McGill

COMP 303

Lecture 4

Types & polymorphism

Winter 2025

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

Announcements

- Teams: deadline Thursday
  - If you made an Ed post and found a partner, please resolve the post.
  - If you would like to be matched with partners, please make a private post.
  - Instructions will be sent out later today about registering.
- Proposal instructions coming out next Monday.

Plan for today

- Recap from last time
- TicTacToe continued
- Types & polymorphism II
- More about generics
- Collections.shuffle and sort
- Comparable<T> interface
- Strategy design pattern
- Class diagrams

Recap

TicTacToe

- Implementation

- Integrating into project

Inheritance vs. interfaces

- Inheritance:
  - for functionality that is shared between classes (is-a relationship).
  - E.g., a dog is a mammal.
- Interfaces:
  - for a shared behaviour that is implemented differently in each implementing class (**subtype**).
  - E.g.: a bird and a bat can both fly, but they do so differently, and a bird is not a mammal while a bat is.

Polymorphism

- There are different types of polymorphism:

  - **subclass** polymorphism: when a class inherits from a parent class, then it can look like the parent class. (We will talk about this later.)

  - **subtype** polymorphism: more general than the above, and includes the case when a class implements an interface (so it can look like that interface) -- meaning that it implements all the operations in it.

  - **parametric** polymorphism: when a function is written that can operate on multiple types, e.g., in Python, or using generics in Java or templates in C++.

Types & polymorphism

Generics

- Allows a **type parameter** to be specified when defining a method, class or interface, which allows objects matching the given type to be accepted.

Generics: Pair

```
public class Pair<T>
{
    final private T aFirst;
    final private T aSecond;

    public Pair(T pFirst, T pSecond)
    {
        aFirst = pFirst;
        aSecond = pSecond;
    }

    public T getFirst() { return aFirst; }
    public T getSecond() { return aSecond; }
}
```

Generics: Pair

```
public class Pair<T>
{
    final private T aFirst;
    final private T aSecond;

    public Pair(T pFirst, T pSecond)
    {
        aFirst = pFirst;
        aSecond = pSecond;
    }

    public T getFirst() { return aFirst; }
    public T getSecond() { return aSecond; }
}
```

```
Pair<Card> cards;
```

Generics: Pair

```
public class Pair<T extends Deck>
{
    final private T aFirst;
    final private T aSecond;

    public Pair(T pFirst, T pSecond)
    {
        aFirst = pFirst;
        aSecond = pSecond;
    }

    public T getFirst() { return aFirst; }
    public T getSecond() { return aSecond; }
}
```

Generics: Pair

```
public class Pair<T extends Deck>
{
    public boolean isTopCardSame()
    {
        Card topCardInFirst = aFirst.draw();
        Card topCardInSecond = aSecond.draw();
        return topCardInFirst.equals(topCardInSecond);
    }
}
```

Collections class

- https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Collections.shuffle

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

Collections.shuffle

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

Collections.shuffle works with any List<T> object.
Very reusable!

Collections.shuffle

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

Collections.shuffle works with any List<T> object.
Very reusable!

Example of parametric polymorphism.

Collections.sort?

```
public class Deck {
   private List<Card> aCards = new ArrayList<>();
   public void sort() {
      Collections.sort(aCards); // ?
   }
}
```

Collections.sort

- Collections.sort tries to compare the elements in the list in order to sort them. Thus, it needs to know a little something about the type.

- In particular, it compares by calling compareTo on each object.

Comparable<T>

- Java defines a Comparable<T> interface that we can implement; this interface specifies the compareTo method. After doing so, we can then call Collections.sort!

- An example of subtype polymorphism.

Comparable<T>

```
public interface Comparable<T>
{
    /*
    * Returns a negative integer, zero, or a
    * positive integer if this object is less
    * than, equal to, or greater than the
    * specified object, respectively.
    */
    int compareTo(T o);
}
```

Interfaces

- In general, interfaces should capture the **smallest cohesive slice** of behaviour that is expected to be used by client code.
  - Cohesive: a set of operations that are logically and conceptually related, which will all be used by implementing classes.

- This principle is tied to **separation of concerns**: the idea that each part of a system should handle a single responsibility.
  - One of the advantages of encapsulation.

