Most students have played card games: blackjack, war, hearts, solitaire, bridge. The list of games isn't infinite, but it's practically unbounded. In this design exposition, we'll discuss the design and implementation of a playing card class. We'll talk about issues in designing classes to represent both a deck of cards and piles of cards used in different games. The Web site that accompanies this design discussion includes references, code, and exercises for many assignments and in-class discussions. In this document we'll concentrate on the playing card classes and the deck class to keep things simple.

Students and teachers often wonder when it's appropriate to use a Java interface rather than a class. Some designers and educators think all object-oriented designs should start with interfaces. It is hard to motivate this stance with only a simple appeal to experts as a justification. In the design of a playing card class, our scenario begins with a teacher providing an initial specification and code to students and then asking them to write programs that play games with cards. Our goal is for one student's game or player to interact with another's. We'd also like to ensure that student-written code for a card player does not change the cards that are dealt. Using an interface provides a simple way for students to use cards in the code they write without having access to a card's internals, without being able to create a specific card, and without knowing how cards are implemented. This process begins with the code for a card interface, an interface we call ICard.

```java
public interface ICard extends Comparable {
    public static final int SPADES = 0;
    public static final int HEARTS = 1;
    public static final int DIAMONDS = 2;
    public static final int CLUBS = 3;

    public int getSuit();
    public int getRank();
}
```

The interface specifies the behavior of a card without providing information about how cards are implemented. Once they know that `getSuit()` returns a value like `ICard.HEARTS`, and that `getRank()` returns a value in the range of 1 (ace) to 13 (king), students can write code from this specification. For example, here's code to check whether an array of cards is sorted. We don’t know how it’s been sorted (e.g., do all the aces come before the twos or do all the spades come before the hearts?), but we can determine that an array is sorted.

---

1 Beginning interface names with an uppercase I, followed by a capitalized name, is a common naming convention in object-oriented programming in many languages, not just Java.
public boolean isSorted(ICard[] list)
    { for(int k=1; k < list.length; k++){
       if (list[k-1].compareTo(list[k]) > 0){
           return false;
       }
    }
    return true;
}

Starting with this simple ICard interface, we can ask students many kinds of questions to test and review concepts ranging from Java syntax to problem-solving with respect to one, two, or many cards. Some simple examples are included here, and more are available on the Web site. In answering these questions students must understand the interface since there is no implementation. Students focus on behavior rather than on instance variables and other implementation details, such as how to create a string to represent the ace of spades.

**ICard Study/Code Questions**

1. Write the function isRed that returns true if its ICard parameter is red (hearts or diamonds) and returns false otherwise.
   
   public boolean isRed(ICard card){…}

2. A pair is two cards of the same rank (e.g., two kings or two eights). Write the function isPair that returns true if its two ICard parameters represent a pair and returns false otherwise.
   
   public boolean isPair(ICard a, ICard b){…}

3. A flush is a hand, say in poker, in which all the cards have the same suit (e.g., five hearts, or five clubs for a five-card hand). Write the function isFlush that returns true if the array of cards is a flush and returns false otherwise.
   
   public boolean isFlush(ICard[] hand){…}

4. In blackjack or 21, the value of a hand is the total of the cards, where jacks, queens, and kings (11, 12, and 13, respectively, as returned by getRank()) each count as 10, and an ace counts as 1 or 10, whichever is better. A total over 21 is a bust; it’s not good to bust. Write function handTotal, which returns the total value of a hand.
   
   public int handTotal(ICard[] hand){…}

**From Interface to Implementation**

The ICard interface provides enough information to write code about cards, but there’s no way to create an array of cards, for example, or even a single card to test the functions written above (like isPair and handTotal). Where do cards come from? In most real-world examples, cards come from a Deck. We’ll
design a class that models a Deck—which is basically a factory for creating and obtaining cards.

To keep things simple, and to encourage the study of some standard Java interfaces, the class Deck will implement the java.util.Iterator interface. For example, to store all the cards from a deck into an ArrayList variable, we can use the following code:

```java
Deck d = new Deck();
ArrayList cards = new ArrayList();
while (d.hasNext()){
    ICard card = (ICard) d.next();
    System.out.println(card);
    cards.add(card);
}
System.out.println("# of cards dealt = " + cards.size());
```

The last few lines output by this code snippet might be as shown below. They will be different each time because the Deck class developed here shuffles the cards it deals via iteration.

```
... 
ace of spades
jack of clubs
six of spades
ten of hearts
ten of spades
# of cards dealt = 52
```

