```
                }
            } while (f*f <= n);
            set(n);
            return n;
        }
    }
```

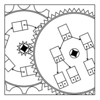
Clearly, the `reset` method is responsible for restarting the sequence at 2. While it would be possible for each `Generator` to keep track of its current value in its own manner, placing that general code in the `AbstractGenerator` reduces the overall cost of keeping track of this information for each of the many implementations.

Exercise 7.1 *Implement a* `Generator` *that provides a stream of random integers. After a call to* `reset`*, the random sequence is "rewound" to return the same sequence again. Different generators should probably generate different sequences. (Hint: You may find it useful to use the* `setSeed` *method of* `java.util.Random`*.)*

7.3 Example: Playing Cards

Many games involve the use of playing cards. Not all decks of cards are the same. For example, a deck of bridge cards has four suits with thirteen cards in each suit. There are no jokers. Poker has a similar deck of cards, but various games include special joker or wild cards. A pinochle deck has 48 cards consisting of two copies of 9, jack, queen, king, 10, and ace in each of four suits. The ranking of cards places 10 between king and ace. A baccarat deck is as in bridge, except that face cards are worth nothing. Cribbage uses a standard deck of cards, with aces low.

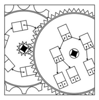
While there are many differences among these card decks, there are some common features. In each, cards have a suit and a face (e.g., ace, king, 5). Most decks assign a value or rank to each of the cards. In many games it is desirable to be able to compare the relative values of two cards. With these features in mind, we can develop the following `Card` interface:


Card

```
    public interface Card
    {
        public static final int ACE = 1;
        public static final int JACK = 11;
        public static final int QUEEN = 12;
        public static final int KING = 13;
        public static final int JOKER = 14;
        public static final int CLUBS = 0;
        public static final int DIAMONDS = 1;
        public static final int HEARTS = 2;
        public static final int SPADES = 3;
        public int suit();
        // post: returns the suit of the card

        public int face();
        // post: returns the face of the card, e.g., ACE, 3, JACK

        public boolean isWild();
        // post: returns true iff this card is a wild card

        public int value();
        // post: return the point value of the card

        public int compareTo(Object other);
        // pre: other is valid Card
        // post: returns int <,==,> 0 if this card is <,==,> other

        public String toString();
        // post: returns a printable version of this card
    }
```

The card interface provides all the public methods that we need to have in our card games, but it does not provide any hints at how the cards are implemented. The interface also provides standard names for faces and suits that are passed to and returned from the various card methods.

In the expectation that most card implementations are similar to a standard deck of cards, we provide an AbstractCard class that keeps track of an integer—a card index—that may be changed with set or retrieved with get (both are protected methods):



AbstractCard

```
    import java.util.Random;
    public abstract class AbstractCard implements Card
    {
        protected int cardIndex;
        protected static Random gen = new Random();

        public AbstractCard()
        // post: constructs a random card in a standard deck
        {
            set(randomIndex(52));
        }

        protected static int randomIndex(int max)
        // pre: max > 0
        // post: returns a random number n, 0 <= n < max
        {
            return Math.abs(gen.nextInt()) % max;
        }

        protected void set(int index)
        // post: this card has cardIndex index
        {
```

```
        cardIndex = index;
    }

    protected int get()
    // post: returns this card’s card index
    {
        return cardIndex;
    }

    public int suit()
    // post: returns the suit of the card
    {
        return cardIndex / 13;
    }

    public int face()
    // post: returns the face of the card, e.g. ACE, 3, JACK
    {
        return (cardIndex % 13)+1;
    }

    public boolean isWild()
    // post: returns true iff this card is a wild card
    // (default is false)
    {
        return false;
    }

    public int value()
    // post: return the point value of the card, Ace..King
    {
        return face();
    }

    public String toString()
    // post: returns a printable version of this card
    {
        String cardName = "";
        switch (face())
        {
          case ACE: cardName = "Ace"; break;
          case JACK: cardName = "Jack"; break;
          case QUEEN: cardName = "Queen"; break;
          case KING: cardName = "King"; break;
          default: cardName = cardName + face(); break;
        }
        switch (suit())
        {
          case HEARTS: cardName += " of Hearts"; break;
          case DIAMONDS: cardName += " of Diamonds"; break;
```

```
            case CLUBS: cardName += " of Clubs"; break;
            case SPADES: cardName += " of Spades"; break;
          }
          return cardName;
      }
  }
```

Our abstract base class also provides a protected random number generator that returns values from 0 to `max-1`. We make use of this in the default constructor for a standard deck; it picks a random card from the usual 52. The cards are indexed from the ace of clubs through the king of spades. Thus, the `face` and `suit` methods must use division and modulo operators to split the card index into the two constituent parts. By default, the `value` method returns the face of the card as its value. This is likely to be different for different implementations of cards, as the face values of cards in different games vary considerably.

