# Lab Exercise 7: Generics, Lists and Stacks
# CS 2334

October 5, 2017

## 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 Colors 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. Create and use *enumerated data types*
3. Use generic classes to solve larger problems
4. 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 Card Game

The game has three decks of cards:

- Green Deck: contains one green card and four black cards.
- Red Deck: contains one red card and four black cards.
- Fate Deck: contains one *Riches* card and one *Revolution* card.

Each of these decks consists of two stacks: the drawing stack is the source of newly drawn cards; the discard stack contains cards that have already been drawn. The set of cards in a deck (between these two stacks) stays constant throughout the game.

To play one instance of a game:

- All three decks are shuffled.
- The player draws one card from the Fate Deck.
  - If the player drew a Riches Card, they will receive the Green Deck, and the opponent the Red Deck.

  - If the player drew a Revolution Card, they will receive the Red Deck, and the opponent the Green Deck.

- The player and the opponent then draw the top card from their own decks and play them face up.
  - If both cards are Black, 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 a Green card and a Red card are played against each other, the Red card wins and the owner of the Red card wins 4 points.
  - If a Green card is played against a Black card, the Green wins and the owner of the Green card wins 1 point.
  - If a Red card is played against a Black card, the Black card wins and the owner of the Black card wins 1 point.

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

The full implementation of our program will first create one Green, one Red, and one Fate Deck, and then shuffle all three. Your program will then play our game five 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

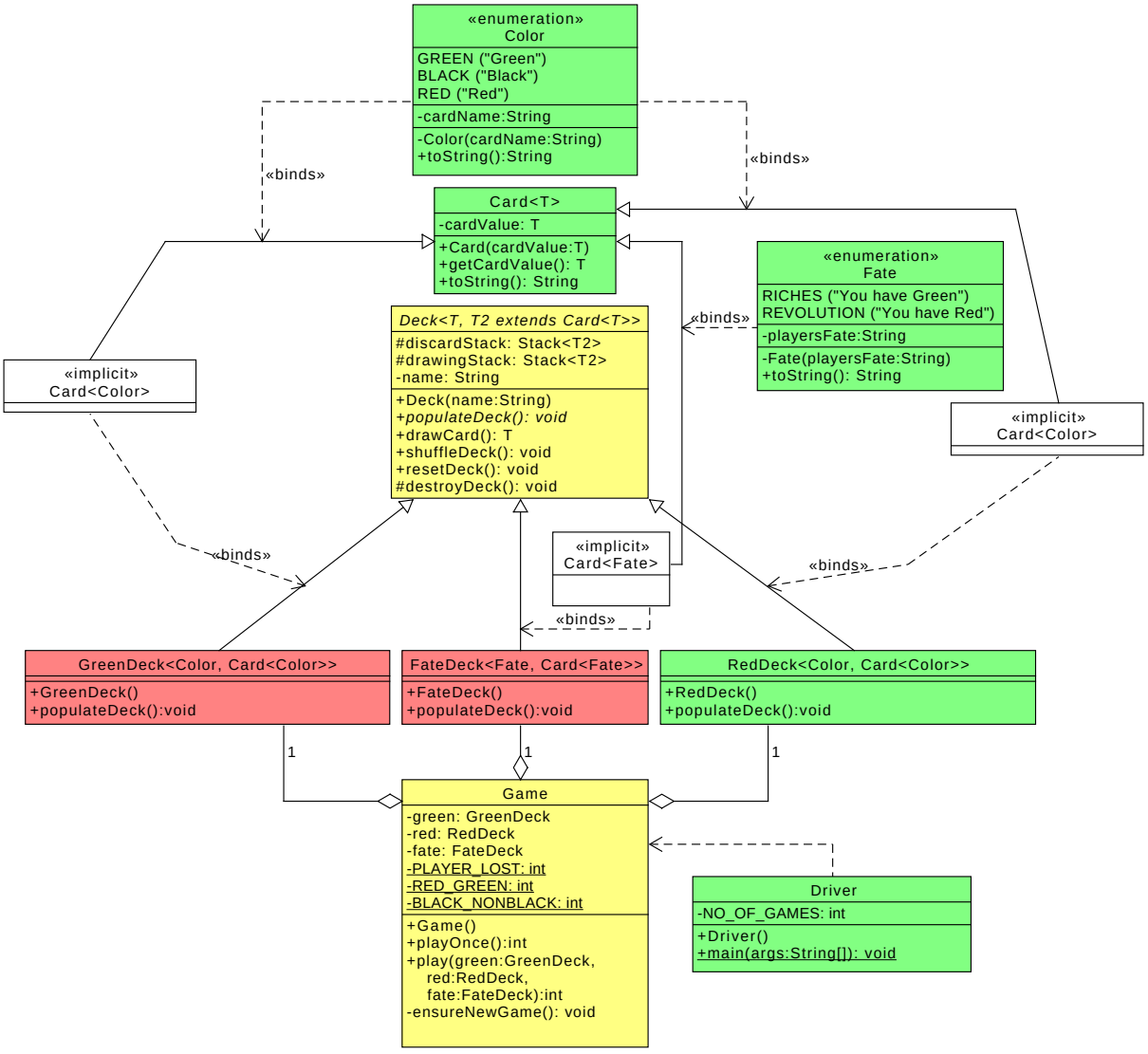
Below is the UML representation of the set of classes that make up the implementation for this lab. Note that we have introduced a couple of new bits of notation:

- 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<Color>`).

