

# M1 (a) – Encapsulation

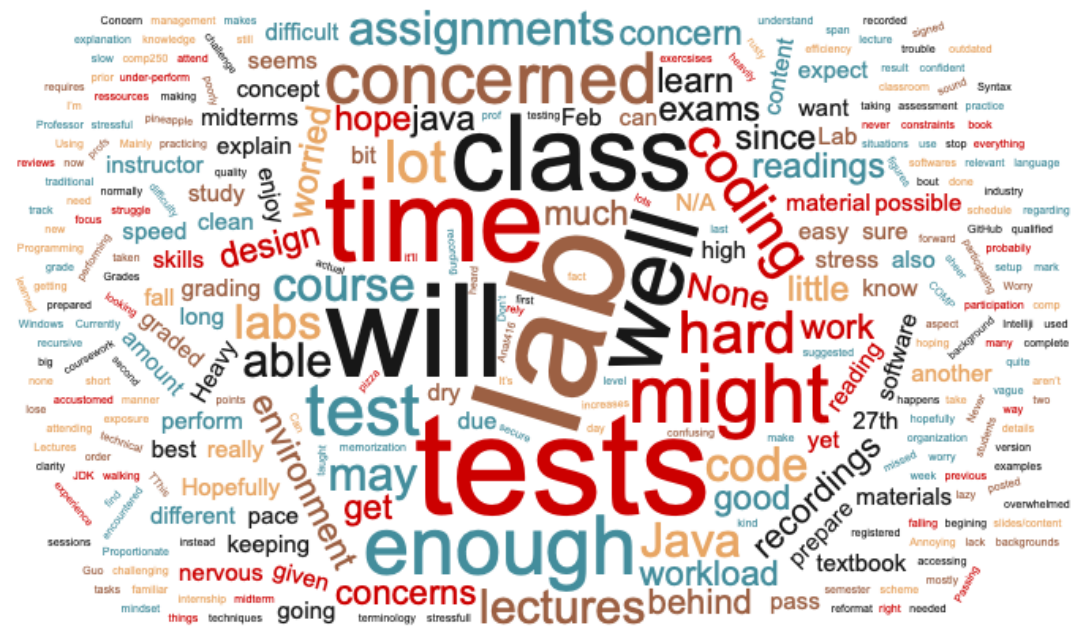
---

Jin L.C. Guo

# What do you want to learn most?



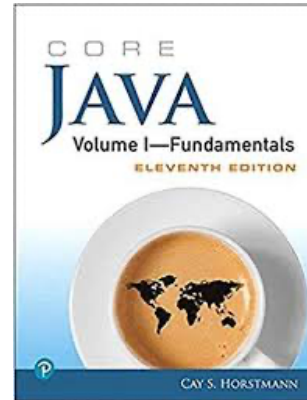
# What are your concerns?



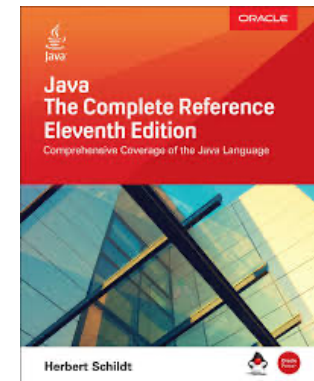
# Additional references for Java

- <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html>

- Core Java Volume I—Fundamentals, Eleventh Edition



- Java: The Complete Reference, Eleventh Edition



# Lab Test

- Four sessions
- Location: Trottier 3120
- Max 10 people for each time slot

# Lab Test Plan

Week	Monday Class	Friday Class	Lab Test
1	Introduction	Encapsulation	
2	Encapsulation	Types and Polymorphism	
3	Types and Polymorphism	Types and Polymorphism	
4	Object State	Object State	Session 1
5	Defensive Design	Defensive Design	Session 1
6	Unit Testing	Unit Testing	Session 2
7	Composition	Composition	Session 2
8	Review	Midterm (Feb 28th)	
9	Study Break	Study Break	
10	Composition	Inheritance	
11	Inheritance	Inversion of Control	Session 3
12	Inversion of Control	Inversion of Control	Session 3
13	Refactoring	Design for Usability	Session 4
14	Software Engineering Ethics	Wrap Up	Session 4

