

# Chapter 11

## Arrays of Objects

### 11.1 Composition

By now we have seen several examples of composition (the ability to combine language features in a variety of arrangements). One of the first examples we saw was using a method invocation as part of an expression. Another example is the nested structure of statements: you can put an `if` statement within a `while` loop, or within another `if` statement, etc.

Having seen this pattern, and having learned about arrays and objects, you should not be surprised to learn that you can have arrays of objects. In fact, you can also have objects that contain arrays (as instance variables); you can have arrays that contain arrays; you can have objects that contain objects, and so on.

In the next two chapters we will look at some examples of these combinations, using `Card` objects as an example.

### 11.2 Card objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are 52 cards in a deck, each of which belongs to one of four suits and one of 13 ranks. The suits are Spades, Hearts, Diamonds and Clubs (in descending order in Bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King. Depending on what game you are playing, the rank of the Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is pretty obvious what the instance variables should be: `rank` and `suit`. It is not as obvious what type the instance variables should be. One possibility is `Strings`, containing things like `"Spade"` for suits and `"Queen"` for ranks. One problem with this

implementation is that it would not be easy to compare cards to see which had higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By “encode,” I do not mean what some people think, which is to encrypt, or translate into a secret code. What a computer scientist means by “encode” is something like “define a mapping between a sequence of numbers and the things I want to represent.” For example,

Spades	↦	3
Hearts	↦	2
Diamonds	↦	1
Clubs	↦	0

The obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	↦	11
Queen	↦	12
King	↦	13

The reason I am using mathematical notation for these mappings is that they are not part of the Java program. They are part of the program design, but they never appear explicitly in the code. The class definition for the **Card** type looks like this:

```
class Card
{
    int suit, rank;

    public Card () {
        this.suit = 0;  this.rank = 0;
    }

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

As usual, I am providing two constructors, one of which takes a parameter for each instance variable and the other of which takes no parameters.

To create an object that represents the 3 of Clubs, we would use the **new** command:

```
Card threeOfClubs = new Card (0, 3);
```

The first argument, 0 represents the suit Clubs.

## 11.3 The printCard method

When you create a new class, the first step is usually to declare the instance variables and write constructors. The second step is often to write the standard methods that every object should have, including one that prints the object, and one or two that compare objects. I will start with `printCard`.

In order to print `Card` objects in a way that humans can read easily, we want to map the integer codes onto words. A natural way to do that is with an array of `Strings`. You can create an array of `Strings` the same way you create an array of primitive types:

```
String[] suits = new String [4];
```

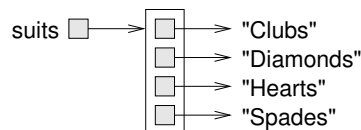
Then we can set the values of the elements of the array.

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

Creating an array and initializing the elements is such a common operation that Java provides a special syntax for it:

```
String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

The effect of this statement is identical to that of the separate declaration, allocation, and assignment. A state diagram of this array might look like:



The elements of the array are *references* to the `Strings`, rather than `Strings` themselves. This is true of all arrays of objects, as I will discuss in more detail later. For now, all we need is another array of `Strings` to decode the ranks:

```
String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                  "7", "8", "9", "10", "Jack", "Queen", "King" };
```

The reason for the "narf" is to act as a place-keeper for the zeroeth element of the array, which will never be used. The only valid ranks are 1–13. This wasted element is not necessary, of course. We could have started at 0, as usual, but it is best to encode 2 as 2, and 3 as 3, etc.

Using these arrays, we can select the appropriate `Strings` by using the `suit` and `rank` as indices. In the method `printCard`,

```
public static void printCard (Card c) {
    String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                      "7", "8", "9", "10", "Jack", "Queen", "King" };

    System.out.println (ranks[c.rank] + " of " + suits[c.suit]);
}
```

the expression `suits[c.suit]` means “use the instance variable `suit` from the object `c` as an index into the array named `suits`, and select the appropriate string.” The output of this code

```
Card card = new Card (1, 11);
printCard (card);
```

is Jack of Diamonds.

## 11.4 The `sameCard` method

The word “same” is one of those things that occur in natural language that seem perfectly clear until you give it some thought, and then you realize there is more to it than you expected.

For example, if I say “Chris and I have the same car,” I mean that his car and mine are the same make and model, but they are two different cars. If I say “Chris and I have the same mother,” I mean that his mother and mine are one and the same. So the idea of “sameness” is different depending on the context.

When you talk about objects, there is a similar ambiguity. For example, if two `Cards` are the same, does that mean they contain the same data (rank and suit), or they are actually the same `Card` object?

To see if two references refer to the same object, we can use the `==` operator. For example:

```
Card card1 = new Card (1, 11);
Card card2 = card1;

if (card1 == card2) {
    System.out.println ("card1 and card2 are the same object.");
}
```

This type of equality is called **shallow equality** because it only compares the references, not the contents of the objects.

To compare the contents of the objects—**deep equality**—it is common to write a method with a name like `sameCard`.

