M6 (a) - Inheritance

Jin L.C. Guo

Image source https://cdn.pixabay.com/photo/2015/01/11/21/30/cats-596782_1280.jpg

Midterm Related

- Check your Conflict before Wednesday.
- Allocate rooms within this week.
- Cover Module 1-5.
- Review important content next Tuesday.
- No assignment this week.

Recap

- Composition
  - Polymorphic Object Cloning
  - Prototype Pattern
  - Command Pattern
  - Principle: Law of Demeter

Objective

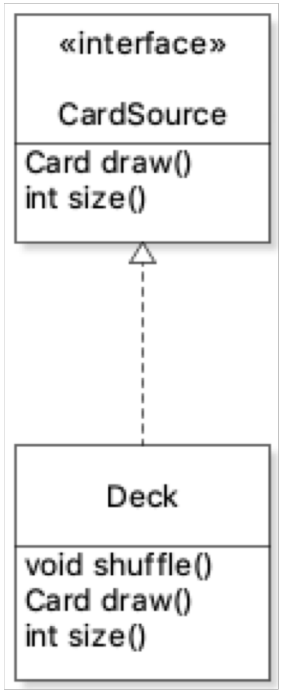
- Conceptual foundations of inheritance
- Inheritance in Java
- Abstract Class
- Template Method Pattern

Run-time vs Compile-time Type

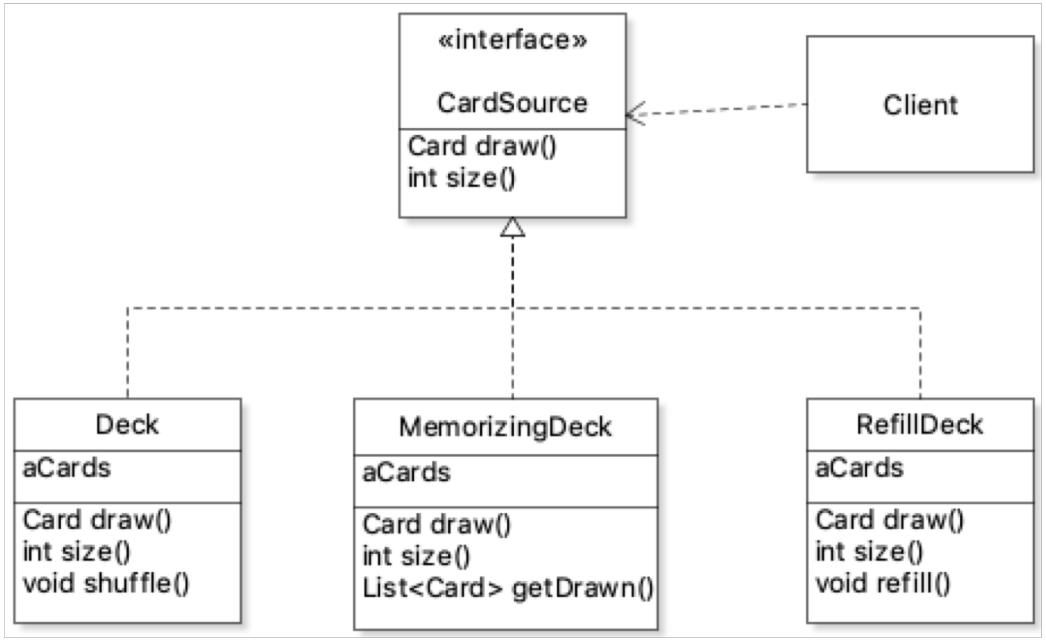
```
Deck deck1 = new Deck();
CardSource deck2 = deck1;
```

compares an object to a specified type.