Comparing cards

```
public class Card implements Comparable<Card> {
   public int compareTo(Card pCard) {
      // compare the cards
   }
}
```

Comparing cards

```
public class Card implements Comparable<Card> {
   public int compareTo(Card pCard) {
      return aRank.compareTo(pCard.aRank);
   }
}
```

Comparing cards

- What if we wanted to be able to compare in different ways? For example, compare by suit, rank, or suit and then rank?

Comparing cards

```
public class Card implements Comparable<Card> {
  enum ComparisonStrategy {ByRank, BySuit, ByRankThenSuit}
  ComparisonStrategy aStrategy; // set in constructor
  public int compareTo(Card pCard) {
    switch (aStrategy) {
      case ByRank:
        return compareByRank(pCard);
      case BySuit:
        return compareBySuit(pCard);
      case ByRankThenSuit:
        return compareByRankThenSuit(pCard);
      default:
        throw new IllegalStateException("Unexpected
comparison strategy: " + aStrategy);
    }
  }
}
```

Comparing cards

```
public class Card implements Comparable<Card> {
  enum ComparisonStrategy {ByRank, BySuit, ByRankThenSuit}
  ComparisonStrategy aStrategy; // set in constructor
  public int compareTo(Card pCard) {
    switch (aStrategy) {
      case ByRank:
        return compareByRank(pCard);
      case BySuit:
        return compareBySuit(pCard);
      case ByRankThenSuit:
        return compareByRankThenSuit(pCard);
      default:
        throw new IllegalStateException("Unexpected
comparison strategy: " + aStrategy);
    }
  }
}
```

Anti-pattern: SWITCH STATEMENT

Collections.sort

```
static <T extends Comparable<? super T>> void
sort(List<T> list)
```

```
public static <T> void
sort(List<T> list, Comparator<? super T> c)
```

Comparing cards

```
public class ByRankComparator implements Comparator<Card>
{
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
}
```

Comparing cards

```
public class BySuitComparator implements Comparator<Card>
{
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getSuit().compareTo(pCard2.getSuit());
  }
}
```

Comparing cards

```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  public void sort() {
    Collections.sort(aCards, new ByRankComparator());
  }
}
```

STRATEGY design pattern

- If we have a bunch of algorithms to accomplish a task, and want to switch between them flexibly.
  - E.g., switching between different Card comparisons.
  - Or, different AI implementations for a card game bot.
- The main idea is to be able to switch without the client needing to know the algorithm implementation; the algorithms should be interchangeable.

Comparing cards

```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  public void sort() {
    Collections.sort(aCards, new ByRankComparator());
  }
}
```

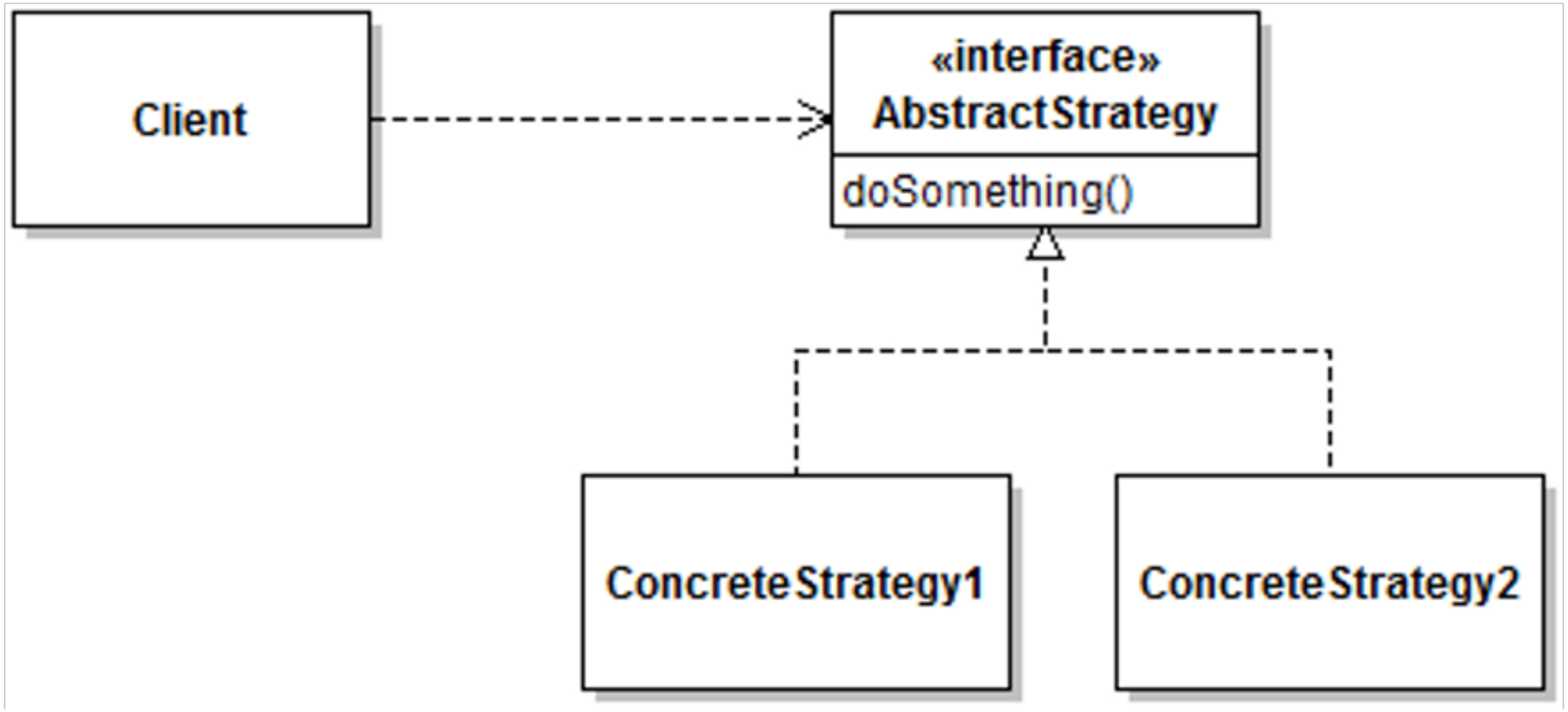
Comparing cards

```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  public void sort() {
    Collections.sort(aCards, new BySuitComparator());
  }
}
```