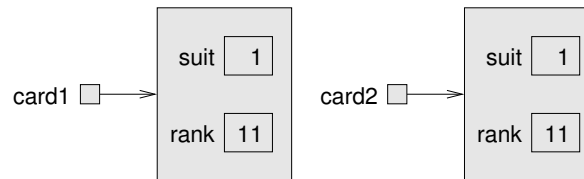
```
public static boolean sameCard (Card c1, Card c2) {
    return (c1.suit == c2.suit && c1.rank == c2.rank);
}
```

Now if we create two different objects that contain the same data, we can use `sameCard` to see if they represent the same card:

```
Card card1 = new Card (1, 11);
Card card2 = new Card (1, 11);

if (sameCard (card1, card2)) {
    System.out.println ("card1 and card2 are the same card.");
}
```

In this case, `card1` and `card2` are two different objects that contain the same data



so the condition is true. What does the state diagram look like when `card1 == card2` is true?

In Section 7.10 I said that you should never use the `==` operator on `Strings` because it does not do what you expect. Instead of comparing the contents of the `String` (deep equality), it checks whether the two `Strings` are the same object (shallow equality).

## 11.5 The compareCard method

For primitive types, there are conditional operators that compare values and determine when one is greater or less than another. These operators (`<` and `>` and the others) don't work for object types. For `Strings` there is a built-in `compareTo` method. For `Cards` we have to write our own, which we will call `compareCard`. Later, we will use this method to sort a deck of cards.

Some sets are completely ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are totally ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why we cannot compare apples and oranges. In Java, the `boolean` type is unordered; we cannot say that `true` is greater than `false`.

The set of playing cards is partially ordered, which means that sometimes we can compare cards and sometimes not. For example, I know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, we have to decide which is more important, rank or suit. To be honest, the choice is completely arbitrary. For the sake of choosing, I will say that suit is more important, because when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `compareCard`. It will take two `Cards` as parameters and return 1 if the first card wins, -1 if the second card wins, and 0 if they tie (indicating deep equality). It is sometimes confusing to keep those return values straight, but they are pretty standard for comparison methods.

First we compare the suits:

```
if (c1.suit > c2.suit) return 1;
if (c1.suit < c2.suit) return -1;
```

If neither statement is true, then the suits must be equal, and we have to compare ranks:

```
if (c1.rank > c2.rank) return 1;
if (c1.rank < c2.rank) return -1;
```

If neither of these is true, the ranks must be equal, so we return 0. In this ordering, aces will appear lower than deuces (2s).

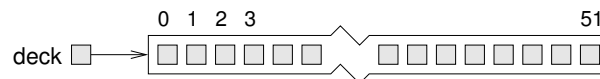
As an exercise, fix it so that aces are ranked higher than Kings, and encapsulate this code in a method.

## 11.6 Arrays of cards

The reason I chose `Card`s as the objects for this chapter is that there is an obvious use for an array of cards—a deck. Here is some code that creates a new deck of 52 cards:

```
Card[] deck = new Card [52];
```

Here is the state diagram for this object:



The important thing to see here is that the array contains only *references* to objects; it does not contain any `Card` objects. The values of the array elements are initialized to `null`. You can access the elements of the array in the usual way:

```
if (deck[3] == null) {
    System.out.println ("No cards yet!");
}
```

But if you try to access the instance variables of the non-existent `Cards`, you will get a `NullPointerException`.

```
deck[2].rank; // NullPointerException
```

Nevertheless, that is the correct syntax for accessing the `rank` of the “twoeth” card in the deck (really the third—we started at zero, remember?). This is another example of composition, the combination of the syntax for accessing an element of an array and an instance variable of an object.

The easiest way to populate the deck with `Card` objects is to write a nested loop:

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
```

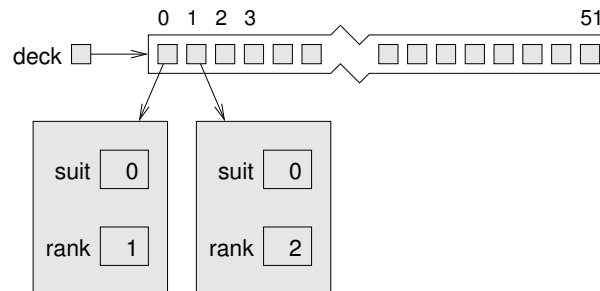
```

        deck[index] = new Card (suit, rank);
        index++;
    }
}

```

The outer loop enumerates the suits, from 0 to 3. For each suit, the inner loop enumerates the ranks, from 1 to 13. Since the outer loop iterates 4 times, and the inner loop iterates 13 times, the total number of times the body is executed is 52 (13 times 4).

I used the variable `index` to keep track of where in the deck the next card should go. The following state diagram shows what the deck looks like after the first two cards have been allocated:



**Exercise 11.1** Encapsulate this deck-building code in a method called `buildDeck` that takes no parameters and that returns a fully-populated array of `Cards`.

## 11.7 The printDeck method

Whenever you are working with arrays, it is convenient to have a method that will print the contents of the array. We have seen the pattern for traversing an array several times, so the following method should be familiar:

```

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

```

Since `deck` has type `Card[]`, an element of `deck` has type `Card`. Therefore, `deck[i]` is a legal argument for `printCard`.

## 11.8 Searching

The next method I want to write is `findCard`, which searches through an array of `Cards` to see whether it contains a certain card. It may not be obvious why this method would be useful, but it gives me a chance to demonstrate two ways to go searching for things, a **linear** search and a **bisection** search.

Linear search is the more obvious of the two; it involves traversing the deck and comparing each card to the one we are looking for. If we find it we return the index where the card appears. If it is not in the deck, we return -1.

