Chapter 14

Objects of Arrays

WARNING: In this chapter, we take another step toward object-oriented programming, but we are not there yet. So many of the examples are non-idiomatic; that is, they are not good Java. This transitional form will help you learn (I hope), but don't write code like this.

You can download the code in this chapter from http://thinkapjava.com/code/Card2.java.

14.1 The Deck class

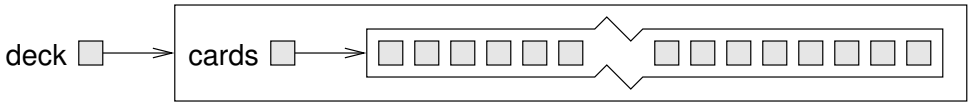
In the previous chapter, we worked with an array of objects, but I also mentioned that it is possible to have an object that contains an array as an instance variable. In this chapter we create a `Deck` object that contains an array of `Card`s.

The class definition looks like this:

```
class Deck {
    Card[] cards;

    public Deck(int n) {
        this.cards = new Card[n];
    }
}
```

The constructor initializes the instance variable with an array of cards, but it doesn't create any cards. Here is a state diagram showing what a `Deck` looks like with no cards:



Here is a no-argument constructor that makes a 52-card deck and populates it with `Card`s:

```
public Deck() {
    this.cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            cards[index] = new Card(suit, rank);
            index++;
        }
    }
}
```

This method is similar to `makeDeck`; we just changed the syntax to make it a constructor. To invoke it, we use `new`:

```
Deck deck = new Deck();
```

Now it makes sense to put the methods that pertain to `Deck`s in the `Deck` class definition. Looking at the methods we have written so far, one obvious candidate is `printDeck` (Section 13.7). Here's how it looks, rewritten to work with a `Deck`:

```
public static void printDeck(Deck deck) {
    for (int i = 0; i < deck.cards.length; i++) {
        Card.printCard(deck.cards[i]);
    }
}
```

One change is the type of the parameter, from `Card[]` to `Deck`.

The second change is that we can no longer use `deck.length` to get the length of the array, because `deck` is a `Deck` object now, not an array. It contains an array, but it is not an array. So we have to write `deck.cards.length` to extract the array from the `Deck` object and get the length of the array.

For the same reason, we have to use `deck.cards[i]` to access an element of the array, rather than just `deck[i]`.

The last change is that the invocation of `printCard` has to say explicitly that `printCard` is defined in the `Card` class.

14.2 Shuffling

For most card games you need to be able to shuffle the deck; that is, put the cards in a random order. In Section 12.6 we saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle, which is usually by dividing the deck in two and then choosing alternately from each deck. Since humans usually don't shuffle perfectly, after about 7 iterations the order of the deck is pretty well randomized. But a computer program would have the annoying property of doing a perfect shuffle every time, which is not really very random. In fact, after 8 perfect shuffles, you would find the deck back in the order you started in. For more information, see http://en.wikipedia.org/wiki/Faro_shuffle.

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration choose two cards and swap them.

Here is an outline of how this algorithm works. To sketch the program, I am using a combination of Java statements and English words that is sometimes called **pseudocode**:

```
for (int i = 0; i < deck.cards.length; i++) {
    // choose a number between i and deck.cards.length-1
    // swap the ith card and the randomly-chosen card
}
```

The nice thing about pseudocode is that it often makes it clear what methods you are going to need. In this case, we need something like `randomInt`, which chooses a random integer between `low` and `high`, and `swapCards` which takes two indices and switches the cards at the indicated positions.

This process—writing pseudocode first and then writing methods to make it work—is called **top-down development** (see http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design).

14.3 Sorting

Now that we have messed up the deck, we need a way to put it back in order. There is an algorithm for sorting that is ironically similar to the algorithm for shuffling. It's called **selection sort** because it works by traversing the array repeatedly and selecting the lowest remaining card each time.

During the first iteration we find the lowest card and swap it with the card in the 0th position. During the `i`th, we find the lowest card to the right of `i` and swap it with the `i`th card.

Here is pseudocode for selection sort:

```
for (int i = 0; i < deck.cards.length; i++) {
    // find the lowest card at or to the right of i
    // swap the ith card and the lowest card
}
```

Again, the pseudocode helps with the design of the **helper methods**. In this case we can use `swapCards` again, so we only need one new one, called `indexLowestCard`, that takes an array of cards and an index where it should start looking.

14.4 Subdecks

How should we represent a hand or some other subset of a full deck? One possibility is to create a new class called `Hand`, which might extend `Deck`. Another possibility, the one I will demonstrate, is to represent a hand with a `Deck` object with fewer than 52 cards.

