Jin L.C. Guo

M2 (b) - Types and Polymorphism

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

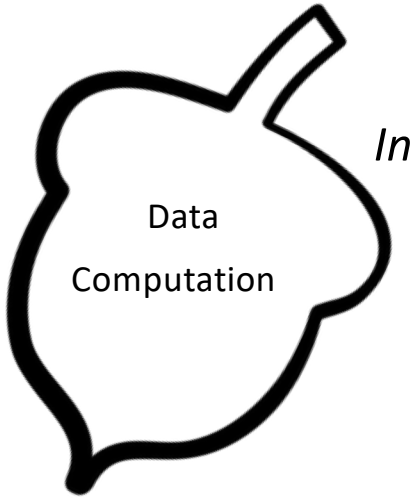
Recap

- A Class's Interface

- An interface type

Recap

- A Class's Interface



Interface

```
public final class Deck
{
  private  Stack<Card> aCards = new Stack<>();
  ...

  public Deck( Deck pDeck ) {...}

  public void shuffle() {...]
  public Card draw() {...}
  public boolean isEmpty() {...}
}
```

```
Deck deck = new Deck();

if(!deck.isEmpty())
        deck.draw();
```

Recap

- An Interface Type

```
public interface CardSource {
    Card draw();
    boolean isEmpty();
}

public class Deck implements CardSource
{
  private  Stack<Card> aCards = new Stack<>();
  ...

  public Deck( Deck pDeck ) {...}

  public void shuffle() {...]
  public Card draw() {...}
  public boolean isEmpty() {...}
}
```

Deck are substitutable for **CardSource**

```
CardSource deck = new Deck();

if(!deck.isEmpty())
        deck.draw();
```

Recap

- Subtype is about substation:
  - B is a subtype of A means that if whenever the context requires an element of type A it can accept an element of type B.

- Subclass is about inheritance:
  - B is a subclass of A means that B can reuse unchanged fields and methods from A.
  - Extra dependencies between A and B
  - More in Module-6 (Week 7)

```
public class SpecialDeck extends Deck {

        … …
}
```

Objective

- Generic Basics
- Nested Classes
- Java Comparator Interface
- Function objects
- Java Iterator Interface

Generics

```
public interface ListOfNumbers {
    boolean add(Number pElement);
    Number get(int index);
}

public interface ListOfIntegers {
boolean add(Integer pElement);
Integer get(int index);
}


… …
```

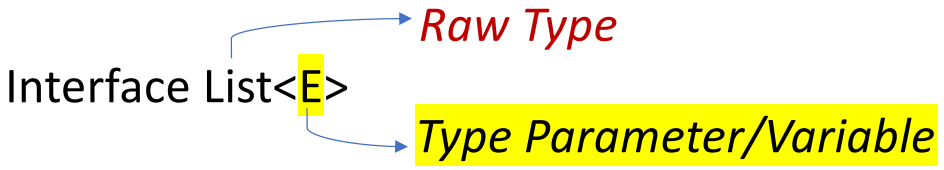
Generics

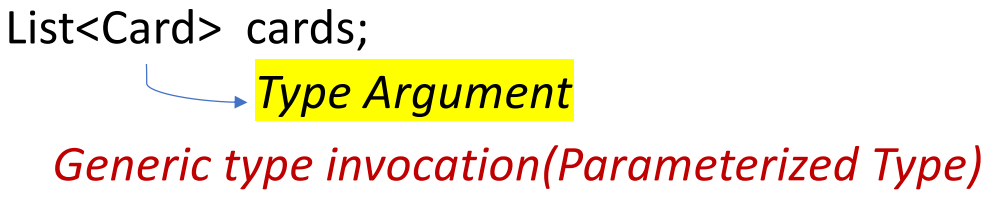
- Purpose: make the code reusable for many different types

```
public interface List<E> {
    boolean add(E pElement);
    E get(int index);
}
```

Generics

- Generic Types
  - A class or interface whose declaration has one or more type parameter





Convention:

E for Element
K for Key
V for Value
T for Type

(Don't use only raw type during generic type invocation)

```
Pair<Card> pair =
    new Pair<>(new Card(Rank.FIVE, Suit.CLUBS),
               new Card(Rank.FOUR, Suit.CLUBS));
Card card1 = pair.getFirst();
```

Type Inferred by Compiler

```
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

- Generic Method
  - A method that takes type parameters

emptySet method in java.util.Collections:

```
static <T> Set<T> emptySet()
```

Type Parameter

Between Modifier and Return Type

Activity 1

- Write a generic method that add elements of Pair in any type to a collection of the same type.

Interface Collection<E>

```
boolean add(E e)
```

```
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; }

}
```

Activity 1

- Write a generic method that add elements of Pair in any type to a collection of the same type.

```
/*
* Add the elements of type T stored in Pair to a Collection of Type T
* @pre pair !=null && collection != null
* @pre pair.getFirst()!=null && pair.getSecond()!=null
* @post collection.contains(pair.getFirst()) && collection.contains(pair.getSecond())
*
* @see Pair
*/
static <T> void fromPairToCollection(Pair<T> pair, Collection<T> collection) {
    /* assertion on pre conditions*/
    collection.add(pair.getFirst());
    collection.add(pair.getSecond());
    /* assertion on post conditions*/
}
```

Adding Restriction on Type Variables

```
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; }

}
```

Adding Restriction on Type Variables

```
public class Pair<T extends CardSource>
{
   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; }

   public boolean isTopCardSame()
   {
       Card topCardInFirst = aFirst.draw();
       Card topCardInSecond = aSecond.draw();
       return topCardInFirst.equals(topCardInSecond);
   }
}
```

Type can only be CardSource or its subtype

call methods of CardSource

Generic Method With Type Bound

```
static <T extends CardSource>
   void fromPairToCollection(Pair<T> pair, Collection<T> collection) {}
```

Generic Method With Type Bound

- How to increase flexibility and usability?

```
static <T>
    void fromPairToCollection(Pair<T> pair, Collection<T> collection) {}
```

Generic Method With Type Bound

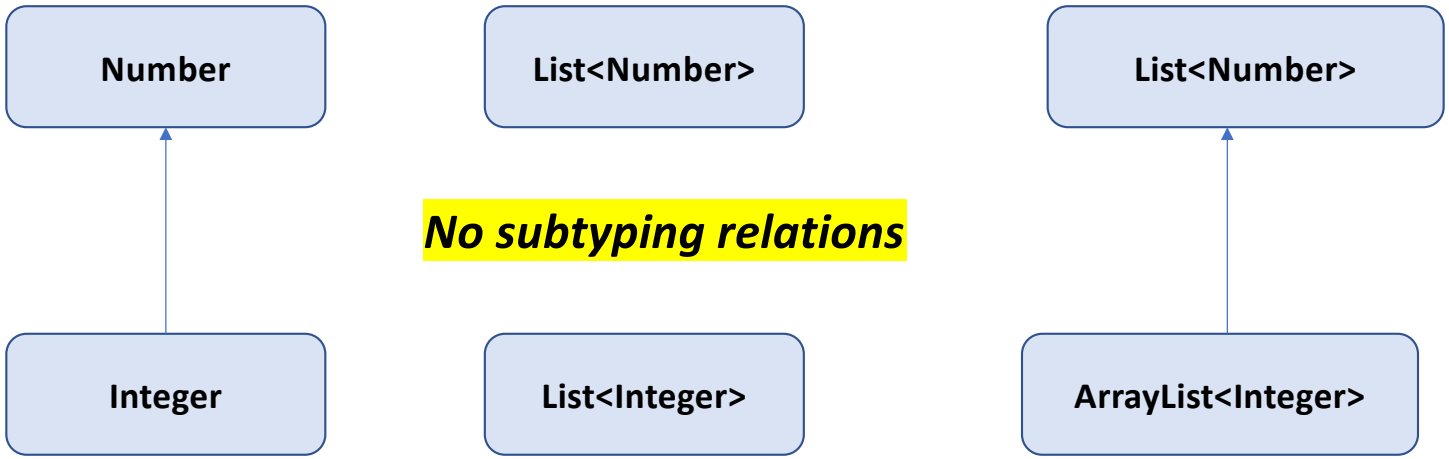
- How to increase flexibility and usability?