```
public static int findCard (Card[] deck, Card card) {  
    for (int i = 0; i < deck.length; i++) {  
        if (sameCard (deck[i], card)) return i;  
    }  
    return -1;  
}
```

The arguments of `findCard` are named `card` and `deck`. It might seem odd to have a variable with the same name as a type (the `card` variable has type `Card`). This is legal and common, although it can sometimes make code hard to read. In this case, though, I think it works.

The method returns as soon as it discovers the card, which means that we do not have to traverse the entire deck if we find the card we are looking for. If the loop terminates without finding the card, we know the card is not in the deck and return -1.

If the cards in the deck are not in order, there is no way to search that is faster than this. We have to look at every card, since otherwise there is no way to be certain the card we want is not there.

But when you look for a word in a dictionary, you don't search linearly through every word. The reason is that the words are in alphabetical order. As a result, you probably use an algorithm that is similar to a bisection search:

1. Start in the middle somewhere.
2. Choose a word on the page and compare it to the word you are looking for.
3. If you found the word you are looking for, stop.
4. If the word you are looking for comes after the word on the page, flip to somewhere later in the dictionary and go to step 2.
5. If the word you are looking for comes before the word on the page, flip to somewhere earlier in the dictionary and go to step 2.

If you ever get to the point where there are two adjacent words on the page and your word comes between them, you can conclude that your word is not in the dictionary. The only alternative is that your word has been misfiled somewhere, but that contradicts our assumption that the words are in alphabetical order.

In the case of a deck of cards, if we know that the cards are in order, we can write a version of `findCard` that is much faster. The best way to write a bisection search is with a recursive method. That's because bisection is naturally recursive.



The trick is to write a method called `findBisect` that takes two indices as parameters, `low` and `high`, indicating the segment of the array that should be searched (including both `low` and `high`).

1. To search the array, choose an index between `low` and `high` (call it `mid`) and compare it to the card you are looking for.
2. If you found it, stop.
3. If the card at `mid` is higher than your card, search in the range from `low` to `mid-1`.
4. If the card at `mid` is lower than your card, search in the range from `mid+1` to `high`.

Steps 3 and 4 look suspiciously like recursive invocations. Here's what this all looks like translated into Java code:

```
public static int findBisect (Card[] deck, Card card, int low, int high) {
    int mid = (high + low) / 2;
    int comp = compareCard (deck[mid], card);

    if (comp == 0) {
        return mid;
    } else if (comp > 0) {
        return findBisect (deck, card, low, mid-1);
    } else {
        return findBisect (deck, card, mid+1, high);
    }
}
```

Rather than call `compareCard` three times, I called it once and stored the result.

Although this code contains the kernel of a bisection search, it is still missing a piece. As it is currently written, if the card is not in the deck, it will recurse forever. We need a way to detect this condition and deal with it properly (by returning `-1`).

The easiest way to tell that your card is not in the deck is if there are *no* cards in the deck, which is the case if `high` is less than `low`. Well, there are still cards in the deck, of course, but what I mean is that there are no cards in the segment of the deck indicated by `low` and `high`.

With that line added, the method works correctly:

```
public static int findBisect (Card[] deck, Card card, int low, int high) {
    System.out.println (low + ", " + high);

    if (high < low) return -1;

    int mid = (high + low) / 2;
    int comp = deck[mid].compareCard (card);
```

```
        if (comp == 0) {
            return mid;
        } else if (comp > 0) {
            return findBisect (deck, card, low, mid-1);
        } else {
            return findBisect (deck, card, mid+1, high);
        }
    }
}
```

I added a print statement at the beginning so I could watch the sequence of recursive calls and convince myself that it would eventually reach the base case. I tried out the following code:

```
Card card1 = new Card (1, 11);
System.out.println (findBisect (deck, card1, 0, 51));
```

And got the following output:

```
0, 51
0, 24
13, 24
19, 24
22, 24
23
```

Then I made up a card that is not in the deck (the 15 of Diamonds), and tried to find it. I got the following:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1
```

These tests don't prove that this program is correct. In fact, no amount of testing can prove that a program is correct. On the other hand, by looking at a few cases and examining the code, you might be able to convince yourself.

The number of recursive calls is fairly small, typically 6 or 7. That means we only had to invoke `compareCard` 6 or 7 times, compared to up to 52 times if we did a linear search. In general, bisection is much faster than a linear search, and even more so for large arrays.