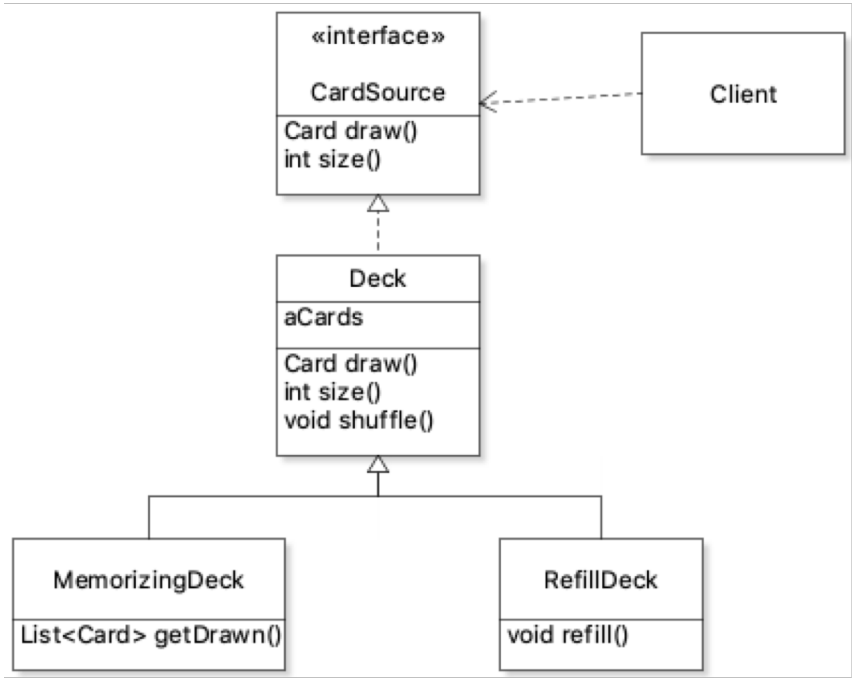
```
System.out.println(deck1 instanceof Deck);
System.out.println(deck2 instanceof Deck);

deck1.shuffle();
((Deck)deck2).shuffle();//not allowed
```



Design of CardSource



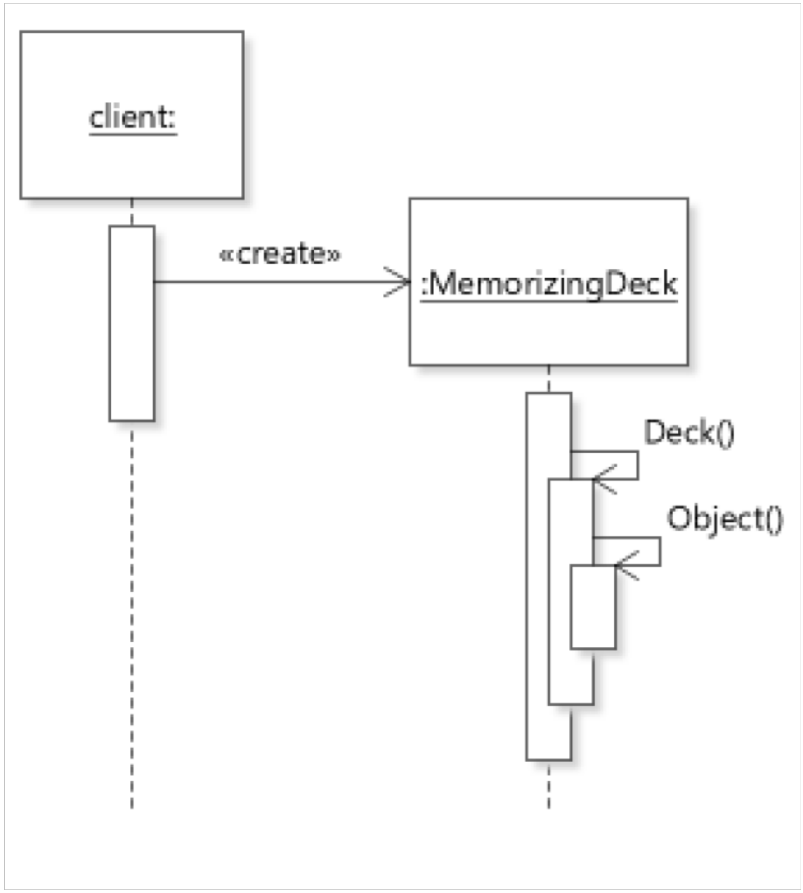
Remove Redundancy by Inheritance



```
public class Deck implements CardSource
{
    private final List<Card> aCards;


    public Deck()
    {
        aCards = new ArrayList<>();
        shuffle();
    }

}
```

