McGill

COMP 303

Lecture 2

Encapsulation II

Winter 2025

slides by Jonathan Campbell, adapted in part from those of Prof. Jin Guo

Announcements

- Lecture recordings...
- Office hours.
- Midterm date.
- Team forming: deadline Thursday, Jan. 23
  - PDF to be sent out

Recap: terms from last time

- Client code
- Primitive obsession anti-pattern
- Encapsulation
- enum
- Escaping reference

Recap: Primitive Obsession

- **Primitive Obsession** is an **anti-pattern** (a common problem that should be avoided).
- It is the practice of using primitive types (int, String, etc.) to represent domain concepts.
  - Primitive types do not contain any model-specific logic or behaviour.
  - Primitive types lose **type safety** (no compiler errors).

Recap: Encapsulation

- Creating a type for our design abstraction is the first step of **encapsulation**:
  - the idea that data and computation should be bundled together,
  - external code should not need to worry about exactly how the data is represented, nor how the computation is done.
    - we want to hide as much as possible about the internal representation of a class (**information hiding**), e.g., by having all fields be private, and not letting any references escape.

Recap: enum

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

Recap: enum

```
from enum import Enum
class Suit(Enum):
  CLUBS = 1
  SPADES = 2
  DIAMONDS = 3
  HEARTS = 4
```

Escaping references

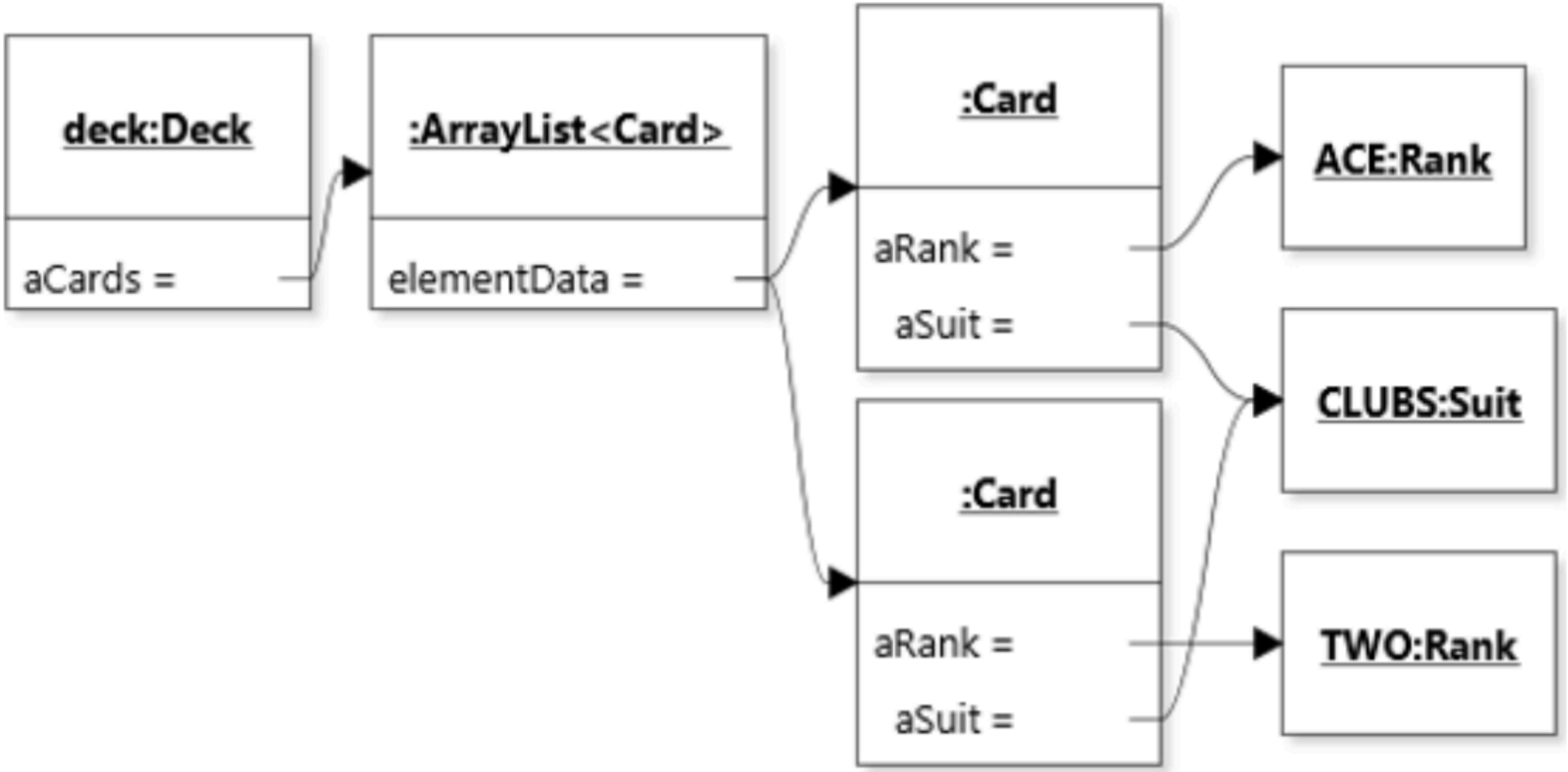
```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  public Deck() {
    /* Add all 52 cards to the deck */
    /* Shuffle the cards */
  }
  public Card draw() {
    return aCards.remove(0);
  }
  public List<Card> getCards() {
     return aCards;
  }
}
```