Two common errors in recursive programs are forgetting to include a base case and writing the recursive call so that the base case is never reached. Either error will cause an infinite recursion, in which case Java will (eventually) throw a `StackOverflowException`.

## 11.9 Decks and subdecks

Looking at the interface to `findBisect`

```
public static int findBisect (Card[] deck, Card card, int low, int high)
```

it might make sense to think of three of the parameters, `deck`, `low` and `high`, as a single parameter that specifies a **subdeck**. This way of thinking is quite common, and I sometimes think of it as an **abstract parameter**. What I mean by “abstract,” is something that is not literally part of the program text, but which describes the function of the program at a higher level.

For example, when you invoke a method and pass an array and the bounds `low` and `high`, there is nothing that prevents the invoked method from accessing parts of the array that are out of bounds. So you are not literally sending a subset of the deck; you are really sending the whole deck. But as long as the recipient plays by the rules, it makes sense to think of it, abstractly, as a subdeck.

There is one other example of this kind of abstraction that you might have noticed in Section 9.7, when I referred to an “empty” data structure. The reason I put “empty” in quotation marks was to suggest that it is not literally accurate. All variables have values all the time. When you create them, they are given default values. So there is no such thing as an empty object.

But if the program guarantees that the current value of a variable is never read before it is written, then the current value is irrelevant. Abstractly, it makes sense to think of such a variable as “empty.”

This kind of thinking, in which a program comes to take on meaning beyond what is literally encoded, is a very important part of thinking like a computer scientist. Sometimes, the word “abstract” gets used so often and in so many contexts that it comes to lose its meaning. Nevertheless, abstraction is a central idea in computer science (as well as many other fields).

A more general definition of “abstraction” is “The process of modeling a complex system with a simplified description in order to suppress unnecessary details while capturing relevant behavior.”

## 11.10 Glossary

**encode:** To represent one set of values using another set of values, by constructing a mapping between them.

**shallow equality:** Equality of references. Two references that point to the same object.

**deep equality:** Equality of values. Two references that point to objects that have the same value.

**abstract parameter:** A set of parameters that act together as a single parameter.

**abstraction:** The process of interpreting a program (or anything else) at a higher level than what is literally represented by the code.

## 11.11 Exercises

**Exercise 11.2** Imagine a card game in which the objective is to get a collection of cards with a total score of 21. The total score for a hand is the total of the scores for all the cards. The score for each card is as follows: aces count as 1, all face cards count as ten; for all other cards the score is the same as the rank. Example: the hand (Ace, 10, Jack, 3) has a total score of  $1 + 10 + 10 + 3 = 24$ .

Write a method called `handScore` that takes an array of cards as an argument and that adds up (and returns) the total score. You should assume that the ranks of the cards are encoded according to the mapping in Section 11.2, with Aces encoded as 1.

**Exercise 11.3** The `printCard` method in Section 11.3 takes a `Card` object and returns a string representation of the card.

Write a class method for the `Card` class called `parseCard` that takes a `String` and returns the corresponding card. You can assume that the `String` contains the name of a card in a valid format, as if it had been produced by `printCard`.

In other words, the string will contain a single space between the rank and the word “of,” and between the word “of” and the suit. If the string does not contain a legal card name, the method should return a null object.

The purpose of this problem is to review the concept of parsing and implement a method that parses a specific set of strings.

**Exercise 11.4** Write a method called `suitHist` that takes an array of `Cards` as a parameter and that returns a histogram of the suits in the hand. Your solution should only traverse the array once.

**Exercise 11.5** Write a method called `isFlush` that takes an array of `Cards` as a parameter and that returns `true` if the hand contains a flush, and `false` otherwise. A flush is a poker hand that contains five or more cards of the same suit.

## Chapter 12

# Objects of Arrays

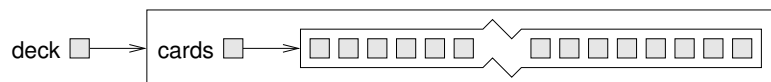
### 12.1 The Deck class

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 are going to create a new object, called a **Deck**, that contains an array of **Cards** as an instance variable.

The class definition looks like this

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

The name of the instance variable is **cards** to help distinguish the **Deck** object from the array of **Cards** that it contains. Here is a state diagram showing what a **Deck** object looks like with no cards allocated:



As usual, the constructor initializes the instance variable, but in this case it uses the **new** command to create the array of cards. It doesn't create any cards to go in it, though. For that we could write another constructor that creates a standard 52-card deck and populates it with **Card** objects:

```
public Deck () {  
    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++;
        }
    }
}
```

Notice how similar this method is to `buildDeck`, except that we had to change the syntax to make it a constructor. To invoke it, we use the `new` command:

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

Now that we have a `Deck` class, it makes sense to put all the methods that pertain to `Decks` in the `Deck` class definition. Looking at the methods we have written so far, one obvious candidate is `printDeck` (Section 11.7). Here's how it looks, rewritten to work with a `Deck` object:

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

The most obvious thing we have to 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, itself, an array. Therefore, 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.