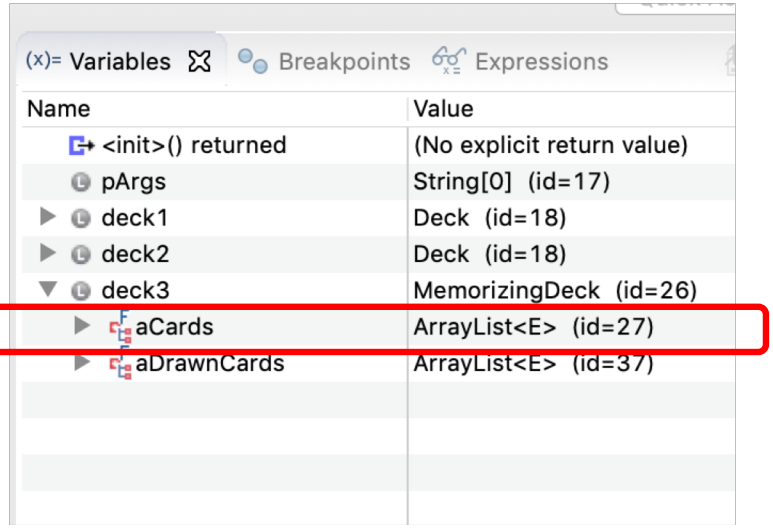
```
public class MemorizingDeck extends Deck
{
    private final List<Card> aDrawnCards = new ArrayList<>();
}
```

```
Deck deck3 = new MemorizingDeck();
```



Inheriting Fields

```
Deck deck3 = new MemorizingDeck();
```




```
public class Deck implements CardSource
{
    private final List<Card> aCards;
```

```java
public class Deck implements CardSource
{
    protected List<Card> aCards;


    /**
    * Initialization is done in the constructor to reduce unnecessary
    object states.
    */
    public Deck()
    {
        aCards = new ArrayList<>();
        shuffle();
    }

    public Deck(List<Card> pCards) {
        aCards = new ArrayList<>(pCards);
    }
```

```
public class MemorizingDeck extends Deck
{
    private final List<Card> aDrawnCards = new ArrayList<>();

    public MemorizingDeck(List<Card> pCards)
    {
        new Deck(pCards);
    }
}
```