```
static <T, T2 extends T>
   void fromPairToCollection(Pair<T2> pair, Collection<T> collection) {}
```

Equivalent with using wildcard

```
static <T>
   void fromPairToCollection(Pair<? extends T> pair, Collection<T> collection) {}
```

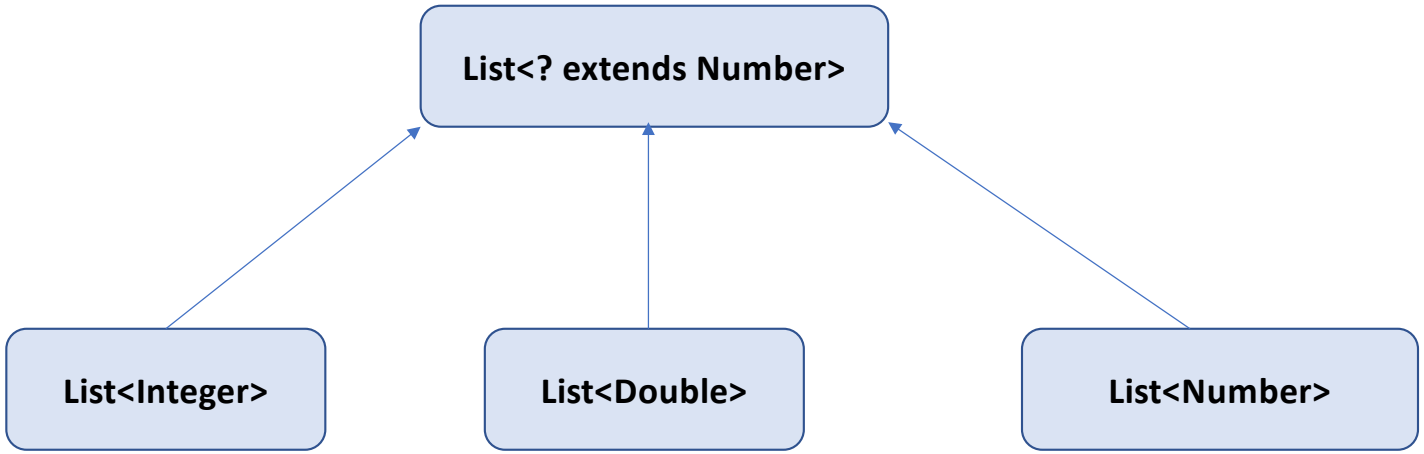
Generic Types and Subtyping




```
List<Number> numberList = new ArrayList<Integer>(); // Compile Error
double doubleNumber = 0.5;
numberList.add(doubleNumber); // You shouldn't be able to do this
```

Generic Types and Subtyping

- With Wildcard