For some of the other methods, it is not obvious whether they should be included in the `Card` class or the `Deck` class. For example, `findCard` takes a `Card` and a `Deck` as arguments; you could reasonably put it in either class. As an exercise, move `findCard` into the `Deck` class and rewrite it so that the first parameter is a `Deck` object rather than an array of `Cards`.

## 12.2 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 10.6 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 reassembling the deck by 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 same order you started in. For a discussion of that claim, see <http://www.wiskit.com/marilyn/craig.html> or do a web search with the keywords “perfect 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.length; i++) {  
    // choose a random number between i and deck.cards.length  
    // swap the ith card and the randomly-chosen card  
}
```

The nice thing about using 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 the parameters **low** and **high**, and **swapCards** which takes two indices and switches the cards at the indicated positions.

You can probably figure out how to write **randomInt** by looking at Section 10.6, although you will have to be careful about possibly generating indices that are out of range.

You can also figure out **swapCards** yourself. The only tricky thing is to decide whether to swap just the references to the cards or the contents of the cards. Does it matter which one you choose? Which is faster?

I will leave the remaining implementation of these methods as an exercise.

## 12.3 Sorting

Now that we have messed up the deck, we need a way to put it back in order. Ironically, there is an algorithm for sorting that is very similar to the algorithm for shuffling. This algorithm is sometimes 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.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 `findLowestCard`, that takes an array of cards and an index where it should start looking.

Once again, I am going to leave the implementation up to the reader.

## 12.4 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 that happens to have 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:

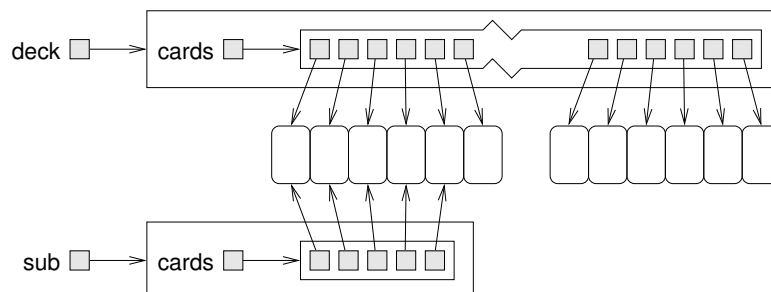
```
public static Deck subdeck (Deck deck, int low, int high) {
    Deck sub = new Deck (high-low+1);

    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 the `new` command, the constructor 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.





I have suggested that aliasing is not generally a good idea, since changes in one subdeck will be reflected in others, which is not the behavior you would expect from real cards and decks. But if the objects in question are immutable, then aliasing is less dangerous. In this case, there is probably no reason ever to change the rank or suit of a card. Instead we will create each card once and then treat it as an immutable object. So for **Cards** aliasing is a reasonable choice.

## 12.5 Shuffling and dealing

In Section 12.2 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 create and shuffle a deck:

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

Then, to deal out several hands, we can use **subdeck**:

```
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 out one card at a time 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. Beware of misleading analogies.

## 12.6 Mergesort

In Section 12.3, we saw a simple sorting algorithm that turns out not to be very efficient. In order 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 will 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* in order to achieve the performance advantage I promised. In order to make it work, though, you have to add a base case so that it doesn't recurse 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.” I have deliberately constructed this example to encourage you to make the leap of faith.

When you were using `sortDeck` to sort the subdecks, you didn't feel compelled to follow the flow of execution, right? You just assumed that the `sortDeck` method would work 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.

Well, 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!

## 12.7 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.

## 12.8 Exercises

**Exercise 12.1** Write a version of `findBisect` that takes a subdeck as an argument, rather than a deck and an index range (see Section 11.8). Which version is more error-prone? Which version do you think is more efficient?

**Exercise 12.2** In the previous version of the `Card` class, a deck is implemented as an array of `Cards`. For example, when we pass a “deck” as a parameter, the actual type of the parameter is `Card[]`.

In this chapter, we developed an alternative representation for a deck, an object type named `Deck` that contains an array of cards as an instance variable. In this exercise, you will implement the new representation of a deck.

- a. Add a second file, named `Deck.java` to the program. This file will contain the definition of the `Deck` class.
- b. Type in the constructors for the `Deck` class as shown in Section 12.1.
- c. Of the methods currently in the `Card` class, decide which ones would be more appropriate as members of the new `Deck` class. Move them there and make any changes necessary to get the program to compile and run again.
- d. Look over the program and identify every place where an array of `Cards` is being used to represent a deck. Modify the program throughout so that it uses a `Deck` object instead. You can use the version of `printDeck` in Section 12.1 as an example.

It is probably a good idea to make this transformation one method at a time, and test the program after each change. On the other hand, if you are confident you know what you are doing, you can make most of the changes with search-and-replace commands.

**Exercise 12.3** The goal of this exercise is to implement the shuffling and sorting algorithms from this chapter.

- a. 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 the references to the two cards, rather than the contents of the two objects. This is not only faster, but it makes it easier to deal with the case where cards are aliased.
- b. Write a method called `shuffleDeck` that uses the algorithm in Section 12.2. You might want to use the `randomInt` method from Exercise 10.2.
- c. Write a method called `findLowestCard` that uses the `compareCard` method to find the lowest card in a given range of the deck (from `lowIndex` to `highIndex`, including both).
- d. Write a method called `sortDeck` that arranges a deck of cards from lowest to highest.

**Exercise 12.4** In order to make life more difficult for card-counters, many casinos now use automatic shuffling machines that can do incremental shuffling, which means that after each hand, the used cards are returned to the deck and, instead of reshuffling the entire deck, the new cards are inserted in random locations.

Write a method called `incrementalShuffle` that takes a `Deck` and a `Card` and that inserts the card into the deck at a random location. This is an example of an **incremental algorithm**.

**Exercise 12.5** The goal of this exercise is to write a program that generates random poker hands and classifies them, so that we can estimate the probability of the various poker hands. Don't worry if you don't play poker; I'll tell you everything you need to know.

- a. As a warmup, write a program that uses `shuffleDeck` and `subdeck` to generate and print four random poker hands with five cards each. Did you get anything good? Here are the possible poker hands, in increasing order of value:
  - pair:** two cards with the same rank
  - two pair:** two pairs of cards with the same rank
  - three of a kind:** three cards with the same rank
  - straight:** five cards with ranks in sequence
  - flush:** five cards with the same suit
  - full house:** three cards with one rank, two cards with another
  - four of a kind:** four cards with the same rank
  - straight flush:** five cards in sequence and with the same suit
- b. Write a method called `isFlush` that takes a `Deck` as a parameter and returns a boolean indicating whether the hand contains a flush.
- c. Write a method called `isThreeKind` that takes a hand and returns a boolean indicating whether the hand contains Three of a Kind.
- d. Write a loop that generates a few thousand hands and checks whether they contain a flush or three of a kind. Estimate the probability of getting one of those hands.
- e. Write methods that test for the other poker hands. Some are easier than others. You might find it useful to write some general-purpose helper methods that can be used for more than one test.
- f. In some poker games, players get seven cards each, and they form a hand with the best five of the seven. Modify your program to generate seven-card hands and recompute the probabilities.

**Exercise 12.6** As a special challenge, think of algorithms to check for various poker hands if there are wild cards. For example, if "deuces are wild," that means that if you have a card with rank 2, you can use it to represent any card in the deck.

**Exercise 12.7** The goal of this exercise is to implement mergesort.

- a. Using the pseudocode in Section 12.6, write the method called `merge`. Be sure to test it before trying to use it as part of a `mergeSort`.
- b. 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.
- c. 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
```



## Chapter 13

# Object-oriented programming

### 13.1 Programming languages and styles

There are many programming languages in the world, and almost as many programming styles (sometimes called paradigms). Three styles that have appeared in this book are procedural, functional, and object-oriented. Although Java is usually thought of as an object-oriented language, it is possible to write Java programs in any style. The style I have demonstrated in this book is pretty much procedural. Existing Java programs and the built-in Java packages are written in a mixture of all three styles, but they tend to be more object-oriented than the programs in this book.

It's not easy to define what object-oriented programming is, but here are some of its characteristics:

- Object definitions (classes) usually correspond to relevant real-world objects. For example, in Chapter 12.1, the creation of the `Deck` class was a step toward object-oriented programming.
- The majority of methods are object methods (the kind you invoke on an object) rather than class methods (the kind you just invoke). So far all the methods we have written have been class methods. In this chapter we will write some object methods.
- The language feature most associated with object-oriented programming is **inheritance**. I will cover inheritance later in this chapter.

Recently object-oriented programming has become quite popular, and there are people who claim that it is superior to other styles in various ways. I hope that by exposing you to a variety of styles I have given you the tools you need to understand and evaluate these claims.

## 13.2 Object and class methods

There are two types of methods in Java, called **class methods** and **object methods**. So far, every method we have written has been a class method. Class methods are identified by the keyword **static** in the first line. Any method that does not have the keyword **static** is an object method.

Although we have not written any object methods, we have invoked some. Whenever you invoke a method “on” an object, it’s an object method. For example, `charAt` and the other methods we invoked on `String` objects are all object methods.

Anything that can be written as a class method can also be written as an object method, and vice versa. Sometimes it is just more natural to use one or the other. For reasons that will be clear soon, object methods are often shorter than the corresponding class methods.

## 13.3 The current object

When you invoke a method on an object, that object becomes **the current object**. Inside the method, you can refer to the instance variables of the current object by name, without having to specify the name of the object.

Also, you can refer to the current object using the keyword **this**. We have already seen **this** used in constructors. In fact, you can think of constructors as being a special kind of object method.