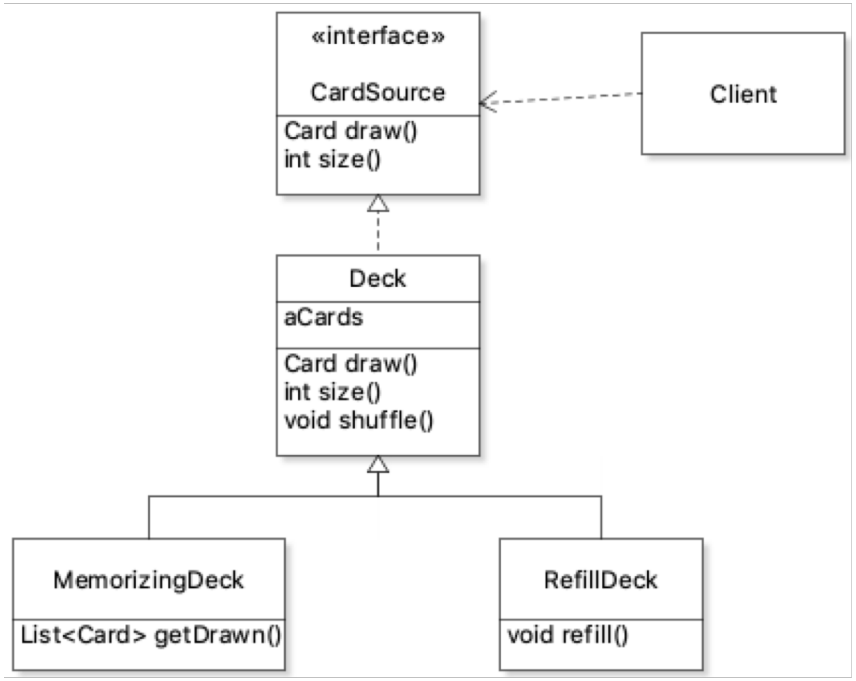
Can I do this?

```
public class MemorizingDeck extends Deck
{
    private final List<Card> aDrawnCards = new ArrayList<>();

    public MemorizingDeck(List<Card> pCards)
    {
        super(pCards);
    }
}
```

Proper way to call the constructor of super class

Inheriting Methods

```
Deck deck4 = new MemorizingDeck();
deck4.draw();
deck4.shuffle();
```



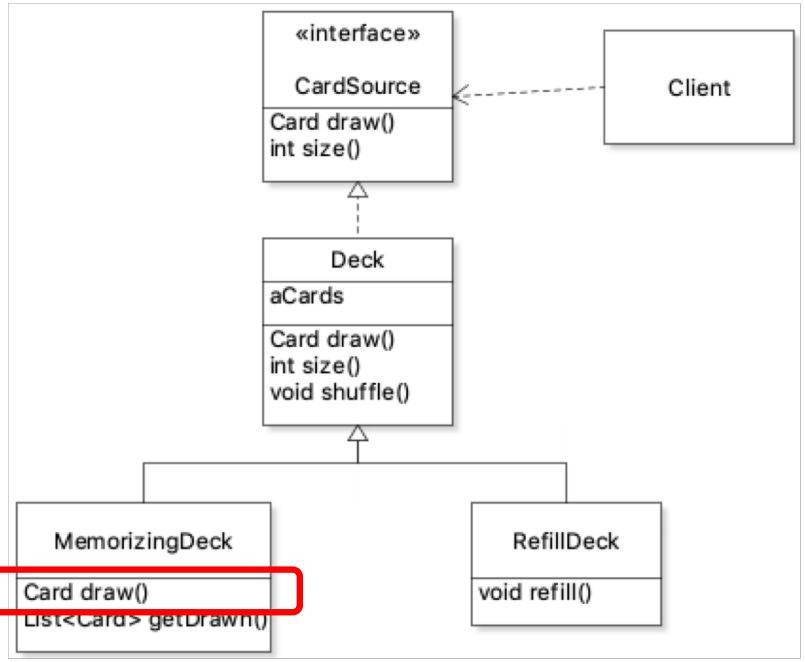
Override Methods

```
public class MemorizingDeck extends Deck
{
    private final List<Card> aDrawnCards = new ArrayList<>();

    @Override
    public Card draw()
    {
        assert size()>0;
        Card cardDrawn = aCards.get(aCards.size()-1);
        aCards.remove(aCards.size()-1);
        aDrawnCards.add(cardDrawn);
        return cardDrawn;
    }
}
```

Inheriting Methods

```
Deck deck4 = new MemorizingDeck();
deck4.draw();
deck4.shuffle();
```