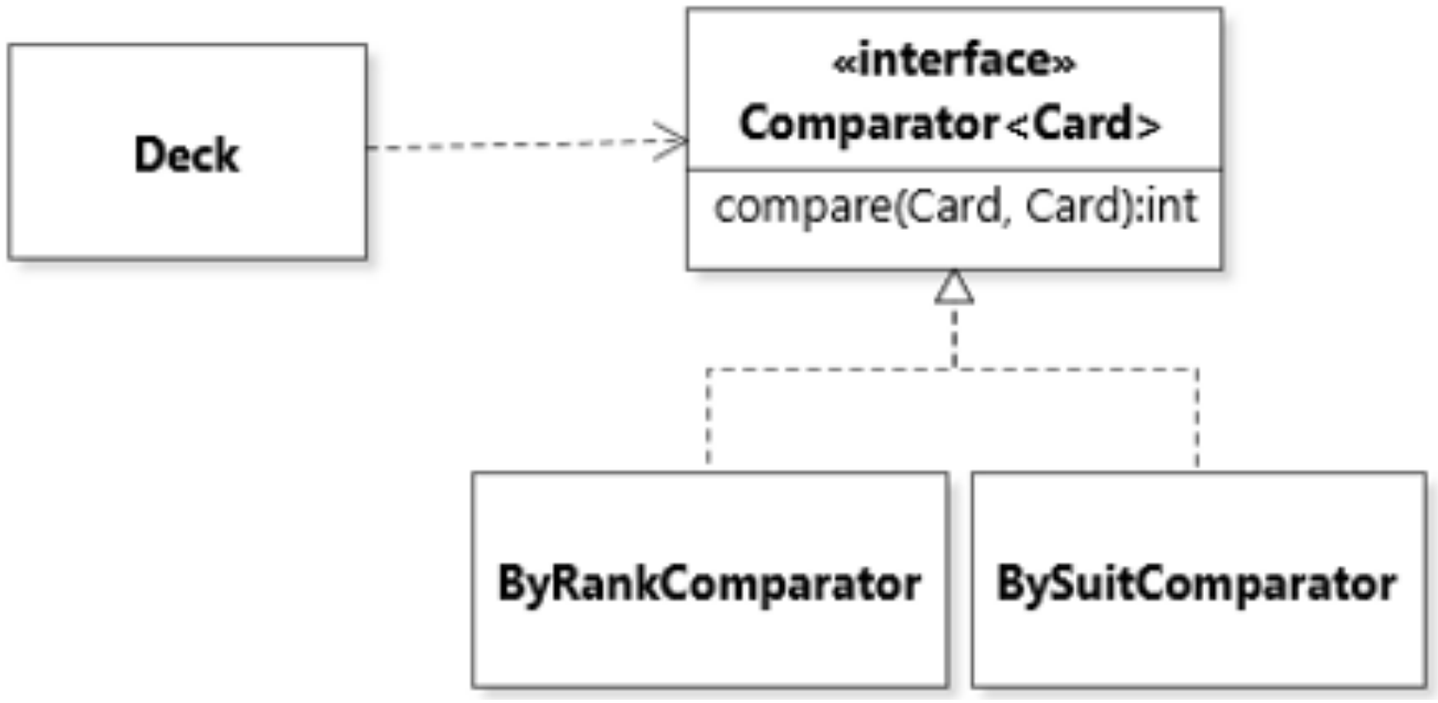
Comparing cards

```
public class Deck {
  private List<Card> aCards = new ArrayList<>();
  private Comparator<Card> aComparator;
  public Deck(Comparator<Card> pComparator) {
    aComparator = pComparator;
    shuffle();
  }
  public void sort() {
    Collections.sort(aCards, aComparator);
  }
}
```