We also provide a standard `toString` method that allows us to easily print out a card. We do not provide a `compareTo` method because there are complexities with comparing cards that cannot be predicted at this stage. For example, in bridge, suits play a predominant role in comparing cards. In baccarat they do not.

Since a poker deck is very similar to our standard implementation, we find the `PokerCard` implementation is very short. All that is important is that we allow aces to have high values:

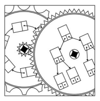
PokerCard

```
public class PokerCard extends AbstractCard
{
    public PokerCard(int face, int suit)
    // pre: face and suit have valid values
    // post: constructs a card with the particular face value
    {
        set(suit*13+face-1);
    }

    public PokerCard()
    // post: construct a random poker card.
    {
        // by default, calls the AbstractCard constructor
    }

    public int value()
    // post: returns rank of card - aces are high
    {
        if (face() == ACE) return KING+1;
        else return face();
    }

    public int compareTo(Object other)
    // pre: other is valid PokerCard
    // post: returns relationship between this card and other
```

```
        {
            PokerCard that = (PokerCard)other;
            return value()-that.value();
        }
    }
```

Exercise 7.2 *Write the* `value` *and* `compareTo` *methods for a pair of cards where suits play an important role. Aces are high, and assume that suits are ranked clubs (low), diamonds, hearts, and spades (high). Assume that face values are only considered if the suits are the same; otherwise ranking of cards depends on their suits alone.*

The implementation of a pinochle card is particularly difficult. We are interested in providing the standard interface for a pinochle card, but we are faced with the fact that there are two copies each of the six cards 9, jack, queen, king, 10, and ace, in each of the four suits. Furthermore we assume that 10 has the unusual ranking between king and ace. Here's one approach:


PinochleCard

```
    public class PinochleCard extends AbstractCard
    {
        // cardIndex        face    suit
        // 0                9       clubs
        // 1                9       clubs (duplicate)
        // ...
        // 10               ACE     clubs
        // 11               ACE     clubs (duplicate)
        // 12               9       diamonds
        // 13               9       diamonds (duplicate)
        // ...
        // 47               ACE     spades (duplicate)

        public PinochleCard(int face, int suit, int copy)
        // pre: face and suit have valid values
        // post: constructs a card with the particular face value
        {
            if (face == ACE) face = KING+1;
            set((suit*2+copy)*6+face-9);
        }

        public PinochleCard()
        // post: construct a random Pinochle card.
        {
            set(randomIndex(48));
        }

        public int face()
        // post: returns the face value of the card (9 thru Ace)
        {
            int result = get()%6 + 9;
            if (result == 14) result = ACE;
            return result;
        }

        public int suit()
        // post: returns the suit of the card (there are duplicates!)
        {
            // this is tricky; we divide by 12 cards (including duplicates)
            // per suit, and again by 2 to remove the duplicate
            return cardIndex / 12 / 2;
        }

        public int value()
        // post: returns rank of card - aces are high
        {
            if (face() == ACE) return KING+2;
            else if (face() == 10) return KING+1;
            else return face();
        }

        public int compareTo(Object other)
        // pre: other is valid PinochleCard
        // post: returns relationship between this card and other
        {
            PinochleCard that = (PinochleCard)other;
            return value()-that.value();
        }
    }
```

The difficulty is that there is more than one copy of a card. We choose to keep track of the extra copy, in case we need to distinguish between them at some point, but we treat duplicates the same in determining face value, suit, and relative rankings.

7.4 Conclusions

Throughout the remainder of this book we will find it useful to approach each type of data structure first in an abstract manner, and then provide the details of various implementations. While each implementation tends to have a distinct approach to supporting the abstract structure, many features are common to *all* implementations. The basic interface, for example, is a shared concept of the methods that are used to access the data structure. Other features—including common private methods and shared utility methods—are provided in a basic implementation called the *abstract base class*. This incomplete class serves as a single point of extension for many implementations; the public and private features of the abstract base class are shared (and possibly overridden) by the varied approaches to solving the problem.