Dynamic Binding

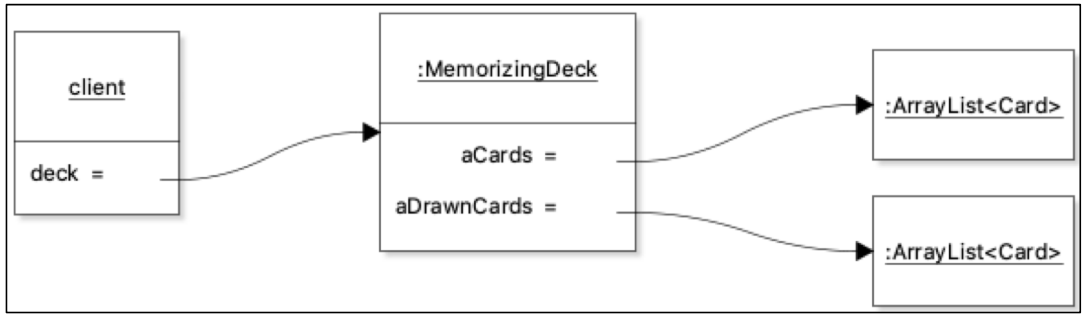


Keep field in super class private?

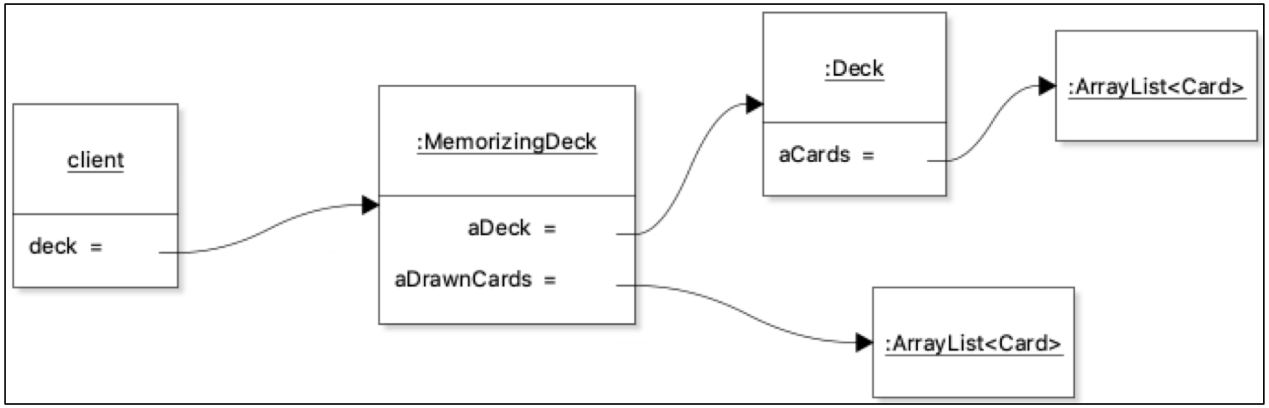
```
public class MemorizingDeck extends Deck
{
    private final List<Card> aDrawnCards = new ArrayList<>();

    @Override
    public Card draw()
    {
        assert size()>0;
        Card cardDrawn = aCards.get(aCards.size()-1);
        aCards.remove(aCards.size()-1);
        aDrawnCards.add(cardDrawn);
        return cardDrawn;
    }
}
```

```
Card cardDrawn = super.draw();
```