STRATEGY design pattern



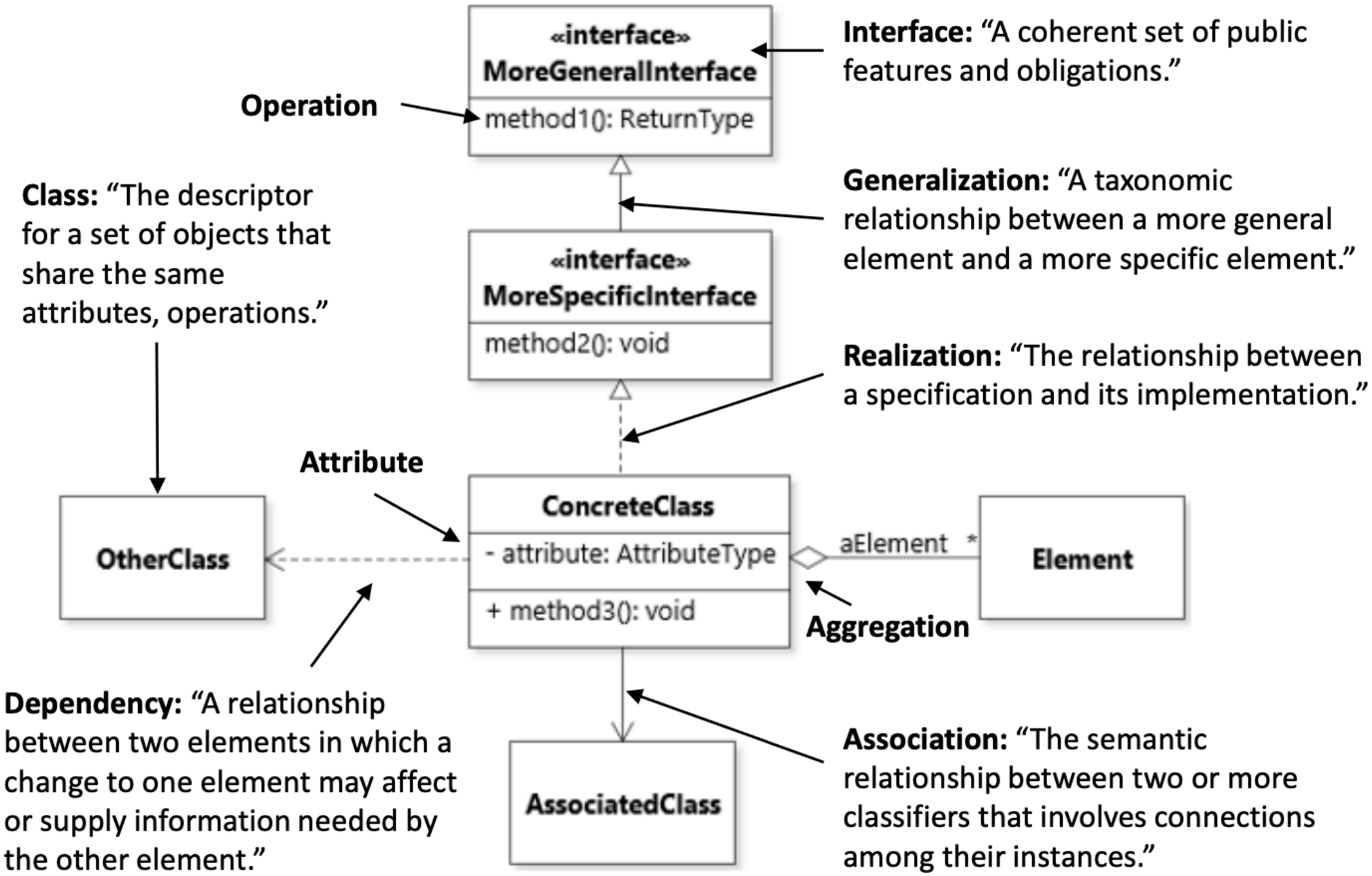
STRATEGY design pattern