## Available Slots:

Slot	Date of the Week	Time
1	Monday	10am - 11am
2	Monday	11am - 12 am
3	Tuesday	10am – 11am
4	Tuesday	11am - 12 am
5	Tuesday	12pm – 1pm
6	Tuesday	1pm – 2pm
7	Wednesday	10:30am-11:30am
8	Wednesday	10:30am-11:30am
9	Wednesday	11:30am-12:30pm
10	Wednesday	11:30am-12:30pm
11	Thursday	1pm - 2pm
12	Thursday	2pm - 3pm
13	Friday	8am – 9am
14	Friday	9am - 10am

# Sign up for lab test

- [https://docs.google.com/spreadsheets/d/1dvCugSYc8ivpxAiW5GoW0hCQNfIS8NQWIB\\_C8SYFWEw/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1dvCugSYc8ivpxAiW5GoW0hCQNfIS8NQWIB_C8SYFWEw/edit?usp=sharing)
- Email to Fuyuan Lyu (fuyuan.lyu@mail.mcgill.ca) including
  - Your McGill ID
  - Date, Time of the slot.

**Email Title: [COMP303]Lab Test SignUp**

\*\*\*Otherwise email will not be processed properly\*\*\*



# Lab test

Lab test sample:

- <https://github.com/prmr/COMP303/blob/2019F/Assessment.md>

Lab test grading scheme:

- 4 lab tests -> Using 3 top grades


# Other Questions from you

- Online forum
  - Contributing by: posting, responding, editing, commenting, etc.
- The Goto place when having problems. But check existing posts before posting new ones.

# Recap of last class

- The focus and definition of Software Design
- Role of Design in Software Engineering Process
- How to Store and Share Design Knowledge
- Objective of COMP 303

# Objectives of this class

- Programming mechanisms:
  - Scope and Visibility
- Concepts and Principles:
  - Information Hiding, Encapsulation, Escaping Reference, Immutability
- Design Techniques:
  - Object Diagrams
- Patterns and Antipatterns:
  - Primitive Obsession 

# Programming Mechanism Review

- Classes and Interfaces

# Access Modifiers for class members

- private: accessible from top-level class where it is declared
- package-private (default): from any class in the package
- protected: from subclass and any class in the package
- public: anywhere