If we change the lines after the loop as follows, the output changes as well.

```java
Collections.sort(cards);
for (int k=0; k < cards.size(); k++){
    System.out.println(cards.get(k));
}
System.out.println("# of cards dealt = " + cards.size());
```

The output shows how cards returned from the Deck class implement the Comparable interface.

```
... 
nine of clubs
ten of clubs
jack of clubs
queen of clubs
king of clubs
# of cards dealt = 52
```
The complete code for the class `Deck` is shown below. The methods `hasNext()`, `next()`, and `remove()` are required for classes that implement the `Iterator` interface. The code below shows how objects of type `Card` are constructed.

```java
public class Deck implements Iterator{
    private ArrayList myCardList;
    private int myIndex;

    public Deck(){
        myCardList = new ArrayList();

        for(int suit = ICard.SPADES; suit <= ICard.CLUBS; suit++){
            for (int rank = 1; rank <= 13; rank++){
                myCardList.add(new Card(suit,rank));
            }
        }
        shuffle();
    }

    private void shuffle(){
        Collections.shuffle(myCardList);
        myIndex = 0;
    }

    public boolean hasNext() {
        return myIndex < myCardList.size();
    }

    public Object next() {
        ICard card = (ICard) myCardList.get(myIndex);
        myIndex++;
        return card;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

A `Deck` object stores 52 cards --- these cards can be obtained from a `Deck` object via iteration, but a `Deck` object cannot be reshuffled and re-used. Instead, a new `Deck` object must be created to deal new cards. This keeps things simple and provides an easy-to-follow example of a class that implements the `Iterator` interface. The method `remove()` is optional --- for the `Deck` class calling this method throws an exception.
Deck Study/Code Questions

1. Just before the shuffle method is called in the constructor, describe the order of the objects stored in myCardList.

2. Describe how each Deck method changes if the instance variable myCardList is changed to an array of Card objects, for example,

   ```java
   private ICard[] myCardList;
   ```

   Which choice for myCardList is better? Why?

3. Write client code that defines a Deck object and creates an array of 13 ICard objects that represent the spades that are dealt from the Deck. Do this by examining each object dealt and only storing the spade cards.

4. Write the body of the hypothetical Hand class constructor specified below

   ```java
   private ArrayList myCards;
   /**
    * deal numCards cards from d, store in myCards
    * (assume there are at least numCards cards left in d)
    */
   public Hand(Deck d, int numCards){
   }
   ```

From Decks to Cards

Our original concern was to use the ICard interface rather than worry about how cards are implemented. Nevertheless, at some point, there needs to be an implementation. It's not hard to argue that Card objects should be created by the Deck class. This is the approach we've used here. The Card class is a private class declared within the Deck class. There's actually no good reason to declare it within the Deck (the Deck.java file). However, by declaring it private, we make it impossible for any code class\(^2\); it could just as easily be declared as a non-public class within methods other than the Deck class to construct Card objects. This helps meet our original goal. Client programs can obtain cards from a Deck, but cannot create cards. Since the Deck supplies ICard objects, it's not possible to change a card once it's obtained from the Deck since the ICard interfaced doesn't support modification of a card.

As written, the private Card class defined within the Deck class doesn't support modification either since its private state instance variables are final, but this is extra protection that's likely not needed since no client code has access the private Card class.

\(^2\) Typically classes declared within another class often make reference to the enclosing object's state. In this case the nested class Card is declared as a private static class, so it can't reference private non-static state within a Deck object. The Card class could reference static Deck state, but there is none in this code.
The Card class is available on the Web site; we're not including it here since its implementation isn't directly related to our discussion about design.

A careful reader might claim that our original goal hasn't been met. Client code can cheat, for example, by creating a Deck object and then dealing cards from this object until an ace of spaces (or any other card) is dealt. In the current design of the Deck class this is true. However, we could create a singleton Deck object, in the same way that a single instance of the class Random is used in the Marine Biology Case Study. Singleton objects are typically created by declaring constructors so that they're private. In this case the Deck constructor would change from public to private. Client code obtains a Deck by calling a public getInstance() method, which returns a private static Deck object stored in the Deck class. The getInstance method creates this private object the first time getInstance is called. Details can be found in many texts or by studying the code from the Marine Biology Case Study.

More Code and Details

The images used in this document, and in the card games and supporting code available from the companion website, have been released under the GPL --- the Gnu Public License. They are available from the creator of the images at http://www.waste.org/~oxymoron/cards/ and from many other sites (including the companion site for this material). The code supporting this design document is released under a Creative Commons License, as described at http://www.cs.duke.edu/csed/ap/cards, where the code and more material are available.