
Game
-emperor: EmperorsDeck
-plebeian: PlebeiansDeck
-fate: FateDeck
-PLAYER_LOST: int
-PLEBEIAN_EMPEROR: int
-PATRICIAN_NONPATRICIAN: int
+Game()
+playOnce():int throws
LostCardsException
+play(emperor:EmperorsDeck,
plebeian:PlebeiansDeck,
fate:FateDeck):int
-ensureNewGame(): void
Driver
-NUMBER_OF_GAMES_TO_PLAY: int
+Driver()
+main(args:String[]): void

- **Deck** is an abstract class that is defined by a pair of generic types: **T** and **T2**. **T** defines the underlying generic type of **Card** (in our case, **Character** or **Fate**) and **T2** is a form of **Card** (specifically, **T2** is-a `Card<T>`).
- The lines from **EmperorsDeck**, **PlebeiansDeck**, and **FateDeck** to **Deck** indicate that **EmperorsDeck**, **PlebeiansDeck**, and **FateDeck** extend **Deck**. Note that each of these subclasses also explicitly binds **T** and **T2** to particular Card types. For example, the **FateDeck** explicitly binds **Fate** to **T** and `Card<T>` to `Card<Fate>`.

Lab 7: Implementation Steps

Start from the class files that are provided in lab7.zip.

1. The classes **Card**, **Character**, **PlebiansDeck**, and the **Driver** have been fully implemented and **should not be modified**.
2. The class **Deck** is a generic and abstract class. This class is partially implemented for you. You will need to complete the implementation of each method that is listed in the UML.
3. Create and implement the enumeration class **Fate**. Each enum has a single string associated with which fate it depicts. *RICHES* means that the player's fate is "You are the Emperor". *REVOLUTION* means that the player's fate is "You are the Plebeian". Each Fate enum's string representation should simply return the player's fate.
4. Create and implement the class **FateDeck**. The name of this deck should be stored as "The Deck of Fate". The method *populateDeck()* should first destroy any deck it made previously, and then populate the deck with one card of type *RICHES* and one of type *REVOLUTION*.
5. Create and implement the class **EmperorsDeck**. The name of this deck should be stored as "Emperor's Deck". The method *populateDeck()* should first destroy any deck it made previously, and then populate the deck by placing a full set of cards into the deck. Specifically, this method should place 4 cards of type *PATRICIAN* and one of type *EMPEROR*.
6. The class **Game** is partially implemented for you. You will need to complete the implementation for the method *play()*. *play()* plays one game (as described

in the Fate Card Game section above) given an EmperorsDeck, a PlebeianDeck, and a FateDeck and reports the amount of points the player received. *These point values must utilize the constants created in the Game class, as well as the constPlayerLost modifier*:

- If the two cards played are a Plebeian/Emperor combination, return the constPlebeianEmperor constant multiplied by the constPlayerLost modifier if the player lost.
- If the two cards played are a Patrician/NonPatrician combination, return the constPatricianNonPatrician constant multiplied by the constPlayerLost modifier if the player lost.

7. Implement JUnit tests to thoroughly test all classes and methods you created/implemented.
    - You need to convince yourself that everything is working properly
    - Make sure that you cover all of the cases within the methods while creating your tests. Keep in mind that we have our own tests that we will use for grading.

Example Output

Below is an example output of the full program, playing a total of 6 games and then reporting the total score for the player at the end. The details of your games will vary.

```
////////// FATE GAME START //////////
Player's Current Total Score: 0
///// You are the Plebeian /////
You: Patrician
Opp: Patrician
You: Patrician
Opp: Patrician
You: Patrician
Opp: Patrician
You: Patrician
Opp: Patrician
You: Plebeian
Opp: Emperor
Player's Current Total Score: 4
///// You are the Plebeian /////
You: Plebeian
Opp: Emperor
Player's Current Total Score: 8
///// You are the Emperor /////
You: Emperor
Opp: Patrician
Player's Current Total Score: 9
///// You are the Plebeian /////
You: Patrician
Opp: Patrician
You: Patrician
Opp: Patrician
You: Patrician
Opp: Emperor
Player's Current Total Score: 8
///// You are the Emperor /////
You: Patrician
Opp: Patrician
You: Emperor
Opp: Patrician
Player's Current Total Score: 9
///// You are the Plebeian /////
You: Patrician
Opp: Patrician
You: Patrician
Opp: Patrician
You: Patrician
Opp: Patrician
You: Patrician
Opp: Emperor
////////// FATE GAME END //////////
Player's Final Score: 8
```

Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your project (and all classes within it) is selected
   - Select *Private* visibility
   - Use the default destination folder
   - Click *Finish*.
2. Open the *lab7/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that all of your classes are listed and that all of your documented methods have the necessary documentation.
3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission Instructions

- All required components (source code and compiled documentation) are due at 11:59:00pm on Monday, October 10th.
- Method 1: Submit through Eclipse
   1. From the *Window* menu, select *Preferences/Configured Assignment*.
   2. Select your project.
   3. From the Project menu, select *Submit Assignment*.
   4. Under *Select the assignment to submit*, select *Lab 7: Lab 7: Generics, Lists, and Stacks*.
   5. Click *Change Username or Password....* Enter your Web-Cat username and password. Click *OK*. You should only need to do this step once per session.
   6. Click *Finish*.

7. Your browser should automatically open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

- Method 2: Submit directly to the Web-Cat server

1. From the File menu, select *Export*.
2. Select *Java/JAR File*. Click *Next*.
3. Select and expand your project folder.
4. Select your *src* and *doc* folders.
5. Select *Export Java source files and resources*.
6. Select an export destination location (e.g., your *Documents* folder/directory). This file should end in *.jar*
7. Select *Add directory entries*.
8. Click *Finish*.
9. In your web browser, login to the Web-Cat server.
10. Click the *Submit* button.
11. Browse to your jar file.
12. Click the *Upload Submission* button.
13. The next page will give you a list of all files that you are uploading. If you selected the correct jar file, then click the *Confirm* button.
14. Your browser will then open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests)

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code (up to 15 points)

If you do not submit compiled Javadoc for your lab, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions (where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every two hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late.