We might want a method, `subdeck`, that takes a Deck and a range of indices, and that returns a new Deck that contains the specified subset of the cards:

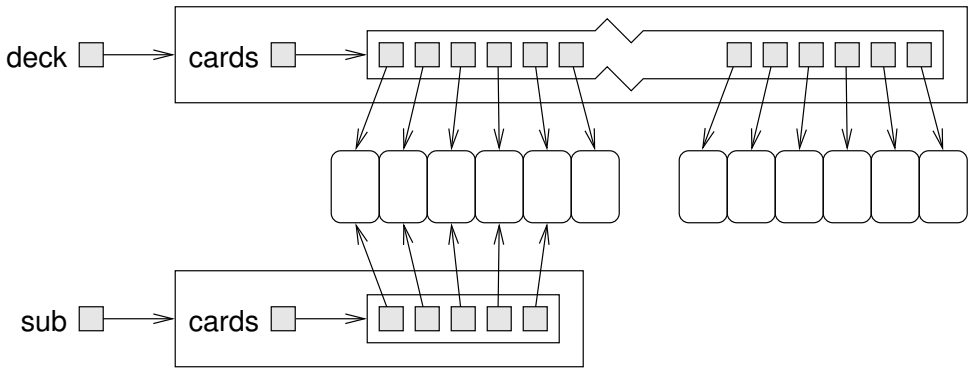
```
public static Deck subdeck(Deck deck, int low, int high) {
    Deck sub = new Deck(high-low+1);

    for (int i = 0; i<sub.cards.length; i++) {
        sub.cards[i] = deck.cards[low+i];
    }
    return sub;
}
```

The length of the subdeck is `high-low+1` because both the low card and high card are included. This sort of computation can be confusing, and lead to "off-by-one" errors. Drawing a picture is usually the best way to avoid them.

Because we provide an argument with `new`, the contructor that gets invoked will be the first one, which only allocates the array and doesn't allocate any cards. Inside the `for` loop, the subdeck gets populated with copies of the references from the deck.

The following is a state diagram of a subdeck being created with the parameters `low=3` and `high=7`. The result is a hand with 5 cards that are shared with the original deck; i.e. they are aliased.



Aliasing is usually not generally a good idea, because changes in one subdeck are reflected in others, which is not the behavior you would expect from real cards and decks. But if the cards are immutable, aliasing is less dangerous. In this case, there is probably no reason ever to change the rank or suit of a card. Instead we can create each card once and then treat it as an immutable object. So for `Card`s aliasing is a reasonable choice.

14.5 Shuffling and dealing

In Section 14.2 I wrote pseudocode for a shuffling algorithm. Assuming that we have a method called `shuffleDeck` that takes a deck as an argument and shuffles it, we can use it to deal hands:

```
Deck deck = new Deck();
shuffleDeck(deck);
```

```
Deck hand1 = subdeck(deck, 0, 4);
Deck hand2 = subdeck(deck, 5, 9);
Deck pack = subdeck(deck, 10, 51);
```

This code puts the first 5 cards in one hand, the next 5 cards in the other, and the rest into the pack.

When you thought about dealing, did you think we should give one card to each player in the round-robin style that is common in real card games? I thought about it, but then realized that it is unnecessary for a computer program. The round-robin convention is intended to mitigate imperfect shuffling and make it more difficult for the dealer to cheat. Neither of these is an issue for a computer.

This example is a useful reminder of one of the dangers of engineering metaphors: sometimes we impose restrictions on computers that are unnecessary, or expect capabilities that are lacking, because we unthinkingly extend a metaphor past its breaking point.

14.6 Mergesort

In Section 14.3, we saw a simple sorting algorithm that turns out not to be very efficient. To sort n items, it has to traverse the array n times, and each traversal takes an amount of time that is proportional to n. The total time, therefore, is proportional to n^2.

In this section I sketch a more efficient algorithm called **mergesort**. To sort n items, mergesort takes time proportional to $n \log n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log n$ can be enormous. Try out a few values of n and see.

The basic idea behind mergesort is this: if you have two subdecks, each of which has been sorted, it is easy (and fast) to merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two subdecks with about 10 cards each and sort them so that when they are face up the lowest cards are on top. Place both decks face up in front of you.

2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.

3. Repeat step two until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. Here's what this looks like in pseudocode:

```
public static Deck merge(Deck d1, Deck d2) {
    // create a new deck big enough for all the cards
    Deck result = new Deck(d1.cards.length + d2.cards.length);

    // use the index i to keep track of where we are in
    // the first deck, and the index j for the second deck
    int i = 0;
    int j = 0;

    // the index k traverses the result deck
    for (int k = 0; k < result.cards.length; k++) {

        // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
        // otherwise, compare the two cards

        // add the winner to the new deck
    }
    return result;
}
```

The best way to test `merge` is to build and shuffle a deck, use subdeck to form two (small) hands, and then use the sort routine from the previous chapter to sort the two halves. Then you can pass the two halves to `merge` to see if it works.