# Programming Mechanism Review

- Java static type system

Interfaces/Annotations

Classes/Enums

Arrays

Primitives

*byte, short, int, long, float, double, boolean, char*



Reference types

# Java Memory Organization

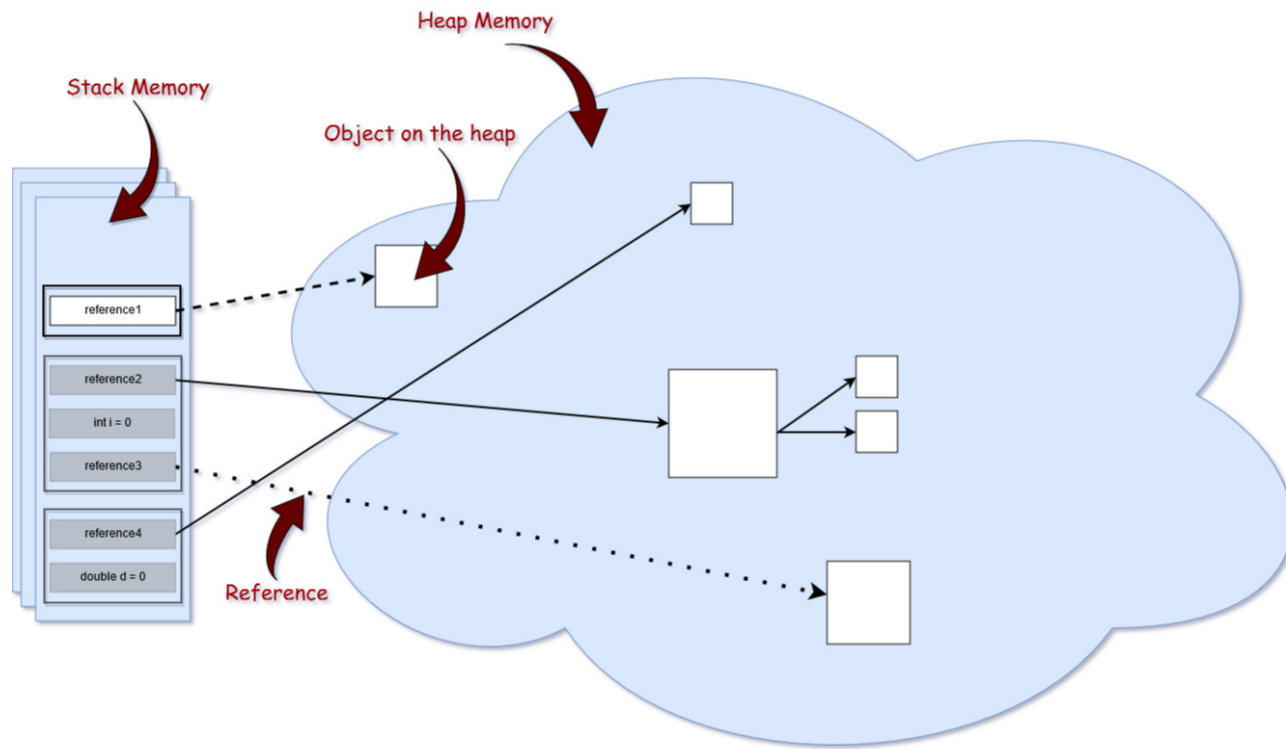


Image Source: <https://dzone.com/articles/java-memory-management>



## Very first task (Activity 1)

- Design the representation of a deck of playing cards.



Code under design

Client code

# Options: using primitive data types

## Use integer

- Clubs 0-12
- Hearts 13-25
- Spades 26-38
- Diamonds 39-51

```
int card = 13; // The Ace of Hearts
int suit = card / 13; // 1 = Hearts
int rank = card % 13; // 0 = Ace
```

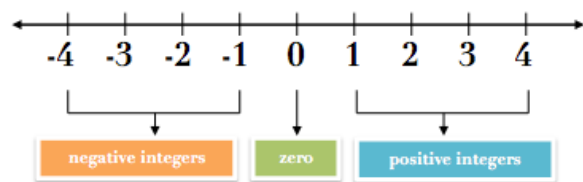
## Options: using primitive data types

- Use pair of values [int, int]
  - Rank 0-12
  - suit 0-4

```
int[] card = {1,0}; // The Ace of Hearts  
int suit = card[0];  
int rank = card[1];
```

**Problems?**

**Representation**



**Domain Concept**



## Activity 2

- Representing phone number with string?
- Note: the String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. [[Java Primitive Data Types](#)]

# Anti-pattern

- Primitive Obsession

- **Symptoms**

- Use of primitives for “simple” tasks (such as currency, ranges, special strings for phone numbers, etc.)

# Anti-pattern

- Primitive Obsession

- **Symptoms**

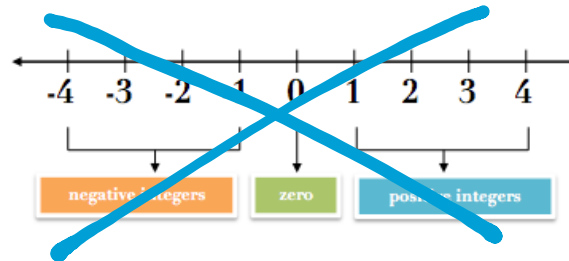
- Use of primitives for “simple” tasks (such as currency, ranges, special strings for phone numbers, etc.)

- **Treatment**

- Replace Primitive with Object (if you are doing things other than simple printing)



**Representation  
Implementation**



**Representation  
Implementation**

**loosely coupled**

# Define our own Card type

```
public class Card  
{  
    ...  
}
```



# Characterizing the Card

int constant? string constant?

- Suit
  - Clubs, Hearts, Spades, Diamonds
- Rank
  - Ace, Two, ..., Jack, Queen, King



# Characterizing the Card

- Suit
  - Clubs, Hearts, Spades, Diamonds


```
public enum Suit
{ CLUBS, DIAMONDS, SPADES, HEARTS
}
```

- Rank
  - Ace, Two, ..., Jack, Queen, King

```
public enum Rank
{ ACE, TWO, THREE, FOUR, FIVE, SIX,
  SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING;
}
```



# Java Enum Type

- For predefined constants
  - Compile-time type safety
- 
- Suit* can only be one of *CLUBS*, *DIAMONDS*, *SPADES*, *HEARTS*
- Add methods and other fields
  - Instance-controlled -- classes that export one instance for each enumeration constant via a public static final field

