Client.java

```
public class Client {
   public static void main(String[] args) {
      CardHand ch = new CardHand();

      for(int i = 0; i < 0; i++) {
         ch.addCard(new Card(Rank.RandomRank(), Suit.RandomSuit()));
      }

      for(Object o : ch) {
         Card c = (Card)o;
         System.out.println(c);
      }

      System.out.println("" + ch);
   }
}
```

Card.java

```
public class Card {
    public Card(int rank, int suit) {
        this.rank = rank;
        this.suit = suit;
    }

    private int suit, rank;

    int getSuit() { return suit; }
    int getRank() { return rank; }

    void setSuit(int suit) { this.suit = suit; }
    void setRank(int rank) { this.rank = rank; }

    @Override
    public String toString() {
        String ret = "Card[Rank: ";
        ret += rank;
        ret += ", Suit: ";
        ret += Suit.getString(suit);
        ret += "]";

        return ret;
    }

}
```

CardHand.java

```
import java.util.*; // Iterator

/**
 * @author joey
 */
public class CardHand implements Iterable {

    private LinkedPositionalList<Card> plist = new LinkedPositionalList<>();

    private Position<Card> pos_Diamond = null;
    private Position<Card> pos_Heart   = null;
    private Position<Card> pos_Spade   = null;
    private Position<Card> pos_Club    = null;

    private int num_cards = 0;

    @Override
    public String toString() {
        String ret = "CardHand[";
        Position<Card> tmp = plist.first();

        while(tmp != null) {
            ret += tmp.getElement().toString();
            ret += ",";
            tmp = plist.after(tmp);
        }
        ret += "]";

        return ret;
    }

    @Override
    public Iterator iterator() {
        return new card_iterator();
    }

    public class card_iterator implements Iterator {

        Position<Card> pcard;
        int suit = 0;
        boolean suit_specific;

        public void setPosition(Position<Card> pcard) {
            this.pcard = pcard;
            this.suit = pcard.getElement().getSuit();
        }

        public card_iterator() {
```

```
        setPosition(plist.first());
        suit_specific = false; // default iterates over entire hand
    }

    @Override
    public boolean hasNext() {
        Position<Card> card = plist.after(pcard);

        if(suit_specific) {
            if(card == null || card.getElement().getSuit() != suit)
                return false;
        } else {
            if(card == null)
                return false; // only check presence
        }
        return true;
    }

    @Override
    public Object next() {
        Object o = pcard.getElement();
        pcard = plist.after(pcard);
        return o;
    }
}

public CardHand() {

}

public void addCard(Card c) {
    int s = c.getSuit();

    Position<Card> tmp_position = null;

    switch(s) {
        case Suit.Diamond:
            if(pos_Diamond != null)
                pos_Diamond = plist.addAfter(pos_Diamond, c);
            else
                pos_Diamond = plist.addLast(c);
            break;
        case Suit.Club:
            if(pos_Club != null)
                pos_Club = plist.addAfter(pos_Club, c);
            else
                pos_Club = plist.addLast(c);
            break;
        case Suit.Heart:
```

```
            if(pos_Heart != null)
               pos_Heart = plist.addAfter(pos_Heart, c);
            else
               pos_Heart = plist.addLast(c);
            break;
         case Suit.Spade:
            if(pos_Spade != null)
               pos_Spade = plist.addAfter(pos_Spade, c);
            else
               pos_Spade = plist.addLast(c);
            break;
         default:
            break;
      }
   }
}
```

Suit.java

```
import java.util.*;

public class Suit {
   public static final int Diamond = 1;
   public static final int Heart   = 2;
   public static final int Spade   = 3;
   public static final int Club    = 4;

   public static int RandomSuit() {
      Random rand = new Random();
      return Math.abs((rand.nextInt() % 4) + 1); // modulo operation gives 0 - 3, plus 1 gives 1 - 4
   }

   public static String getString(int suit) {
      switch(suit) {
         case Diamond:
            return "Diamond";
         case Heart:
            return "Heart";
         case Spade:
            return "Spade";
         case Club:
            return "Club";
         default:
            throw new IllegalArgumentException("Unknown suit identifier");
      }
   }

}
```

