CS 2334: Lab 7

Generics, Lists and Queues

Generics

We know that we can assign an object of one class to an object of another class provided that they are compatible. For example:

```
Public void sampleMethod(Number n){…}

sampleMethod(new Integer(2));
sampleMethod(new Double(2.1));
```

- These are okay because Integer and Double are subtypes of Number

Generics

The same is true with generics. We can perform a generic type invocation with Number as its argument, and it will be compatible with objects of type Number. For example:

```
ArrayList<Number> list1 =
          new ArrayList<Number>();
ArrayList.add(new Integer(2));
ArrayList.add(new Double(2.1));
```

Implementing Generics

- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces or methods.
- This makes it possible to re-use the same code with different types.

Implementing Generics

Example:

```
class Person<E>{
   private E id;

   public Person(E id){
        this.id = id;
   }
   public E getId(){
        return id;
   }
}
Person<Integer> p1 = new Person<Integer>(22);
Person<String> p2 = new Person<String>("22");
```

Multiple Type Parameters

A generic class can have multiple type parameters. For example:

```
    class Instructor<U, V>{
        private U courseNum;
        private V name;

        public Instructor (U courseNum, V name){
            courseNum= courseNum;
            this.name = name;
        }
    }
Person<Integer, String> p1 =
        new Person<Integer, String>(01,”Joe”);
```

Bounded Type Parameters

- Bounded type parameters allow us to restrict the types that can be used as type arguments in a parameterized type.
- They also allow us to invoke methods from the types defined in the bounds.
- To declare a bounded parameter, list the type parameter's name, followed by the ***extends*** keyword (***implements*** for interfaces), then its upper bound.

Implementing Bounded Type Parameters

Example:

```
class NaturalNum<E extends Integer>{
    private E n;

    public NaturalNum(E n){
          this.n = n;
    }
    public E isEven(){
          return n.intValue() % 2 == 0;
    }
}
```

isEven() invokes intValue(), a method defined in the Integer class

Implementing Bounded Type Parameters

Example: Using a pre-defined class as the upper bound

```
class Student<E extends Person<E2>, E2> {
   public E2 StudentId(E gen){
        return gen.getId();
   }
}
```

- First parameter: E
- Second parameter: E2

But, they have a specific relationship: E is-a Person<E2>

Stacks and Queues

- Stacks and Queues are defined by two basic operations: inserting a new item, and removing an item.
- The rules differ when we add and remove items for each container

Queues

- A queue removes an object according to the first-in-first-out (FIFO) principle.
- An object may be inserted at any time, but only the object that has been in the queue the longest is removed.
- Objects are inserted at the rear and removed from the front.

Queues

- The queue supports two main methods:
  - add(Object o): inserts Object o at the rear of the queue
  - remove(): removes the object from the front of the queue
- Other methods supported by the queue data type can be found in the Java API:
  - https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

Stacks

- A stack removes an object according to the last-in-first-out (LIFO) principle, and adds an object to the top of the list.
- Only the last (or most recently) added object can be removed.

Stacks

- The stack data type supports two main methods:
  - push(o): adds Object o to the top of the stack
  - pop(): Removes the top object and returns it.
- Other methods supported by the stack data type can be found in the Java API:
  - https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

Enumerated Data Types

- An enumerated type is a special data type that allows a variable to be one of a set of predefined constants.
- Example: Types of Cars

```
public enum Car{
    FORD, TOYOTA, HONDA;
}
```

Note: the names of an enum type's fields are in uppercase letters because they are constants (this is the convention)

Enumerated Data Types

```
public enum Car{
    FORD, TOYOTA, HONDA;
}
```

Can the use our enum as a variable type:

```
Car c = FORD;
if(c == TOYOTA) {
    …
}
```

Enumerated Data Types

The variables of an enumerated type can also be defined with a value. Example:

```
public enum Car{
  //these are calls to the constructor
  FORD("Truck"), TOYOTA("SUV"), HONDA("Van");

  private Car(String carType){
       this.carType= carType;
    }
  }
```

Note: the constructor for an enum must be private (only the class creates instances)