## 13.4 Complex numbers

As a running example for the rest of this chapter we will consider a class definition for complex numbers. Complex numbers are useful for many branches of mathematics and engineering, and many computations are performed using complex arithmetic. A complex number is the sum of a real part and an imaginary part, and is usually written in the form  $x + yi$ , where  $x$  is the real part,  $y$  is the imaginary part, and  $i$  represents the square root of -1. Thus,  $i \cdot i = -1$ .

The following is a class definition for a new object type called **Complex**:

```
class Complex
{
    // instance variables
    double real, imag;

    // constructor
    public Complex () {
        this.real = 0.0;  this.imag = 0.0;
    }
}
```



```

        // constructor
        public Complex (double real, double imag) {
            this.real = real;
            this.imag = imag;
        }
    }

```

There should be nothing surprising here. The instance variables are doubles that contain the real and imaginary parts. The two constructors are the usual kind: one takes no parameters and assigns default values to the instance variables, the other takes parameters that are identical to the instance variables. As we have seen before, the keyword `this` is used to refer to the object being initialized.

In `main`, or anywhere else we want to create `Complex` objects, we have the option of creating the object and then setting the instance variables, or doing both at the same time:

```

Complex x = new Complex ();
x.real = 1.0;
x.imag = 2.0;
Complex y = new Complex (3.0, 4.0);

```

## 13.5 A function on Complex numbers

Let's look at some of the operations we might want to perform on complex numbers. The absolute value of a complex number is defined to be  $\sqrt{x^2 + y^2}$ . The `abs` method is a pure function that computes the absolute value. Written as a class method, it looks like this:

```

// class method
public static double abs (Complex c) {
    return Math.sqrt (c.real * c.real + c.imag * c.imag);
}

```

This version of `abs` calculates the absolute value of `c`, the `Complex` object it receives as a parameter. The next version of `abs` is an object method; it calculates the absolute value of the current object (the object the method was invoked on). Thus, it does not receive any parameters:

```

// object method
public double abs () {
    return Math.sqrt (real*real + imag*imag);
}

```

I removed the keyword `static` to indicate that this is an object method. Also, I eliminated the unnecessary parameter. Inside the method, I can refer to the instance variables `real` and `imag` by name without having to specify an object. Java knows implicitly that I am referring to the instance variables of the current object. If I wanted to make it explicit, I could have used the keyword `this`:

```

// object method
public double abs () {

```

```
        return Math.sqrt (this.real * this.real + this.imag * this.imag);
    }
```

But that would be longer and not really any clearer. To invoke this method, we invoke it on an object, for example

```
Complex y = new Complex (3.0, 4.0);
double result = y.abs();
```

### 13.6 Another function on Complex numbers

Another operation we might want to perform on complex numbers is addition. You can add complex numbers by adding the real parts and adding the imaginary parts. Written as a class method, that looks like:

```
public static Complex add (Complex a, Complex b) {
    return new Complex (a.real + b.real, a.imag + b.imag);
}
```

To invoke this method, we would pass both operands as arguments:

```
Complex sum = add (x, y);
```

Written as an object method, it would take only one argument, which it would add to the current object:

```
public Complex add (Complex b) {
    return new Complex (real + b.real, imag + b.imag);
}
```

Again, we can refer to the instance variables of the current object implicitly, but to refer to the instance variables of `b` we have to name `b` explicitly using dot notation. To invoke this method, you invoke it on one of the operands and pass the other as an argument.

```
Complex sum = x.add (y);
```

From these examples you can see that the current object (`this`) can take the place of one of the parameters. For this reason, the current object is sometimes called an **implicit parameter**.

### 13.7 A modifier

As yet another example, we'll look at `conjugate`, which is a modifier method that transforms a `Complex` number into its complex conjugate. The complex conjugate of  $x + yi$  is  $x - yi$ .

As a class method, this looks like:

```
public static void conjugate (Complex c) {
    c.imag = -c.imag;
}
```

As an object method, it looks like

```
public void conjugate () {  
    imag = -imag;  
}
```

By now you should be getting the sense that converting a method from one kind to another is a mechanical process. With a little practice, you will be able to do it without giving it much thought, which is good because you should not be constrained to writing one kind of method or the other. You should be equally familiar with both so that you can choose whichever one seems most appropriate for the operation you are writing.

For example, I think that `add` should be written as a class method because it is a symmetric operation of two operands, and it makes sense for both operands to appear as parameters. To me, it seems odd to invoke the method on one of the operands and pass the other as an argument.

On the other hand, simple operations that apply to a single object can be written most concisely as object methods (even if they take some additional arguments).

## 13.8 The toString method

Every object type has a method called `toString` that generates a string representation of the object. When you print an object using `print` or `println`, Java invokes the object's `toString` method. The default version of `toString` returns a string that contains the type of the object and a unique identifier (see Section 9.6). When you define a new object type, you can **override** the default behavior by providing a new method with the behavior you want.

Here is what `toString` might look like for the `Complex` class:

```
public String toString () {  
    return real + " + " + imag + "i";  
}
```

The return type for `toString` is `String`, naturally, and it takes no parameters. You can invoke `toString` in the usual way:

```
Complex x = new Complex (1.0, 2.0);  
String s = x.toString ();
```

or you can invoke it indirectly through `println`:

```
System.out.println (x);
```

In this case, the output is `1.0 + 2.0i`.