```
static <T>
   void fromPairToCollection(Pair<? extends T> pair, Collection<T> collection) {}
```

More on Wildcard

<?>	*Unknown Type*
<? *extends* Number>	*A Type that is Number or its subtype*
<? *super* Number>	*A Type that is Number or its supertype*

Activity 2:

Find the subtypes from the right for the wildcard types:

List<? extends Number>

List<? super Number>

Collection<Integer>

ArrayList<Integer>

List<Object>

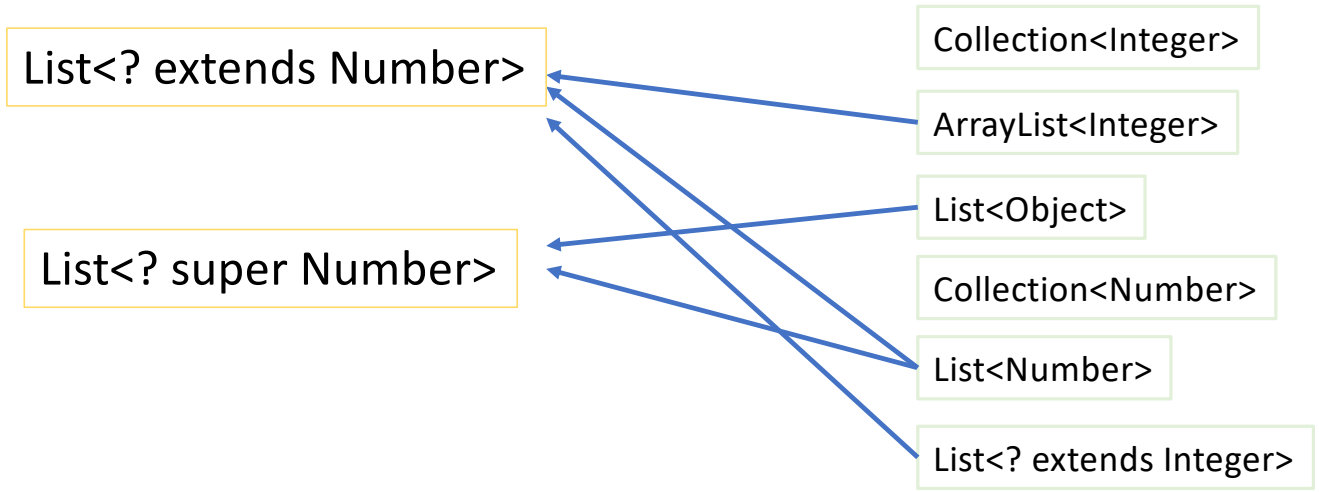
Collection<Number>

List<Number>

List<? extends Integer>

Activity 2:

Find the subtypes from the right for the wildcard types



Back to the sort method for comparable types

- In java.util.collections

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

```
class Card implements Comparable<Card> {…}

class FancyCard extends Card {…}
```

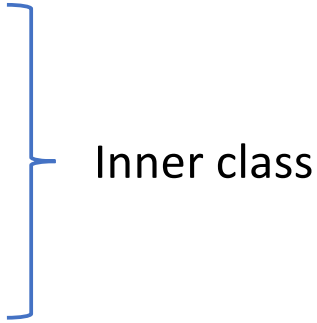
```
List<FancyCard> fancyCardList = new ArrayList<>();

Collections.sort(fancyCardList);
```

Objective

- Generic Basics
- Nested Classes
- Java Comparator Interface
- Function objects
- Java Iterator Interface

Review of Nested Classes in Java

- Classes defined within another class
  - Static member class
  - Non-static member class
  - Local class
  - Anonymous class



Static Member Class

StaticMemberClass do NOT have access to other non-static members of the enclosing class.

```
class OuterClass {
    ...
    static class StaticMemberClass {
        ...
    }
 }
```

```
OuterClass.StaticMemberClass nestedObject
        = new OuterClass.StaticMemberClass();
```

Non-Static Member Class

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

```
OuterClass.InnerClass innerObject =
        outerObject.new InnerClass();
```

Instance of **InnerClass** exists only within an instance of **OuterClass,** and has access to the methods and fields of its enclosing instance.

Local Class

- An inner class that is defined in a block

```
class OuterClass
{
    public void method()
    {
        class LocalClass {
              ……
        }
        LocalClass instance = new LocalClass();
    }