- Although we will explicitly implement the generic class (`Card<T>`), we don't provide an explicit implementation of the bound class (`Card<Color>`) – 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.

Green colored blocks indicate that they are fully implemented.
Yellow colored blocks indicate that they are partially implemented.
Red colored blocks indicate that they have to be implemented by you.



The key classes in our project are:

- **Color** is an enumerated data type that defines a set of colors: GREEN, BLACK and RED.
- **Fate** is an enumerated data type that defines the deck assignment during the game. There are two types of Fate cards: REVOLUTION and RICHES.
- The **Card** generic class is defined by a generic type, **T**. The two arrows from `Card<Color>` and `Card<Fate>` to **Card** indicate that the generic type of **Card** is bound in two different ways: one with **Color** and the other with **Fate**.
- **Deck** is an abstract class that is defined by a pair of generic types: **T** and **T2**. **T** defines the underlying enumeration type and **T2** defines the set of **Cards** that are defined based on that enumeration type. For example, if **T** is **Color** or **Fate**, then **T2** is `Card<Color>` or `Card<Fate>`.
- The lines from **GreenDeck**, **RedDeck**, and **FateDeck** to **Deck** indicate that **GreenDeck**, **RedDeck**, and **FateDeck** are derived from **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**, **Color**, **RedDeck**, **RedDeckTest**, **Fate** and the **Driver** have been fully implemented and **should not be modified**.
2. The class **Deck** is a generic and abstract. 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 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*.

4. Create and implement the class **GreenDeck**. The name of this deck should be stored as "Green 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 *BLACK* and one of type *GREEN*.

5. 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 a GreenDeck, a RedDeck, and a FateDeck and reports the number of points the player received. *These point values must utilize the constants created in the Game class, as well as the PLAYER_LOST constant modifier*:

    - If the two cards played are a Red/Green combination and the player won, return RED_GREEN. If the player lost for this combination of cards, return RED_GREEN $\times$ PLAYER_LOST.
    - If the two cards played are a Black/NonBlack combination and the player won, return BLACK_NONBLACK. If the player lost with this combination of cards, then return BLACK_NONBLACK $\times$ PLAYER_LOST.

6. Implement JUnit tests to thoroughly test all classes and methods you created/implemented.

    - We have given a **RedDeckTest.java** for reference, which you can make use of for creating other tests.
    - 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.

# Hints

- See the method *java.lang.Enum.valueOf()*

# Example Output

Below is an example output of the full program, playing a total of 5 games and then reporting the total score for the player at the end. The details of your games will vary (we do not test the output of Driver).

```
********** GAME START **********
Player's Current Total Score: 0
***** You have Green *****
You: Black
Opp: Red
Player's Current Total Score: 1
***** You have Green *****
You: Black
Opp: Black
You: Black
Opp: Red
Player's Current Total Score: 2
***** You have Red *****
You: Black
Opp: Black
You: Red
Opp: Black
Player's Current Total Score: 1
***** You have Red *****
You: Black
Opp: Black
You: Black
Opp: Black
You: Red
Opp: Black
Player's Current Total Score: 0
***** You have Red *****
You: Black
Opp: Black
You: Red
Opp: Green
********** GAME END **********
Player's Final Score: 4
```

# 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

Before submission, finish testing your program by executing your unit tests. If your program passes all tests and your classes are covered completely by your test classes, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due at 7pm on Saturday, October 7. **Submission must be done through the Web-Cat server.**

- Use the same submission process as you used in lab 1. You must submit your implementation to the *Lab 7: Generics, Lists and Stacks* area on the Web-Cat server.

# Rubric

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

**Correctness/Testing: 40 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 a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

**Style/Coding: 25 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)

If you do not submit compiled Javadoc for your project, 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. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late).

After 15 submissions to Web-Cat, you will be penalized one point for every additional submission.

For labs, the server will continue to accept submissions for three days after the deadline. In these cases, you will still have the benefit of the automatic feedback. However, beyond ten minutes late, you will receive a score of zero.

The grader will make their best effort to select the submission that yields the highest score.