Problem!

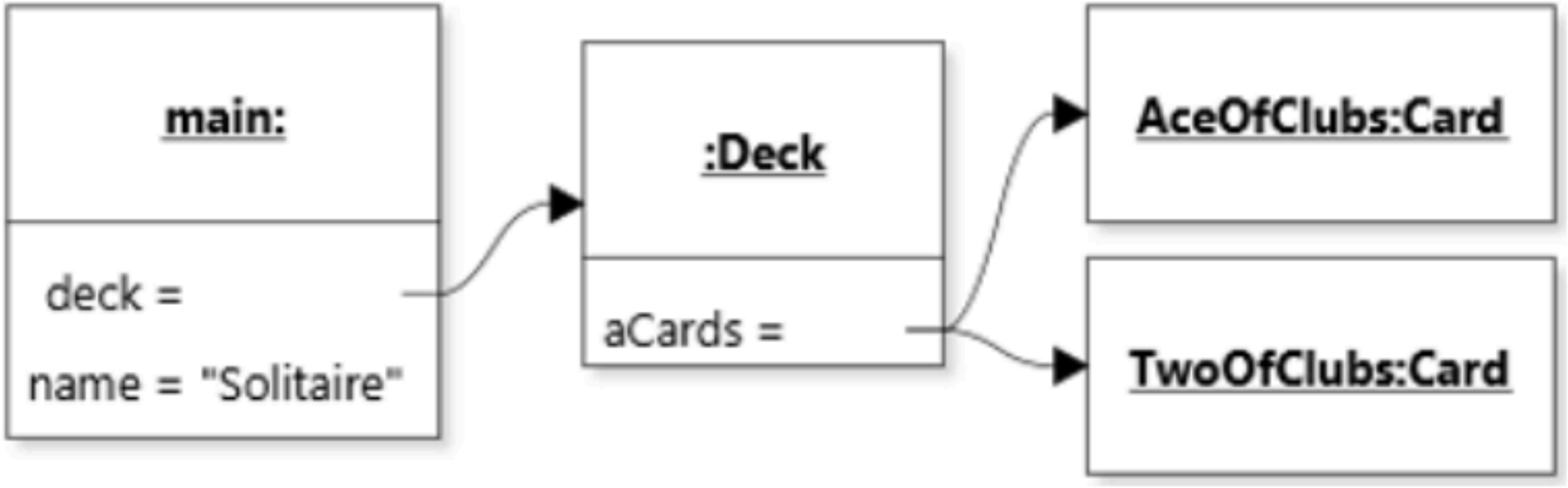
Object diagrams

- A type of UML diagram that represents objects and how they refer to each other. Each rectangle represents an object, with its fields listed.