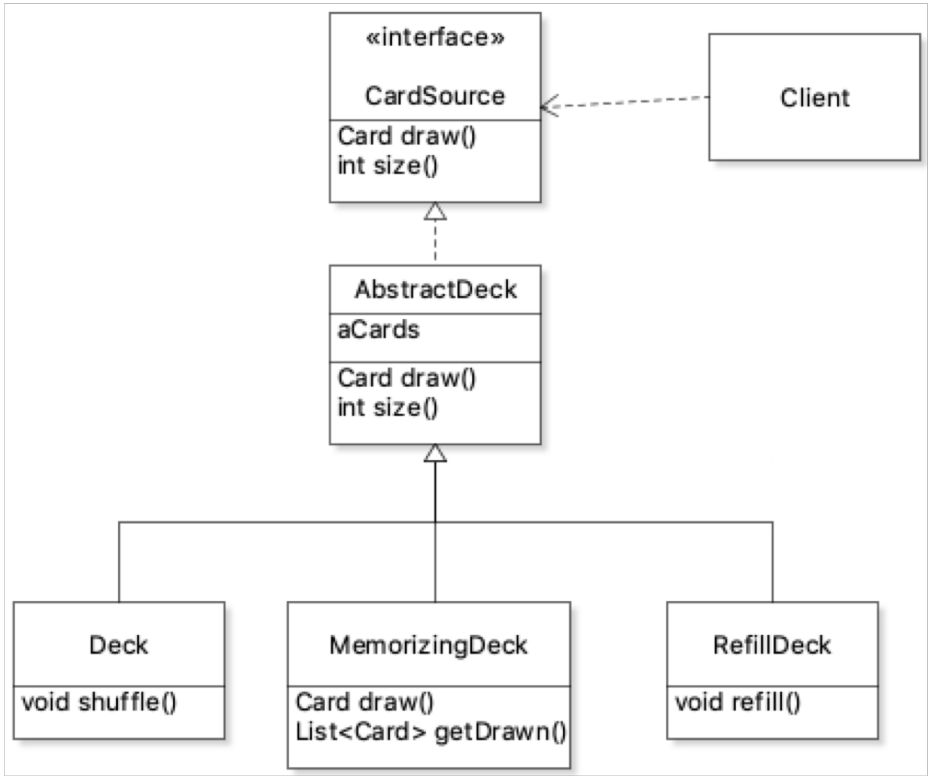
Comparison of Inheritance and Composition



VS



Abstract Class



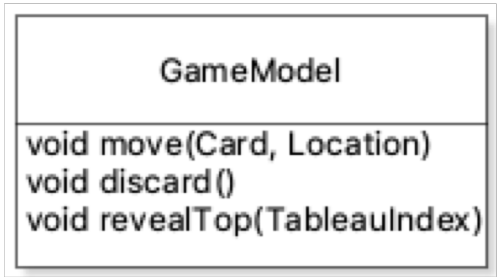
Abstract Class

- The class cannot be instantiated
- No longer needs to supply implementations to all methods in the interface it declares to implement.
  - Subclass needs to implement
- Can declare new abstract methods
  - Subclass needs to implement

```java
public abstract class AbstractDeck implements CardSource
{
    protected final List<Card> aCards;

    protected AbstractDeck(List<Card> pCards)
    {
        aCards = new ArrayList<>(pCards);
    }

    @Override
    public Card draw()
    {
        assert aCards.size()>0;
        return aCards.remove(aCards.size()-1);
    }
    @Override
    public int size()
    {
        return aCards.size();
    }
    public abstract void log();
}
```

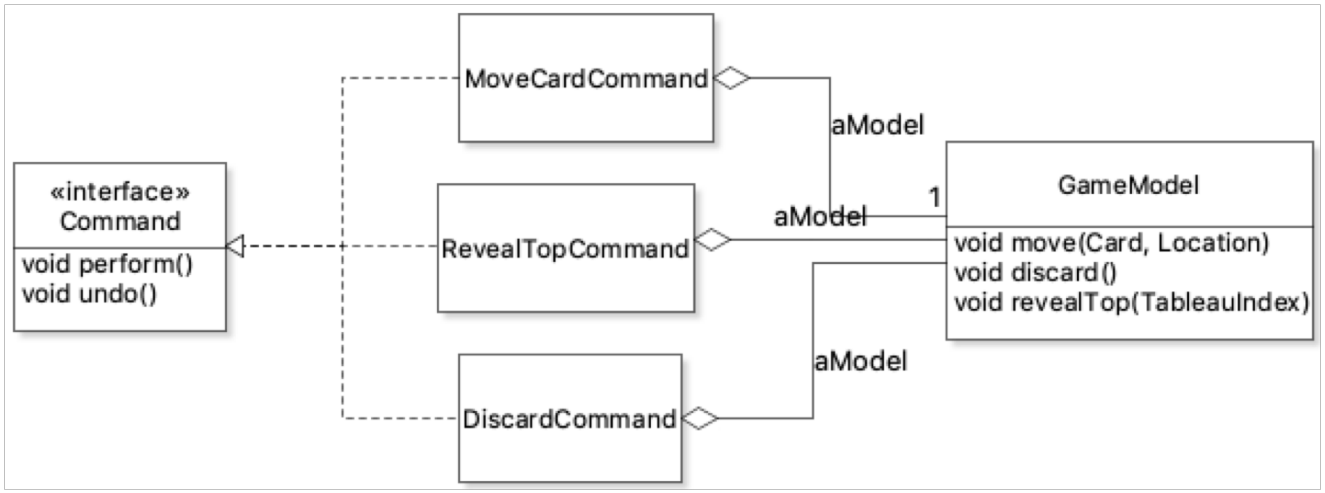
Activity1:

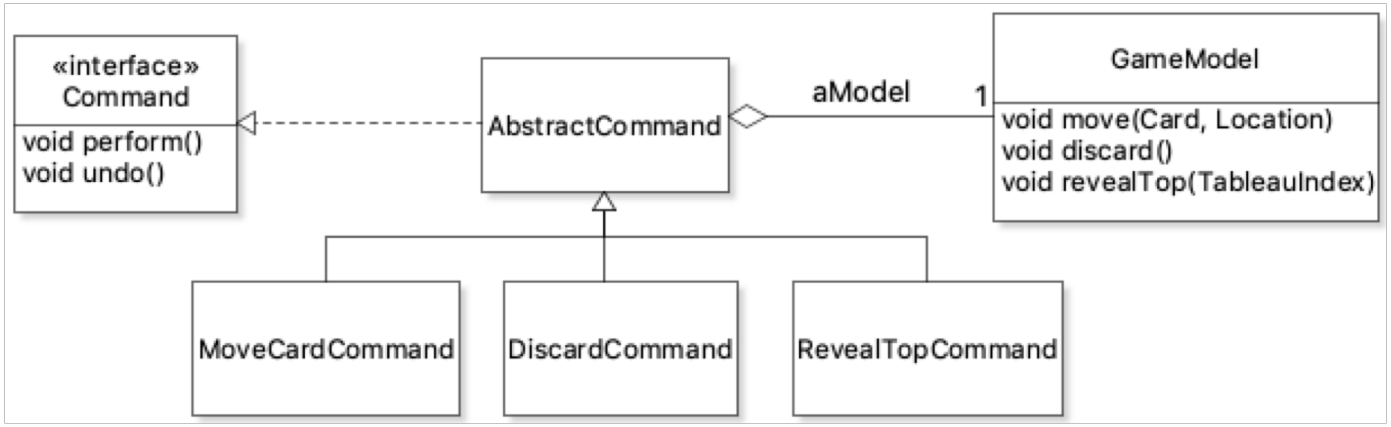
- Consider the following methods in GameModel. Use class diagram to illustrate how to use command pattern and inheritance to support “undo” those opeartions.



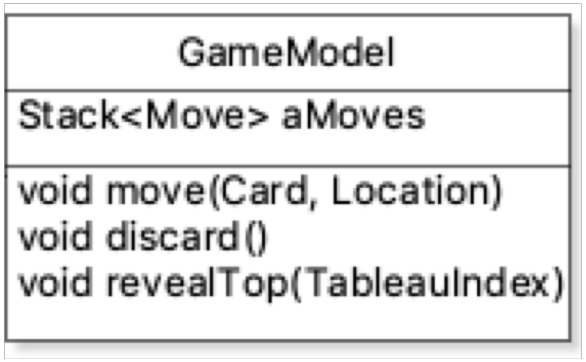
Activity1:

- Use inheritance to remove redundancy in the following design of applying command pattern to game model operations.