If you can get that working, try a simple implementation of `mergeSort`:

```
public static Deck mergeSort(Deck deck) {
    // find the midpoint of the deck
    // divide the deck into two subdecks
    // sort the subdecks using sortDeck
```

```
    // merge the two halves and return the result
}
```

Then, if you get that working, the real fun begins! The magical thing about mergesort is that it is recursive. At the point where you sort the subdecks, why should you invoke the old, slow version of `sort`? Why not invoke the spiffy new `mergeSort` you are in the process of writing?

Not only is that a good idea, it is *necessary* to achieve the performance advantage I promised. But to make it work you have to have a base case; otherwise it recurses forever. A simple base case is a subdeck with 0 or 1 cards. If `mergesort` receives such a small subdeck, it can return it unmodified, since it is already sorted.

The recursive version of `mergesort` should look something like this:

```
public static Deck mergeSort(Deck deck) {
    // if the deck is 0 or 1 cards, return it

    // find the midpoint of the deck
    // divide the deck into two subdecks
    // sort the subdecks using mergesort
    // merge the two halves and return the result
}
```

As usual, there are two ways to think about recursive programs: you can think through the entire flow of execution, or you can make the "leap of faith" (see Section 6.9). I have constructed this example to encourage you to make the leap of faith.

When you use `sortDeck` to sort the subdecks, you don't feel compelled to follow the flow of execution, right? You just assume it works because you already debugged it. Well, all you did to make `mergeSort` recursive was replace one sorting algorithm with another. There is no reason to read the program differently.

Actually, you have to give some thought to getting the base case right and making sure that you reach it eventually, but other than that, writing the recursive version should be no problem. Good luck!

14.7 Class variables

So far we have seen local variables, which are declared inside a method, and instance variables, which are declared in a class definition, usually before the method definitions.

Local variables are created when a method is invoked and destroyed when the method ends. Instance variables are created when you create an object and destroyed when the object is garbage collected.

Now it's time to learn about **class variables**. Like instance variables, class variables are defined in a class definition before the method definitions, but they are identified by the keyword `static`. They are created when the program starts and survive until the program ends.

You can refer to a class variable from anywhere inside the class definition. Class variables are often used to store constant values that are needed in several places.

As an example, here is a version of `Card` where `suits` and `ranks` are class variables:

```
class Card {
    int suit, rank;

    static String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    static String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                      "7", "8", "9", "10", "Jack", "Queen", "King" };

    public static void printCard(Card c) {
        System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
    }
}
```

Inside `printCard` we can refer to `suits` and `ranks` as if they were local variables.

14.8 Glossary

pseudocode: A way of designing programs by writing rough drafts in a combination of English and Java.

helper method: Often a small method that does not do anything enormously useful by itself, but which helps another, more useful method.

class variable: A variable declared within a class as `static`; there is always exactly one copy of this variable in existence.

14.9 Exercises

Exercise 14.1. The goal of this exercise is to implement the shuffling and sorting algorithms from this chapter.

1. Download the code from this chapter from http://thinkapjava.com/code/Card2.java and import it into your development environment. I have provided outlines for the methods you will write, so the program should compile. But when it runs it prints messages indicating that the empty methods are not working. When you fill them in correctly, the messages should go away.

2. If you did Exercise 12.3, you already wrote `randomInt`. Otherwise, write it now and add code to test it.

3. Write a method called `swapCards` that takes a deck (array of cards) and two indices, and that switches the cards at those two locations.

   HINT: it should switch references, not the contents of the objects. This is faster; also, it correctly handles the case where cards are aliased.

4. Write a method called `shuffleDeck` that uses the algorithm in Section 14.2. You might want to use the `randomInt` method from Exercise 12.3.

5. Write a method called `indexLowestCard` that uses the `compareCard` method to find the lowest card in a given range of the deck (from `lowIndex` to `highIndex`, including both).

6. Write a method called `sortDeck` that arranges a deck of cards from lowest to highest.

7. Using the pseudocode in Section 14.6, write the method called `merge`. Be sure to test it before trying to use it as part of a `mergeSort`.

8. Write the simple version of `mergeSort`, the one that divides the deck in half, uses `sortDeck` to sort the two halves, and uses `merge` to create a new, fully-sorted deck.

9. Write the fully recursive version of `mergeSort`. Remember that `sortDeck` is a modifier and `mergeSort` is a function, which means that they get invoked differently:

```
sortDeck(deck);              // modifies existing deck
deck = mergeSort(deck);      // replaces old deck with new
```