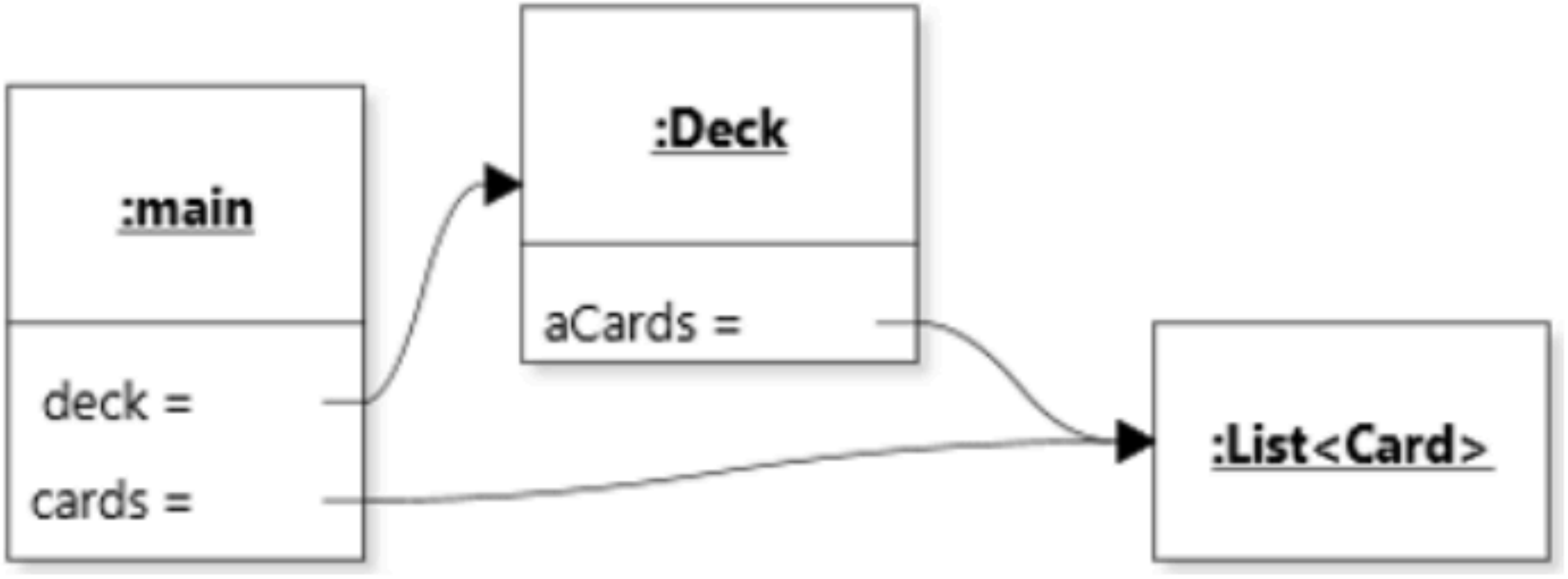
Object diagrams



Object diagrams



Escaping references



Escaping references

- By returning a reference to the private, internal list of cards, it can then be modified outside the class!

```
Deck deck = new Deck();
List<Card> cards = deck.getCards();
cards.add(new Card(Rank.ACE, Suit.HEARTS));
```

- Defeats the purpose of encapsulation.

- Known as the **INAPPROPRIATE INTIMACY** antipattern.

Escaping references

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

Problem!

Escaping references

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

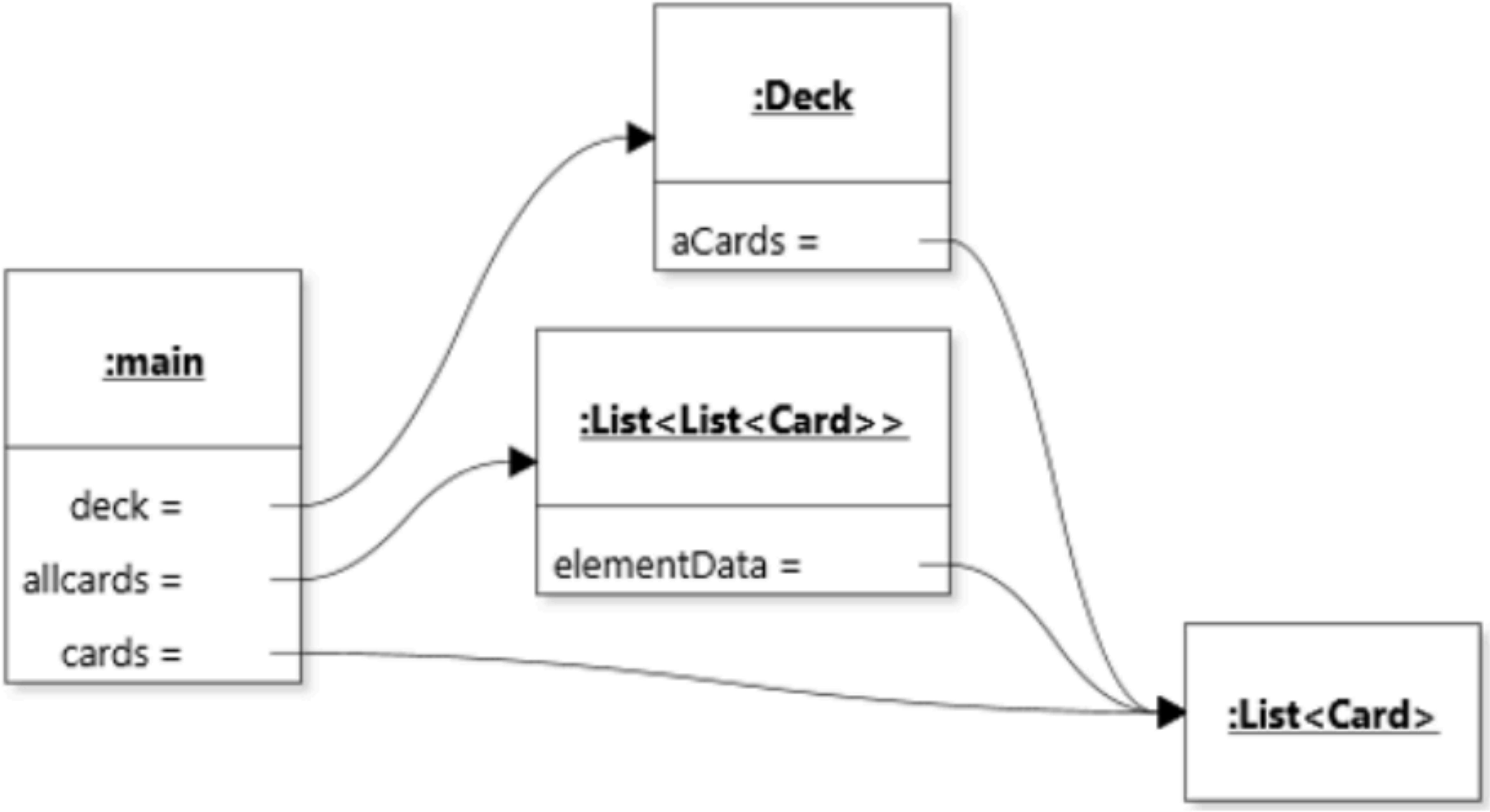
Problem!