«interface»
Command
void perform()
void undo()
AbstractCommand
aModel
1
GameModel
void move(Card, Location)
void discard()
void revealTop(TableauIndex)
MoveCardCommand
DiscardCommand
RevealTopCommand

Multiple steps in Perform




```
public void perform()
{
    aModel.pushMove(this);

    /*  Actual command operation */

    log();
}
```

Step1

Step2

Step3

```java
public abstract class AbstractCommand implements Command
{
    protected final GameModel aModel;
    protected AbstractCommand (GameModel pModel)
    {
        aModel = pModel;
    }

    @Override
    public void perform()
    {
        aModel.pushCommand(this);
        /*  Actual command operation */
        log();
    }
```

```
public abstract class AbstractCommand implements Command
{
    protected final GameModel aModel;
    protected AbstractCommand (GameModel pModel)
    {
        aModel = pModel;
    }

    @Override
    public void perform()
    {
        aModel.pushCommand(this);
        executeCommand();
        log();
    }

    protected abstract void executeCommand();
```

final

Subclass cannot override this method

```java
public class RevealTopCommand extends AbstractCommand
{
    TableauIndex aIndex;


    protected RevealTopCommand(GameModel pModel, TableauIndex pIndex)
    {
        super(pModel);
        aIndex = pIndex;
    }

    @Override
    protected void executeCommand()
    {
        aModel.revealTop(aIndex);
    }


}
```

Template Method Pattern

- Intent:
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Participants:
  - AbstractClass

    implements a template method defining the skeleton of an algorithm

    defines abstract operations that concrete subclasses define to implement steps of an algorithm.
  - ConcreteClass

    implements the operations to carry out subclass-specific steps of the algorithm.

Template Method Pattern

```
@Override
public final void perform()
{
    aModel.pushCommand(this);
    executeCommand();
    log();
}
```

Abstract step method **Protected** or **Public**

Avoid same names for template and abstract methods

Not necessarily abstract (define default behavior)

Examples of Abstract classes and Template Method Pattern in Java

- java.util.AbstractList
- java.util.AbstractSet
- java.util.AbstractMap
- java.io.InputStream
- java.io.OutputStream
- java.io.Reader
- java.io.Writer

… …

java.util.AbstractList

- Implemented Interfaces:

Iterable<E>, Collection<E>, List<E>

- Direct Subclasses:

AbstractSequentialList, ArrayList, Vector

java.util.AbstractList

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the get(int) and size() methods.

To implement an modifiable list, the programmer must additionally override the set(int, E) method (which otherwise throws an UnsupportedOperationException).

… …

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E>
{
     … …
     abstract public E get(int index);

     public E next() {
          checkForComodification();
          try {
              int i = cursor;
              E next = get(i);
              lastRet = i;
              cursor = i + 1;
              return next;
          } catch (IndexOutOfBoundsException e) {
              checkForComodification();
              throw new NoSuchElementException();
          }
     }
     … …
 }
```

Activity 2:

- What methods do you need to override for implementing an unmodifiable list of Card that is constructed through a card array.
- How to override them?

```java
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    CardList(Card[] pCards)
    {
        assert pCards!= null;
        aCards = pCards;
    }
    @Override
    public Card get(int index)
    {
        assert index>=0 && index<size();
        return aCards[index];
    }
    @Override
    public int size()
    {
        return aCards.length;
    }
}
```

```java
public static void main(String[] pArgs)
{
    Card[] cards = new Card[2];
    cards[0] = new Card(Rank.ACE, Suit.CLUBS);
    cards[1] = new Card(Rank.FIVE, Suit.DIAMONDS);
    CardList cardList = new CardList(cards);

    System.out.println(cardList.contains(cards[1]));

    for (Iterator<Card> iter=cardList.iterator();
        iter.hasNext(); )
    {
        Card element = iter.next();
        System.out.println(element);
    }
}
```