This version of `toString` does not look good if the imaginary part is negative. As an exercise, write a better version.

## 13.9 The equals method

When you use the `==` operator to compare two objects, what you are really asking is, "Are these two things the same object?" That is, do both objects refer to the same location in memory.

For many types, that is not the appropriate definition of equality. For example, two complex numbers are equal if their real parts are equal and their imaginary parts are equal. They don't have to be the same object.

When you define a new object type, you can provide your own definition of equality by providing an object method called `equals`. For the `Complex` class, this looks like:

```
public boolean equals (Complex b) {  
    return (real == b.real && imag == b.imag);  
}
```

By convention, `equals` is always an object method that returns a `boolean`.

The documentation of `equals` in the `Object` class provides some guidelines you should keep in mind when you make up your own definition of equality:

The `equals` method implements an equivalence relation:

- It is reflexive: for any reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is consistent: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`.
- For any reference value `x`, `x.equals(null)` should return `false`.

The definition of `equals` I provided satisfies all these conditions except one. Which one? As an exercise, fix it.

## 13.10 Invoking one object method from another

As you might expect, it is legal and common to invoke one object method from another. For example, to normalize a complex number, you divide both parts by the absolute value. It may not be obvious why this is useful, but it is.

Let's write the method `normalize` as an object method, and let's make it a modifier.

```
public void normalize () {  
    double d = this.abs();  
    real = real/d;  
    imag = imag/d;  
}
```

The first line finds the absolute value of the current object by invoking `abs` on the current object. In this case I named the current object explicitly, but I could have left it out. If you invoke one object method within another, Java assumes that you are invoking it on the current object.

**Exercise 13.1** Rewrite `normalize` as a pure function. Then rewrite it as a class method.

## 13.11 Oddities and errors

If you have both object methods and class methods in the same class definition, it is easy to get confused. A common way to organize a class definition is to put all the constructors at the beginning, followed by all the object methods and then all the class methods.

You can have an object method and a class method with the same name, as long as they do not have the same number and types of parameters. As with other kinds of overloading, Java decides which version to invoke by looking at the arguments you provide.

Now that we know what the keyword `static` means, you have probably figured out that `main` is a class method, which means that there is no “current object” when it is invoked.

Since there is no current object in a class method, it is an error to use the keyword `this`. If you try, you might get an error message like: “Undefined variable: this.” Also, you cannot refer to instance variables without using dot notation and providing an object name. If you try, you might get “Can’t make a static reference to nonstatic variable...” This is not one of the better error messages, since it uses some non-standard language. For example, by “nonstatic variable” it means “instance variable.” But once you know what it means, you know what it means.

## 13.12 Inheritance

The language feature that is most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of a previously-defined class (including built-in classes).

The primary advantage of this feature is that you can add new methods or instance variables to an existing class without modifying the existing class. This is particularly useful for built-in classes, since you can’t modify them even if you want to.

The reason inheritance is called “inheritance” is that the new class inherits all the instance variables and methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class.

### 13.13 Drawable rectangles

An example of inheritance, we are going to take the existing `Rectangle` class and make it “drawable.” That is, we are going to create a new class called `DrawableRectangle` that will have all the instance variables and methods of a `Rectangle`, plus an additional method called `draw` that will take a `Graphics` object as a parameter and draw the rectangle.

The class definition looks like this:

```
import java.awt.*;

class DrawableRectangle extends Rectangle {

    public void draw (Graphics g) {
        g.drawRect (x, y, width, height);
    }
}
```

Yes, that’s really all there is in the whole class definition. The first line imports the `java.awt` package, which is where `Rectangle` and `Graphics` are defined.

The next line indicates that `DrawableRectangle` inherits from `Rectangle`. The keyword `extends` is used to identify the class we are inheriting from, which is called the **parent class**.

The rest is the definition of the `draw` method, which refers to the instance variables `x`, `y`, `width` and `height`. It might seem odd to refer to instance variables that don’t appear in this class definition, but remember that they are inherited from the parent class.

To create and draw a `DrawableRectangle`, you could use the following:

```
public static void draw (Graphics g, int x, int y, int width, int height) {
    DrawableRectangle dr = new DrawableRectangle ();
    dr.x = 10;          dr.y = 10;
    dr.width = 200;     dr.height = 200;
    dr.draw (g);
}
```

The parameters of `draw` are a `Graphics` object and the bounding box of the drawing area (not the coordinates of the rectangle).

It might seem odd to use the `new` command for a class that has no constructors. `DrawableRectangle` inherits the default constructor of its parent class, so there is no problem there.

We can set the instance variables of `dr` and invoke methods on it in the usual way. When we invoke `draw`, Java invokes the method we defined in `DrawableRectangle`. If we invoked `grow` or some other `Rectangle` method on `dr`, Java would know to use the method defined in the parent class.