Escaping references

```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  public void collect(List<List<Card>> pAllCards) {
    pAllCards.add(aCards);
  }
}
```

Problem!

Escaping references



Immutable objects

- However, it is OK to return a reference to a private object as long as it is **immutable** (i.e., they provide no way to change their state after initialization).
  - E.g., string, or a class like Card that has no public fields nor public methods which update its internal state.

Safely exposing internal data

- How can we expose information without giving the ability for external code to modify our internal data?
  - Extend the interface to include access methods that only return references to immutable objects.
  - Return copies.

Extending the Deck interface

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

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

Assuming Card is immutable.

Return copies

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

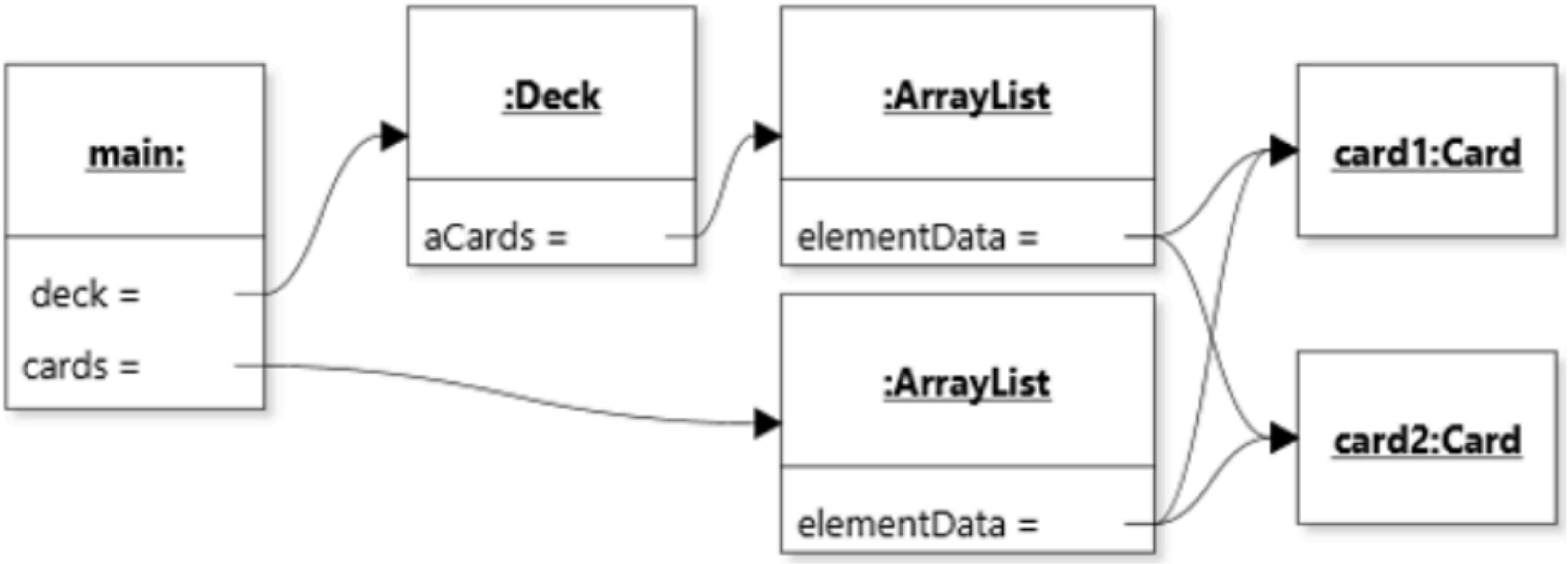
Assuming Card is immutable.