## Back to our Card Class

```
public class Card
{
    public Rank aRank;
    public Suit aSuit;
}
```

```
card.aRank = null;
System.out.println(card.aRank.toString());

java.lang.NullPointerException
```

# Better Encapsulated Card Class

```
public class Card
{
    private Rank aRank;
    private Suit aSuit;

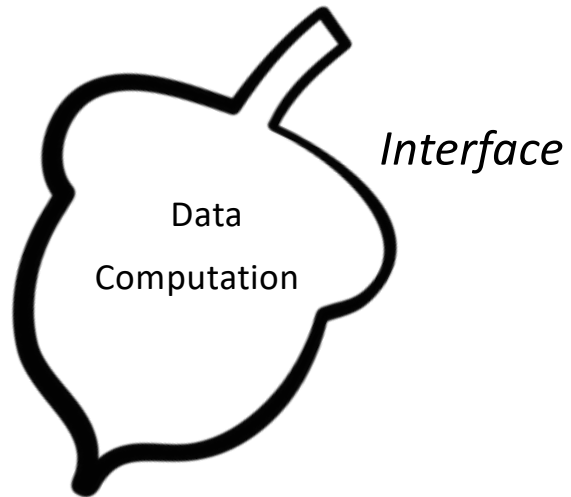
    public Card(Rank pRank, Suit pSuit)
    {
        aRank = pRank;
        aSuit = pSuit;
    }

    public Rank getRank()
    {
        return aRank;
    }
    public void setRank(Rank pRank)
    {
        aRank = pRank;
    }

    .....
}
```



# Encapsulation



Goal: to minimize the contact points

# Representation of Deck?

```
List<Card> Deck = new ArrayList<>();
```

# Representation of Deck?

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public void addCard(Card pCard)
    {
        aCards.add(pCard);
    }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

# Information Hiding

- [\*On the criteria to be used in decomposing systems into modules\*](#)

David Parnas - Communications of the ACM, 1972 - dl.acm.org

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.

# Information Hiding

- A principle to divide any piece of equipment, software or hardware, into modules of functionality.
- Modularization can improve the flexibility and comprehensibility of a system while allowing the shortening of its development time.

# Information Leaking:

- Escaping References: Why this is bad?

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public void addCard(Card pCard)
    {
        aCards.add(pCard);
    }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

# Escaping References

- It should NOT be possible to change the state of an object without going through its own methods.

- Red flag:

Storing an external reference internally!

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public void addCard(Card pCard)
    {
        aCards.add(pCard);
    }
}
```



# Escaping References

- It should NOT be possible to change the state of an object without going through its own methods.

- Red flag:

Returning a reference to an internal object!

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public List<Card> getCards()
    {
        return aCards;
    }
}
```

# Escaping References

- It should NOT be possible to change the state of an object without going through its own methods.

- Red flag:

Leaking references through Shared structures!

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public void collect(List<Card> pAllCard)
    {
        pAllCard.addAll(aCards);
    }
}
```

# Change Card to Immutable

- Immutable: the internal state of the object cannot be changed after initialization.
- How to change the Card Class?

# Change Card to Immutable

```
public class Card
{
    final private Rank aRank;
    private Suit aSuit;

    public Card(Rank pRank, Suit pSuit)
    {
        aRank = pRank;
        aSuit = pSuit;
    }

    public Rank getRank()
    {
        return aRank;
    }
    public void setRank(Rank pRank)
    {
        aRank = pRank;
    }
    .....
}
```

# Change Card to Immutable

```
public class Card
{
    final private Rank aRank;
    final private Suit aSuit;

    public Card(Rank pRank, Suit pSuit)
    {
        aRank = pRank;
        aSuit = pSuit;
    }