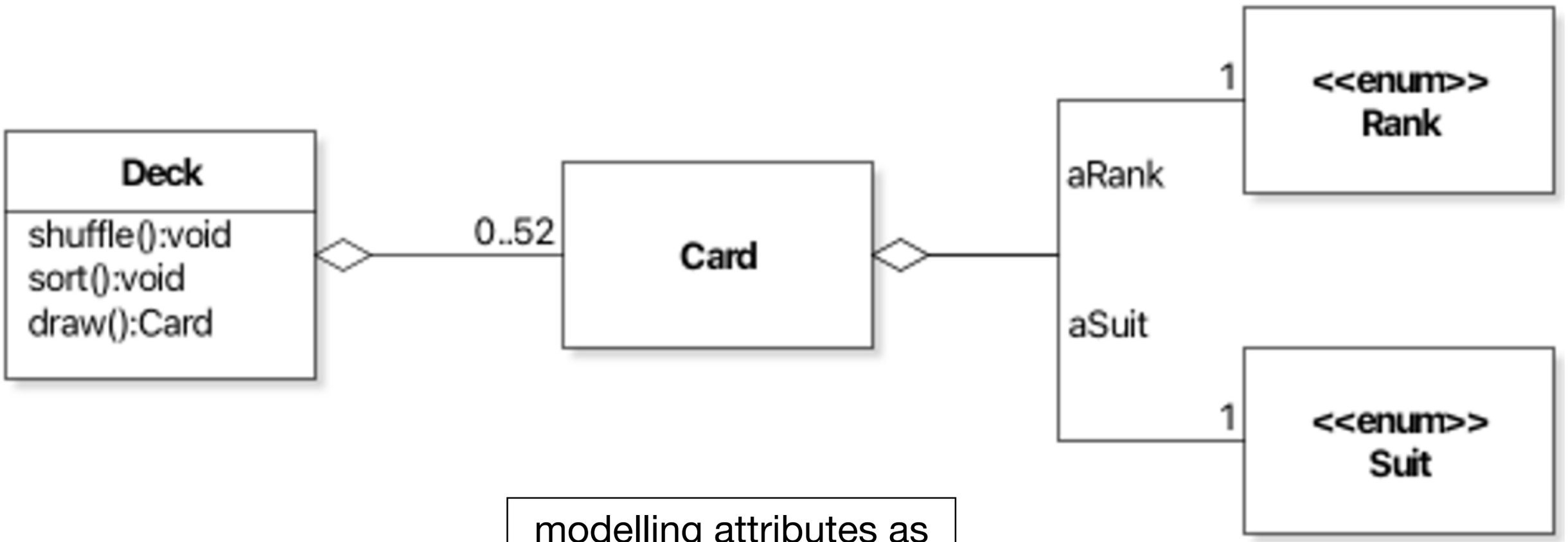
Python implementation

- MovementStrategy for an NPC class.