Return copies

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

Assuming Card is immutable.

Return copies



If Card were not immutable...

- Then we would have to make sure each individual Card is copied too.

- We could introduce a copy constructor into Card:

```
public Card(Card pCard) {
  aRank = pCard.aRank;
  aSuit = pCard.aSuit;
}
```

If Card were not immutable...

- Then, to copy a Deck:

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

Input validation

```
Card card = new Card(null, Suit.CLUBS);
```

Input validation

```
/**
* ...
* @throws IllegalArgumentException if pRank
or pSuit is null
*/

public Card(Rank pRank, Suit pSuit) {
  if (pRank == null || pSuit == null) {
    throw new IllegalArgumentException();
  }
  aRank = pRank;
  aSuit = pSuit;
}
```

Input validation

- Aside from arguments, we may also need to validate fields that are used.

- For instance, in this method, remove will throw an exception if aCards is empty. But this is messy, because it results from us passing an invalid input to remove.

  - Exceptions should only be thrown in **unpredictable** situations. But we can predict if a deck is empty!

```java
public Card draw() {
  return aCards.remove(aCards.size() - 1);
}
```

Input validation

```
/**
* ...
* @throws IllegalStateException if the deck
* is empty
*/
public Card draw() {
  if (isEmpty()) {
    throw new IllegalStateException();
  }
  return aCards.remove(aCards.size() - 1);
}
```

Input validation

- With input validation, client code can no longer corrupt the internal values in an object.

- Drawback: we now have to implement error handling in the client code. (And write tests for it!) Sometimes, therefore, it is not warranted to implement.

```
try {
  card = deck.draw();
} catch (IllegalStateException exception) {
  // Recover
}
```

Design by contract

- Suppose we have a constructor:

```
public Card(Rank pRank, Suit pSuit)
```

- How do we know if it performs input validation? What will it do if a value is null? (Recall that enums can be null.)
- There exists an ambiguity, which can cause problems.

Design by contract

- To remove this ambiguity, when writing a method, we will write a **contract** that specifies what should be true about the inputs (and outputs).
  - This takes the form of **preconditions** and **postconditions**.

Design by contract

```java
/**
* @pre pRank != null && pSuit != null
*/
public Card(Rank pRank, Suit pSuit) {
  // ...
}
```

Design by contract

```java
public Card(Rank pRank, Suit pSuit) {
  assert pRank != null && pSuit != null;
  this.aRank = pRank;
  this.aSuit = pSuit;
}
```

Design by contract

```
/**
* @pre pRank != null && pSuit != null
*/
public Card(Rank pRank, Suit pSuit) {
  if (pRank == null || pSuit == null) {
    throw new IllegalArgumentException();
  }
  // ...
}
```

Design by contract in Python

```python
"""
Parameters
----------
pRank: Rank
   The rank of the playing card. Must not be None.
pSuit: Suit
   The suit of the playing card. Must not be None.
"""
def __init__(self, pRank: Rank, pSuit: Suit):
  if None in [pRank, pSuit]:
    raise TypeError("...")
  # ...
```

Design by contract in Python

```
def __init__(self, pRank: Rank, pSuit: Suit):
  assert pRank is not None
  assert pSuit is not None
  self.aRank = pRank;
  self.aSuit = pSuit;
```

Immutability in Python

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Card:
    suit: Suit
    rank: Rank

card = Card(suit=Suit.SPADES, rank=Rank.ACE)
```

Another example: TicTacToe

- What data needs to be stored?
- What computations need to be done?
- What classes should be used?
- What are the relationships between the classes?

References

- Robillard ch. 2 (p.13-41)
  - Exercises #1-9: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter2.md
- Enum Type Tutorial: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- JetUML: https://github.com/prmr/JetUML

Coming up

- Next lecture:
    - Types and polymorphism