    public Rank getRank()
    {
        return aRank;
    }

    .....
}
```

# What about Deck?

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public void addCard(Card pCard)
    {
        aCards.add(pCard);
    }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

# Object Diagram

- Model the structure of the system at a specific time

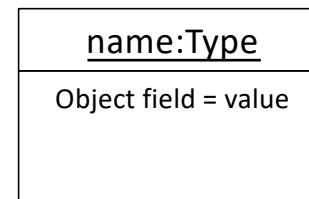
# Object Diagram

- Model the structure of the system at a specific time
- Complete or part of the system



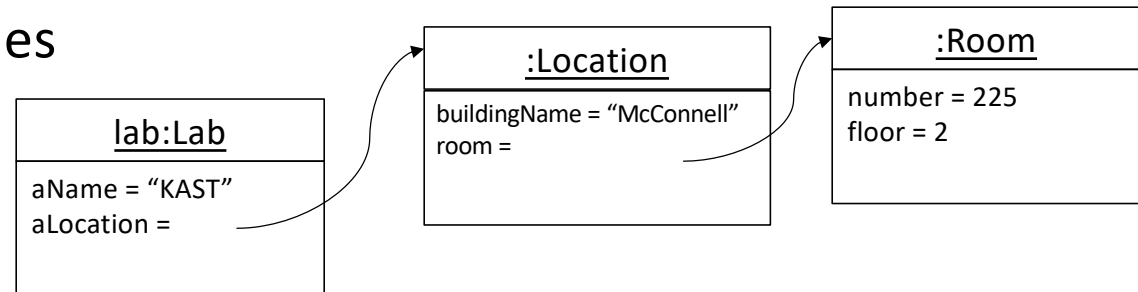
# Object Diagram

- Model the structure of the system at a specific time
- Complete or part of the system
- Include objects and data values



# Object Diagram

- Model the structure of the system at a specific time
- Complete or part of the system
- Include objects and data values



# Object Diagram

- Model the structure of the system at a specific time
- Complete or part of the system
- Include objects and data values
- To discover or explain facts of software design (by capturing object relations)

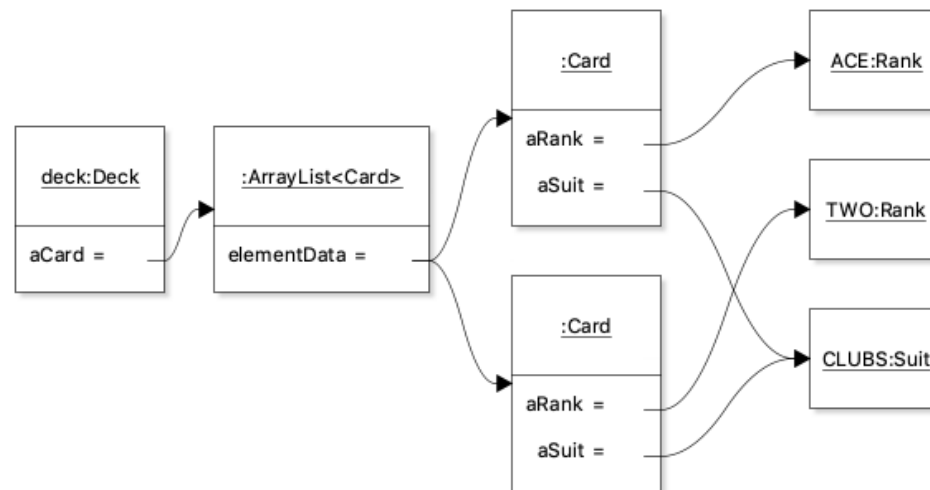
# Activity 3 - Draw Object Diagram

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    public void addCard(Card pCard)
    {
        aCards.add(pCard);
    }
}
```

```
Deck deck = new Deck();
Card card1 = new Card(Rank.ACE, Suit.CLUBS);
Card card2 = new Card(Rank.TWO, Suit.CLUBS);
deck.addCard(card1);
deck.addCard(card2);
```