}
```

LocalClass has access to local variables that are final or effectively final

Anonymous Class

- A inner class that is declared and instantiated at the same time.

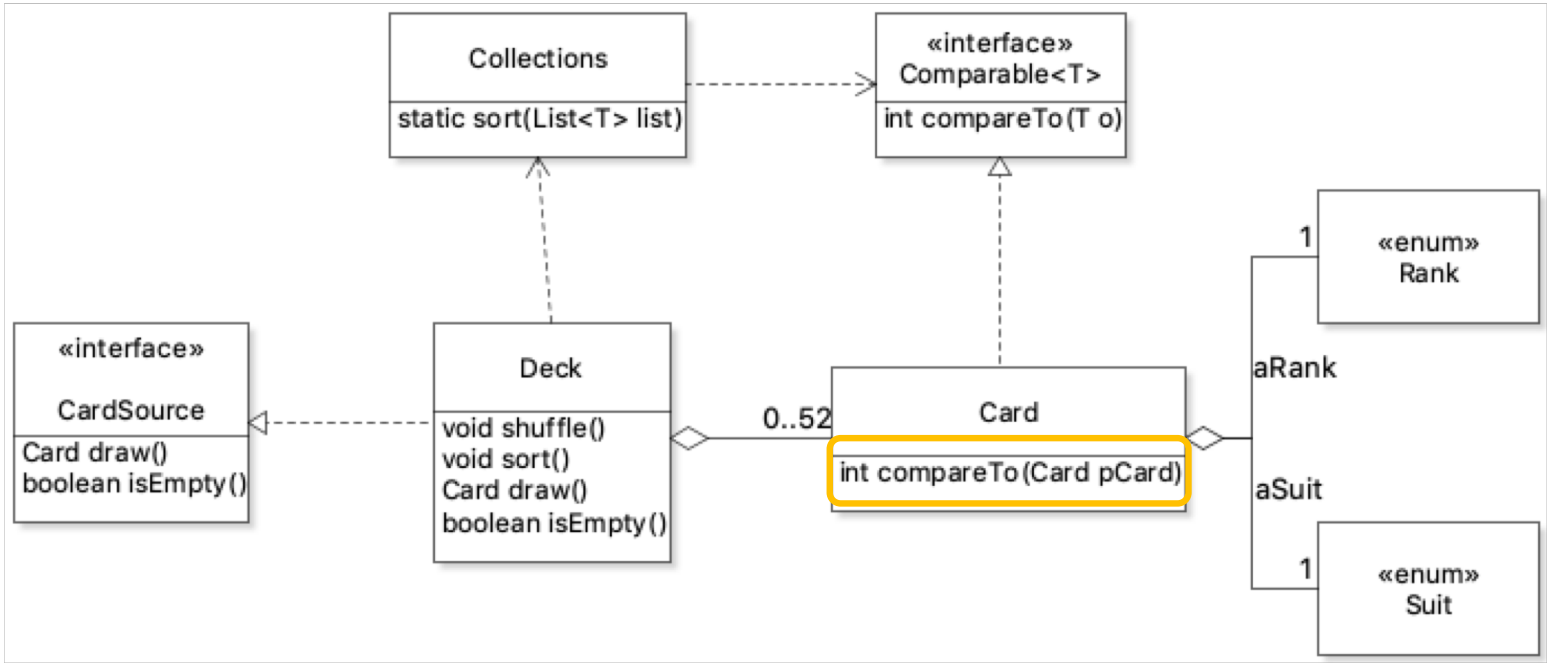
```
class OuterClass
{
    public void method()
    {
        SuperType instance = new SuperType() {
                ⋯ ⋯
        };
    }

}
```

Anonymous Class has access to local variables that are final or effectively final

Objective

- Generic Basics
- Nested Classes
- Java Comparator Interface
- Function objects
- Java Iterator Interface

Current Design of Deck



Comparator Interface

- **Interface Comparator<T>**

```
public int compare(T o1, T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

```
@Override
public int compare(Card pCard1, Card pCard2) {
	return pCard1.compareTo(pCard2);
}
```