Class diagrams



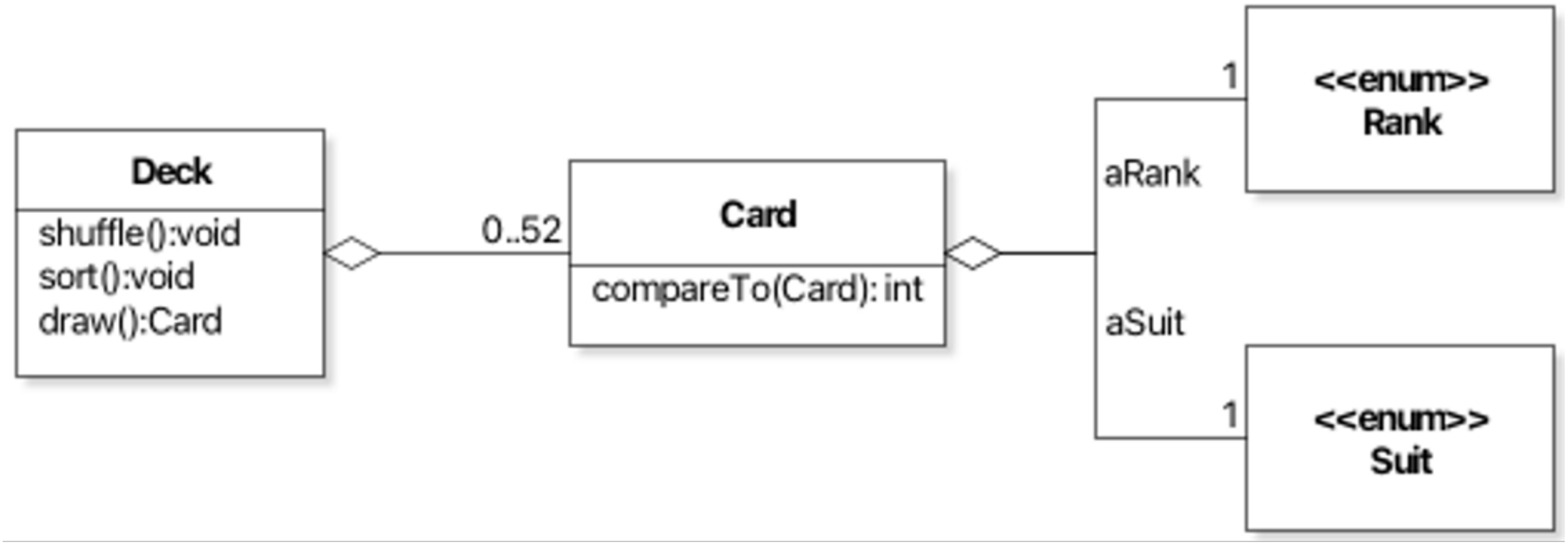
Class diagrams



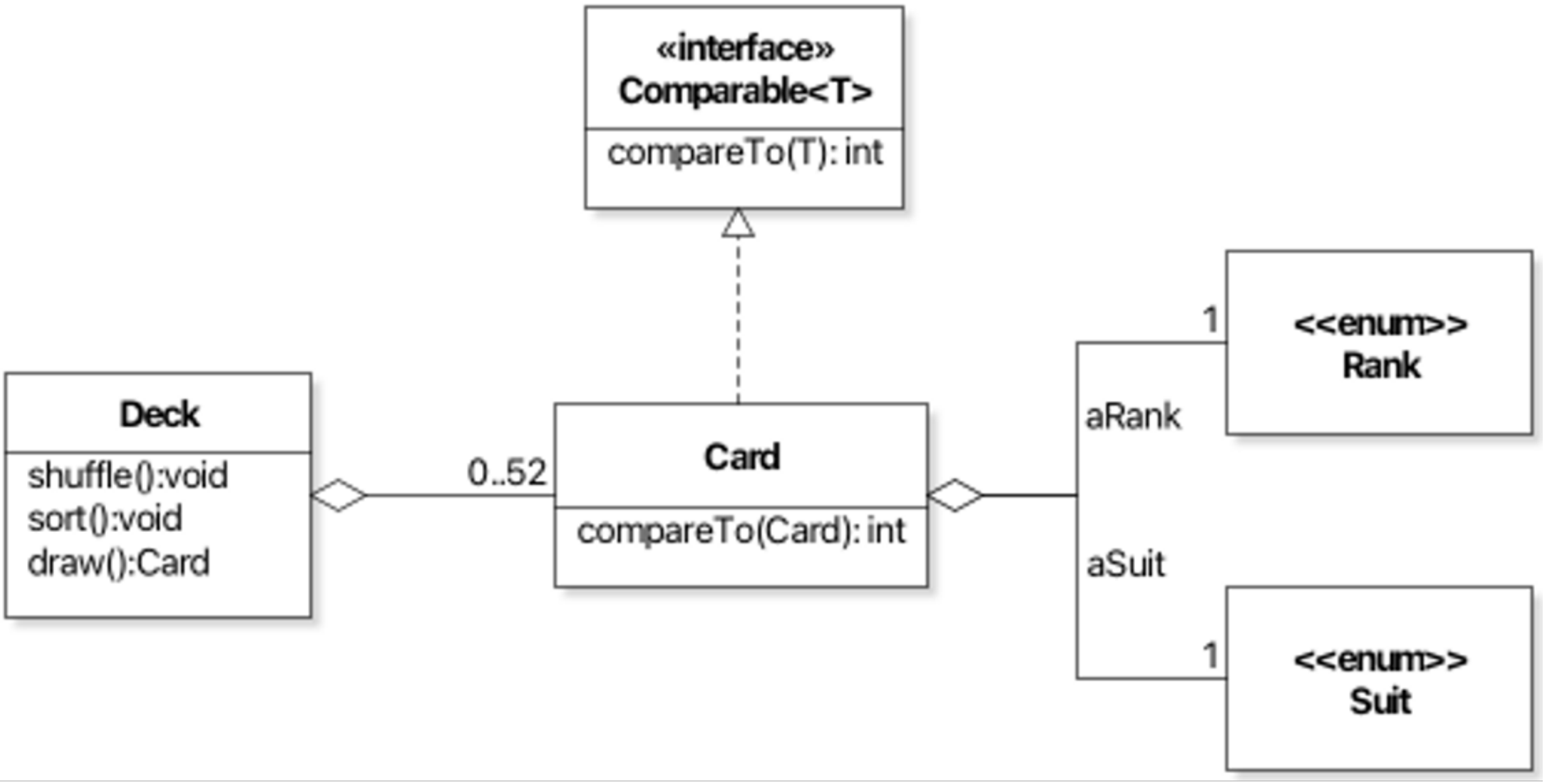
modelling attributes as aggregations

omitting constructor and getter methods

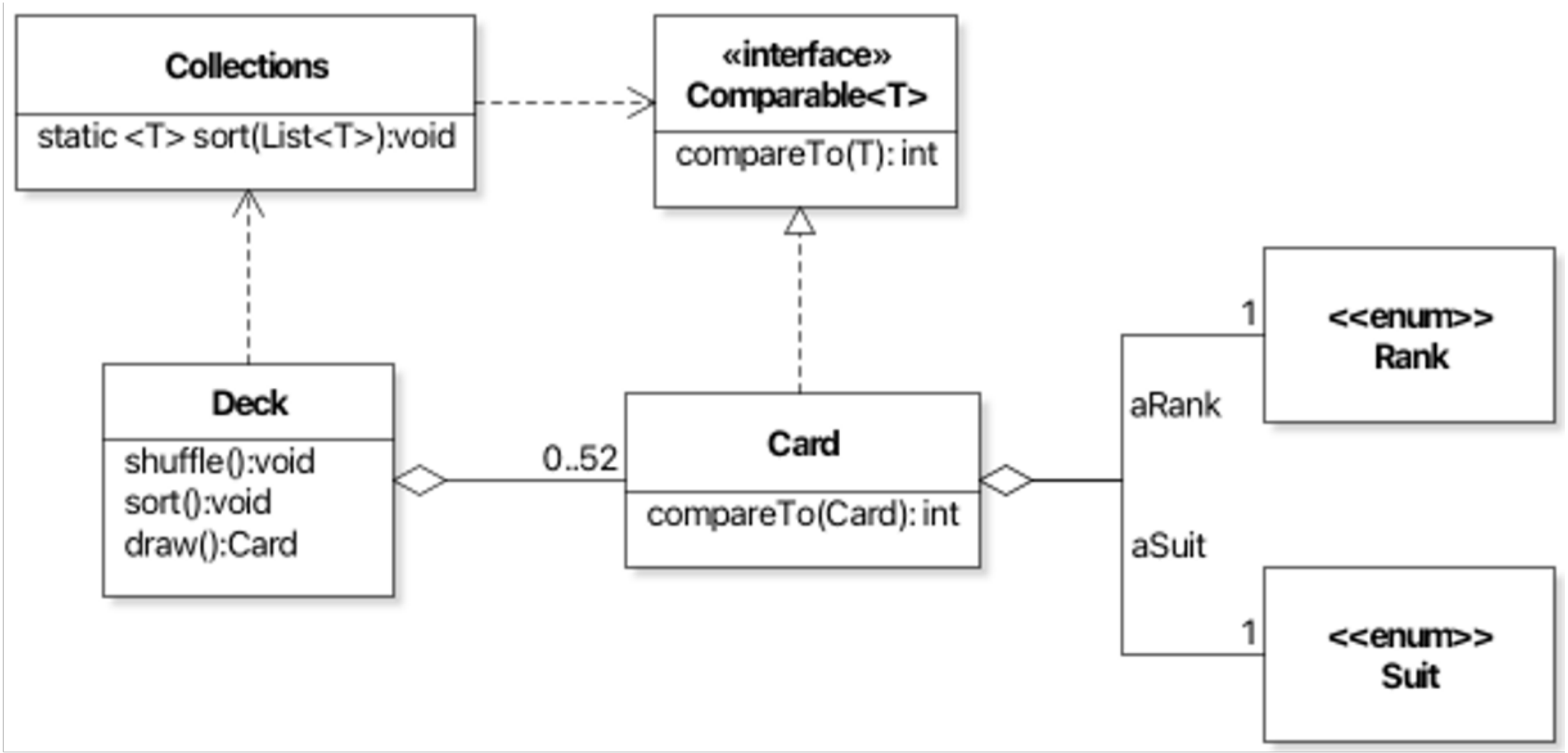
Class diagrams



Class diagrams



Class diagrams



Comparators as top-level classes

```
public class ByRankComparator implements Comparator<Card>
{
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
}
```

Won't have access to private Card fields.
Have to use getter methods.

Comparators as nested classes

```
public class Card
{
  static class ByRankComparator implements Comparator<Card> {
    public int compare(Card pCard1, Card pCard2) {
      return pCard1.getRank().compareTo(pCard2.getRank());
    }
  }
}

// Collections.sort(aCards, new Card.CompareBySuitFirst());
```

Now has access to private fields.

Comparators as anonymous classes

```
public class Deck {
   public void sort() {
      Collections.sort(aCards, new Comparator<Card>() {
         public int compare(Card pCard1, Card pCard2) {
            /* Comparison code */
         }
      });
   }
}
```

Good if only used in one place.
But, can't store state.

Comparators as anonymous classes

```
public class Deck {
   public void sort() {
      Collections.sort(aCards, new Comparator<Card>() {