# Object Diagram - Capturing Object Relations

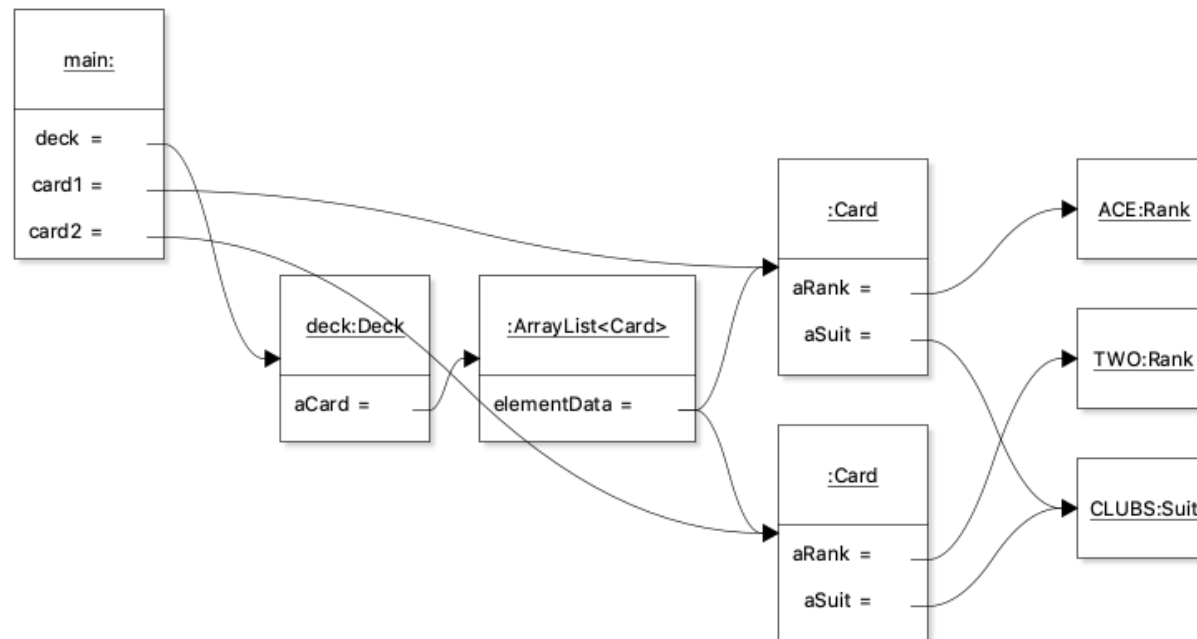


# Java Enum Type

- For predefined constants
- Compile-time type safety  
*Suit* can only be one of *CLUBS*, *DIAMONDS*, *SPADES*, *HEARTS*
- Add methods and other fields
- Instance-controlled -- classes that export one instance for each enumeration constant via a public static final field

# Capturing Object Relations – Object Diagram

method scope



# What about Deck?

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    ... ..
    public int size ()
    {
        return aCards.size();
    }

    public Card getCard(int pIndex)
    {
        return aCards.get(pIndex);
    }
}
```

Add access methods that only return references to immutable objects.



# What about Deck?

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    ... ..

    public List<Card> getCards()
    {
        ArrayList<Card> result = new ArrayList<>();
        for(Card card:aCards)
        {
            result.add(new Card(card.getRank(), card.getSuit()));
        }
        return result;
    }
}
```

Returning a copy

```
public Card(Card pCard){ ... .. }

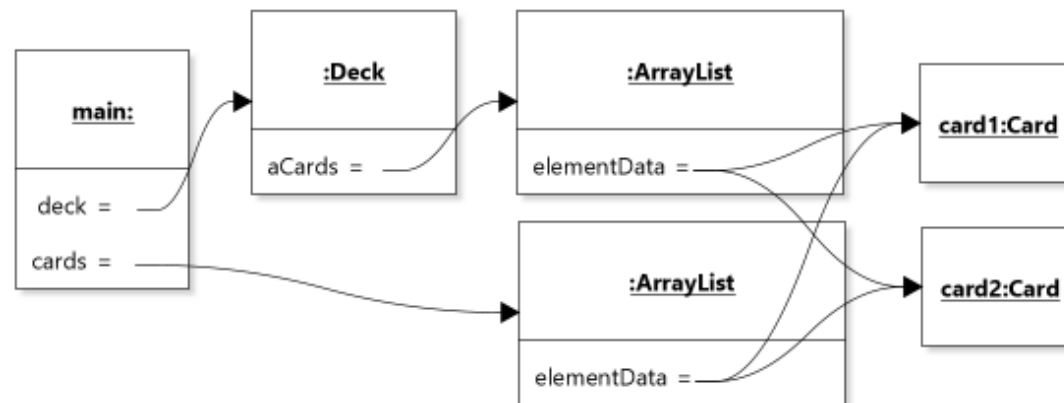
public static copyCard(Card pCard){ ... .. }
```

# What about Deck?

```
public class Deck
{
    private List<Card> aCards = new ArrayList<>();

    ... ..

    public List<Card> getCards()
    {
        return new ArrayList<> (aCards);
    }
}
```



Returning a copy

# Shallow Copy VS Deep Copy