ByRank Comparator

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

}
```

BySuit Comparator

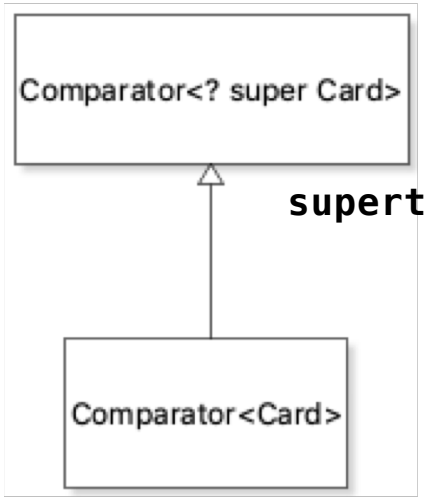
```
public class BySuitComparator implements Comparator<Card>
{
    @Override
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getSuit().compareTo(pCard2.getSuit());
    }
}
```

Another sort method

- In java.util.collections

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


Collections.sort(aCards, new ByRankComparator());
```

List<Card>



Objective

- Generic Basics
- Nested Classes
- Java Comparator Interface
- Function objects
- Java Iterator Interface

Function Object

- An interface with single abstract method
- The actual function is achieved by the object of a class which implements that interface

Is the function is only used once?

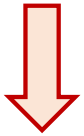
Does the function need to access the private field?

Should the function have state?

Anonymous Class for Function Object

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

}
```

```
Collections.sort(aCards, new ByRankComparator());
```



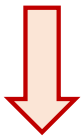
Interface to implement or class to extend

```
Collections.sort(aCards, new Comparator<Card>() {
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getRank().compareTo(pCard2.getRank());
    }
});
```

More Concise Option

- Lambda Expression

```
Collections.sort(aCards, new Comparator<Card>() {
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getRank().compareTo(pCard2.getRank());
    }
});
```



```
Collections.sort(aCards, (pCard1, pCard2) ->
    pCard1.getRank().compareTo(pCard2.getRank()));
```

Enable access to the private field

```
public class Card
{
    public static Comparator<Card> createByRankComparator()
    {
        return new Comparator<Card>()
            {
                @Override
                public int compare(Card pCard1, Card pCard2) {
                        return pCard1.aRank.compareTo(pCard2.aRank);
            }
        };
    }
}
```

Activity 3

- Design a UniversalComarator that can compare two cards with more than one strategies including by rank, suit, reversed rank, suit first then rank.

Objective

- Generic Basics
- Nested Classes
- Java Comparator Interface
- Function objects
- Java Iterator Interface

Iterate Collections

- Another common service needed for collections

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

        … …          Information Leaking

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



}
```

```
for(Card card:deck.getCards())
{
        /* do something using card*/
}
```

Iterator Interface

- Interface Iterator<E>

  E - the type of elements returned by this iterator

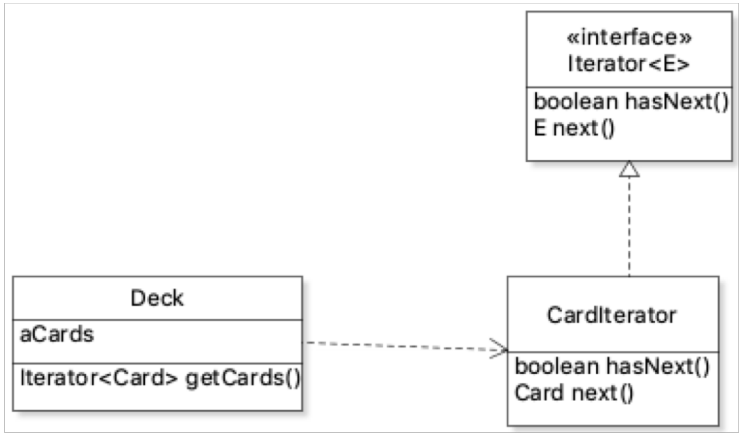
  ```
  boolean hasNext();
  ```

  Returns true if the iteration has more elements.

  ```
  E next();
  ```

  Returns the next element in the iteration.