Rank.java

```
import java.util.*;

public class Rank {
   public static final int Ace   = 1;
   public static final int Two   = 2;
   public static final int Three = 3;
   public static final int Four  = 4;
   public static final int Five  = 5;
   public static final int Six   = 6;
   public static final int Seven = 7;
   public static final int Eight = 8;
   public static final int Nine  = 9;
   public static final int Ten   = 10;
   public static final int Jack  = 11;
   public static final int Queen = 12;
   public static final int King  = 13;

   public static int RandomRank() {
      Random rand = new Random();
      return Math.abs((rand.nextInt() % 13) + 1);
   }

}
```

PositionalList.java

```
public interface PositionalList<E> {
   public int size();
   public boolean isEmpty();
   Position<E> first();
   Position<E> last();
   Position<E> before(Position<E> p) throws IllegalArgumentException;
   Position<E> after(Position<E> p) throws IllegalArgumentException;
   Position<E> addFirst(E e);
   Position<E> addLast(E e);
   Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;
   Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;
   E set(Position<E> p, E e) throws IllegalArgumentException;
   E remove(Position<E> p) throws IllegalArgumentException;
}
```

Position.java

```
public interface Position<E> {
   public E getElement() throws IllegalStateException;
}
```

LinkedPositionalList.java

```
public class LinkedPositionalList<E> implements PositionalList<E> {

   private static class Node<E> implements Position<E> {

      private E element;
      private Node<E> prev;
      private Node<E> next;

      public Node(E e, Node<E> p, Node<E> n) {
         element = e;
         prev = p;
         next = n;
      }

      @Override
      public E getElement() throws IllegalStateException {
         if(next == null)
            throw new IllegalStateException("Position no longer valid");
         return element;
      }

      public Node<E> getPrev() {
         return prev;
      }

      public Node<E> getNext() {
         return next;
      }

      public void setElement(E e) {
         element = e;
      }

      public void setPrev(Node<E> p) {
         prev = p;
      }

      public void setNext(Node<E> n) {
         next = n;
      }
   }

   private Node<E> header, trailer;
   private int size = 0;

   public LinkedPositionalList() {
      header = new Node<>(null, null, null);
      trailer = new Node<>(null, header, null);
```

```
    header.setNext(trailer);
}

private Node<E> validate(Position<E> p) throws IllegalArgumentException {
    if(!(p instanceof Node))
        throw new IllegalArgumentException("Invalid p");
    Node<E> node = (Node<E>)p; // safe cast
    if(node.getNext() == null)
        throw new IllegalArgumentException("p is no longer in the list");
    return node;
}

private Position<E> position(Node<E> node) {
    if(node == header || node == trailer)
        return null;
    return node;
}

@Override
public int size() {
    return size;
}

@Override
public boolean isEmpty() {
    return (size == 0);
}

@Override
public Position<E> first() {
    return position(header.getNext());
}

@Override
public Position<E> last() {
    return position(trailer.getPrev());
}

@Override
public Position<E> before(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    return position(node.getPrev());
}

@Override
public Position<E> after(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    return position(node.getNext());
}
```

```
private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
   Node<E> newest  = new Node<>(e, pred, succ);
   pred.setNext(newest);
   succ.setPrev(newest);
   size++;
   return newest;
}

@Override
public Position<E> addFirst(E e) {
   return addBetween(e, header, header.getNext());
}

@Override
public Position<E> addLast(E e) {
   return addBetween(e, trailer.getPrev(), trailer);
}

@Override
public Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException {
   Node<E> node = validate(p);
   return addBetween(e, node.getPrev(), node);
}

@Override
public Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException {
   Node<E> node = validate(p);
   return addBetween(e, node, node.getNext());
}

@Override
public E set(Position<E> p, E e) throws IllegalArgumentException {
   Node<E> node = validate(p);
   E answer = node.getElement();
   node.setElement(e);
   return answer;
}

@Override
public E remove(Position<E> p) throws IllegalArgumentException {
   Node<E> node = validate(p);
   Node<E> predecessor = node.getPrev();
   Node<E> successor = node.getNext();

   predecessor.setNext(successor);
   successor.setPrev(predecessor);

   size--;
```

```
        E answer = node.getElement();

        node.setElement(null);
        node.setNext(null);
        node.setPrev(null);

        return answer;
    }

}
```