         public int compare(Card pCard1, Card pCard2) {
            /* Comparison code */
         }
      });
   }
}
```

Good if only used in one place.
But, can't store state.

No need for constructor since we are implementing an interface.

Nested class

- A nested class is any class defined inside another (including anonymous classes).

- Why use them?

- If a class is useful to only one other class, then it is logical to embed it into that class.

- It increases encapsulation. By hiding class B inside class A, A's members can be declared private and B can access them; B itself can be hidden from the outside world.

- It places code closer to where it is used.

Iterators

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

Even though it returns an unmodifiable version
(so no escaping reference),
it still leaks the internal representation of a Deck.

ITERATOR design pattern

- A mechanism to iterate over the elements stored in an aggregate object, e.g., the cards in a Deck, **without exposing the internal representation** (list, array, etc.).

- It is done by implementing the Iterable<E> interface.

Iterators

- Java classes like the ArrayList already implement Iterator<E>:

```
List<String> stringList = new ArrayList<>();
stringList.add("Apple");
stringList.add("Banana");
stringList.add("Cherry");

Iterator<String> iterator = stringList.iterator();

// Use the iterator to traverse the list
while (iterator.hasNext()) {
    String element = iterator.next();
    System.out.println(element);
}
```

Iterators

- Java classes like the ArrayList already implement Iterator<E>:

```
List<String> stringList = new ArrayList<>();
stringList.add("Apple");
stringList.add("Banana");
stringList.add("Cherry");



for (String string : stringList) {
    System.out.println(string);
}
```

Implementing Iterable<E>

```
public class Deck implements Iterable<Card>
{
    private List<Card> aCards;
    public Iterator<Card> iterator() {
        return aCards.iterator();
    }
}

// later
for (Card card : deck) {
    System.out.println(card);
}
```

Iterator<E>

- We haven't discussed exactly how iteration is done for the built-in types like ArrayList.

- That's because, normally, when we want to iterate, we can just use the standard implementation.

- But if we want to write our own iteration code, we can implement the Iterator<E> interface (not covered here).

References

- Robillard ch. 3.2-3.7 (p. 46-60)

  - Exercises 1-10: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter3.md

Coming up

- Next lecture:
  - More about types and polymorphism