Iterator Interface

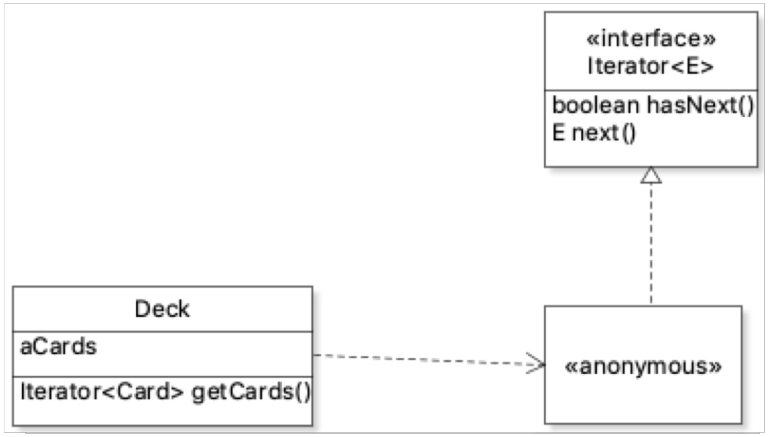



```
public class Deck
{
        private List<Card> aCards = new ArrayList<>();

        … …

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


}
```

```
Iterator<Card> iterator = deck.getCards();
while(iterator.hasNext())
{
    Card card = iterator.next();
    /* do something using card*/
}
```

```
for(Card card : deck.getCards())
{
    /* do something using card*/
}
```

Iterator Interface




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

        … …

        public Iterator<Card> getCards()
        {
                /* Get iterator for aCards */
                return aCards.iterator();
        }



}
```

```
Iterator<Card> iterator = deck.getCards();
while(iterator.hasNext())
{
    Card card = iterator.next();
    /* do something using card*/
}
```

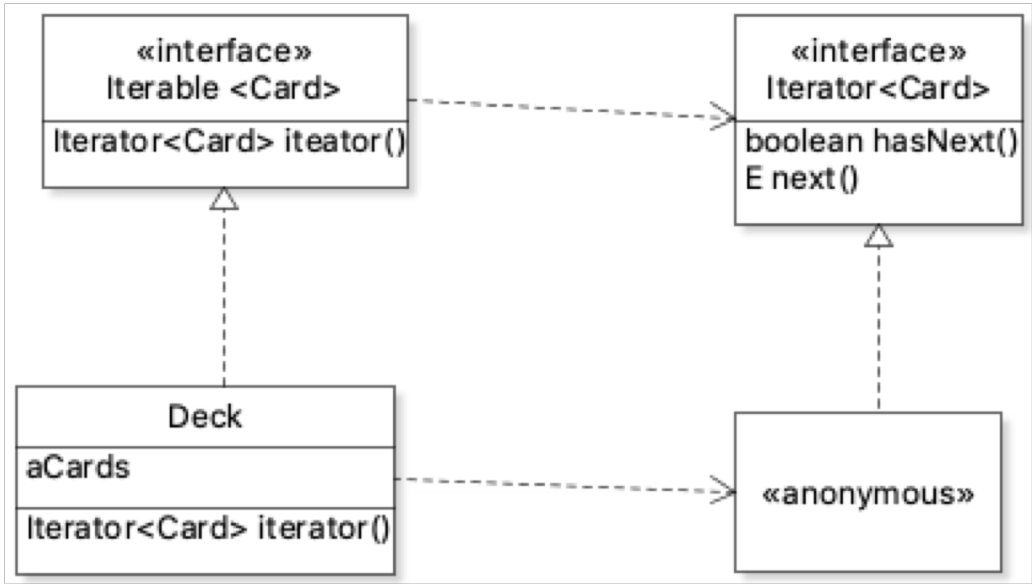
Encapsulate Iterable Behavior

- **Interface Iterable<T>**

  T - the type of elements returned by the iterator

  ```
  public Iterator<T> iterator()
  ```

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

        … …


    @Override
    public Iterator<Card> iterator() {
        return aCards.iterator();
    }
}
```


«interface»
Iterable <Card>
Iterator<Card> iteator()
«interface»
Iterator<Card>
boolean hasNext()
E next()
Deck
aCards
Iterator<Card> iterator()
«anonymous»