Lab 7: Card Game

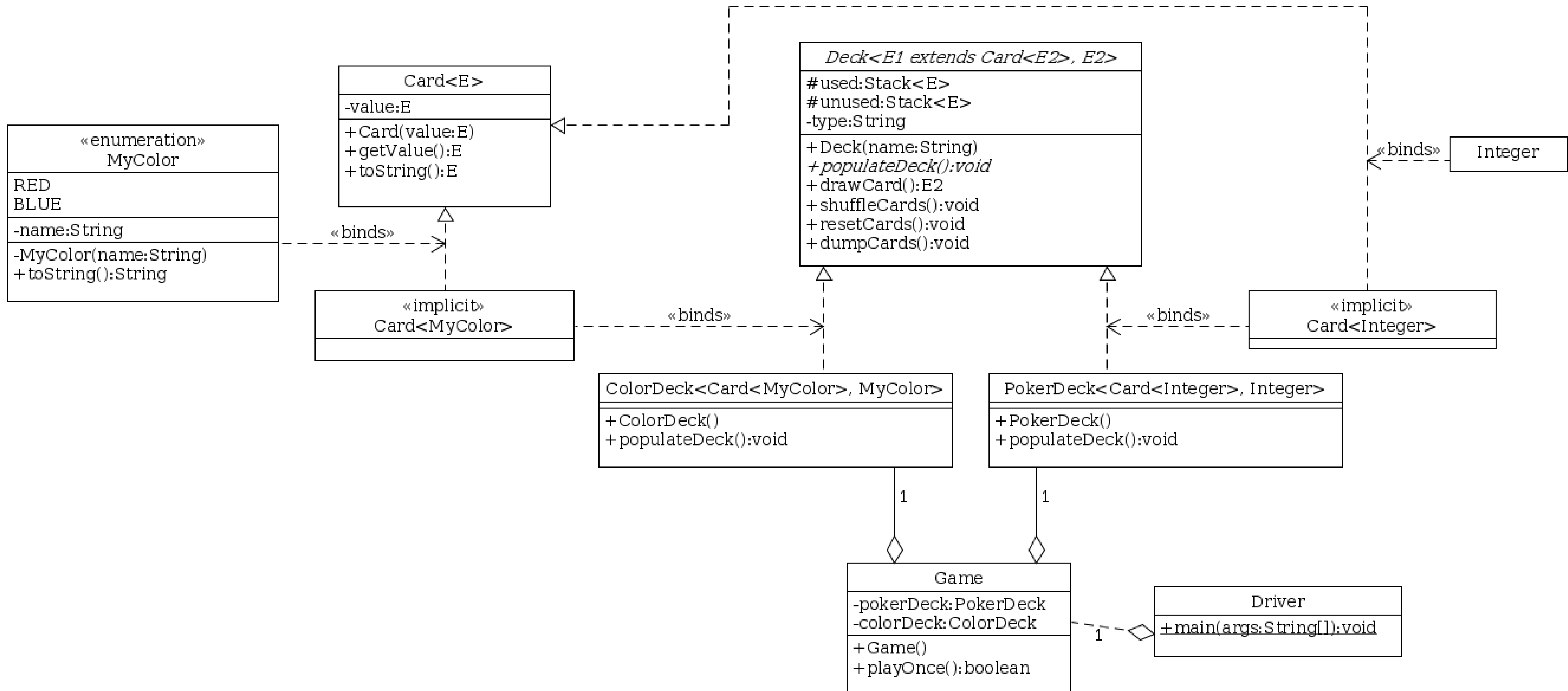
- We will create a card game. The game has two decks of cards: a poker deck and a color deck.
- To play the game:
  - The player draws one card from each deck
  - If the color card is RED, the player wins if the poker card has an even value
  - If the color card is BLUE, the player wins if the poker card is divisible by three

Demonstration

Implementation

We need several pieces:

- An enum for representing colors (RED/BLUE) – we provide this
- A general notion of a Card.
  - Cards have a generic type associated with them
- A general notion of a Deck
  - Stacks of used and unused cards
  - Shuffling & drawing operations
  - A generic Deck is made up of a specific type of Card


Deck<E1 extends Card<E2>, E2>
#used:Stack<E>
#unused:Stack<E>
-type:String
+Deck(name:String)
+populateDeck():void
+drawCard():E2
+shuffleCards():void
+resetCards():void
+dumpCards():void
Card<E>
-value:E
+Card(value:E)
+getValue():E
+toString():E
«enumeration»
MyColor
RED
BLUE
-name:String
-MyColor(name:String)
+toString():String
«binds»
«implicit»
Card<MyColor>
«binds»
«binds»
Integer
«binds»
«implicit»
Card<Integer>
ColorDeck<Card<MyColor>, MyColor>
+ColorDeck()
+populateDeck():void
PokerDeck<Card<Integer>, Integer>
+PokerDeck()
+populateDeck():void
1
1
Game
-pokerDeck:PokerDeck
-colorDeck:ColorDeck
+Game()
+playOnce():boolean
1
Driver
+main(args:String[]):void

Lab 7 Preparation

- Download lab7-initial.zip
- Import into your Eclipse project

(details of how to do this are in the lab specification)

Lab 7

- We've provided three fully implemented classes
  - Card
  - MyColor
  - PokerDeck

  (**Do not modify these classes**)
- Implement the other classes represented in the UML
  - Watch spelling and casing

* *During the lab: stop part way into their work to discuss one or two methods in Deck()*

Submission

- Submit only one file: lab7.zip (casing matters)
- Due date: Sunday, October 11th @11:59pm
- Submit to lab7 dropbox on D2L