Algorithms in Java: Shuffling and Random Numbers

Algorithms

Introduction

It's impossible to study programming – or even talk in depth about it – without talking about *algorithms*. They're an essential part of programming. Fortunately, all of us – programmers and non-programmers alike – have experience with algorithms. Every time you give someone driving directions; every time you use a recipe to cook; every time you follow instructions for using a piece of software – all of these activities (and countless others) involve algorithms.

So what is an algorithm? Simply put, an algorithm is a set of instructions, intended to solve a specific problem or accomplish a specific, finite goal, in a direct fashion.

What Makes an Effective Algorithm?

In general, ambiguity doesn't make for a good algorithm; this is particularly true in programming, because computers are mostly incapable of knowing what we want them to do, unless we spell it out precisely. So the algorithms we use should also be precise – even before we translate them into a programming language.

An effective algorithm usually has the following characteristics:

- Clear conditions of application: What task is the algorithm intended to accomplish, and what are the requirements for us to be able to use it?
- Initialization steps & starting conditions: What do we do to prepare our data for the algorithm? How can we tell if we're ready to proceed?
- Explicit stopping conditions and states: How do we know when we're done? How do we know whether we met the desired goal?
- Unambiguous steps that lead from the starting conditions to the stopping conditions.

Note that "unambiguous" is a subjective term. No matter how careful we are, there will always be details left out that the writer assumes are understood. This is a problem in some cases, but not others. For example, a complicated recipe will certainly omit definitions for terms assumed to be understood by an experienced chef; if the recipe isn't intended for beginners, including such definitions would be a waste of space.

Shuffling: The Flip Side of Sorting

The shuffling problem is easily stated: How can we rearrange a list of items so that the order is random and fair?

Fortunately, this isn't a very difficult problem – though it's easy to do it incorrectly. There are two commonly used approaches to shuffling in computer programs:

Sorting on a random number

1. Assign a randomly generated value to each item to be shuffled.
2. Sort the items, in order of the assigned random numbers.
3. Stop. The list is now shuffled.

Though the need for sorting makes this method less efficient than the one that follows, it's very easy to implement in some languages and environments – for example, this can be a good approach for retrieving and presenting database records in a random order.

Durstenfeld's version of the Fisher-Yates shuffle (aka Knuth shuffle)

1. Begin with the list of items X, containing the N items $x_{1,}\ x_{2,}\ \ldots,\ x_N$.
2. For each integer value j, starting at 1 and ending with $N-1$, do the following:
   a. Generate a random value k, which can be any one of $\{j,\ j+1,\ \ldots,\ N\}$; each of the values must be equally likely to be selected.
   b. Exchange the positions of x_k and x_j in the list. Note that it's possible that $k = j$; obviously, no exchange is needed when that's the case.
3. Stop. The list is now shuffled.

As we repeat steps 2a-b with values of j from 1 to $N-1$ (such repetition is called *iteration*), each randomly selected item is shuffled by moving it to the start of the unshuffled items, trading places with the item already there. In effect, the list is divided into two parts, one shuffled and one unshuffled, with the latter growing as the former shrinks, until no unshuffled items remain.

Exercise 1: Shuffling a List of Six Items by Rolls of a Die

Using a six-sided die to generate random numbers, let's shuffle the words "apple", "banana", "chile", "donut", "egg", and "flan" into a random order. To help us keep track of the items as we shuffle them, we'll use tables 1-3.

		Items (N = Number of Items = 6)					
		Position (i)					
Iteration (j)	**Roll (k)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial		apple	banana	chile	donut	egg	flan
1							
2							
3							
4							
5							

Table 1: Example of Fisher-Yates shuffle, initial state

We start at row 1 – i.e. the row where 1 appears in the **Iteration (*j*)** column – and roll the die to get a value of k. For example, assume we roll a 4. We write this value in the **Roll (*k*)** column. Next, we copy the six words in our list from the previous row to the current row. As we do so, we exchange the item in column 4 (since that was our roll) with the first unshuffled item, which is in column 1; thus we exchange "donut" with "apple". The result appears in table 2.

		Items (N = Number of Items = 6)					
		Position (i)					
Iteration (j)	**Roll (k)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial		apple	banana	chile	donut	egg	flan
1	4	donut	banana	chile	apple	egg	flan
2							
3							
4							
5							

Table 2: Example of Fisher-Yates shuffle, after 1 iteration

(The line that zig-zags through the table divides the list into shuffled and unshuffled items. In our example, "donut" is now shuffled, and will stay in column 1.)

In each successive *iteration* (a repeated set of steps in an algorithm), we move down one row (increasing the value of j by 1) and roll the die. If the roll is less than the value in

the **Iteration (*j*)** column, we keep rolling until we get a value greater than or equal to j; then, we write our roll in the **Roll (*k*)** column. Finally, we copy the items from the previous row to the current row, exchanging the items in columns j and k.

Assume that in iteration 2, we roll a 1. However, because 1 is less than the current value of j, we roll again; this time, we get a 3, so we write that value in the **Roll (*k*)** column. Then, we copy our six items from the previous row to the current row, exchanging the item in column k (or 3), "chile", with the item in column j (or 2), "banana". Our randomly selected item, "chile", is now shuffled.

		Items (*N* = Number of Items = 6)					
		Position (*i*)					
Iteration (*j*)	**Roll (*k*)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial		apple	banana	chile	donut	egg	flan
1	4	donut	banana	chile	apple	egg	flan
2	3	donut	chile	banana	apple	egg	flan
3							
4							
5							

Table 3: Example of Fisher-Yates shuffle, after 2 iterations

Complete this shuffle on your own, by performing iterations 3-5.

Exercise 2: Additional Questions and Tasks

I. What is a fair shuffle? Does random always imply fair?

II. If each iteration of steps 2a-b of the Fisher-Yates algorithm shuffles one item in the list, how is it that we're able to shuffle 6 items in 5 iterations? More generally, how are we able to shuffle all N items in $N-1$ iterations?

III. In any given iteration of steps 2a-b of the Fisher-Yates shuffle, it's possible that item j will trade places with itself. Even if we don't need to do anything to exchange an item with itself, it might seem inefficient to allow this to happen. Would eliminating this possibility, by limiting our random selection to the range from $j+1$ to N, still give us a complete, fair shuffle?

IV. Is there any value in performing a Fisher-Yates shuffle multiple times in a row, similar to the way we would manually shuffle a deck of cards several times?

Shuffling with Java: Lotteries

Introduction

Lotteries have existed – legally and illegally, and with many variations – for over 2000 years. In the U.S.A., lotteries were illegal for most of the 20th century, but by the 1960s, some states were modifying their constitutions to allow state lotteries. 43 states now run (or participate in) lotteries, using them as a source of state revenues. All of these lotteries set the payoffs so that the total amount paid to the winners is always much less than the total amount paid in, giving the state an unbeatable advantage.

In most of the state lotteries, the drawing of numbers is done with actual balls in a container. However, some lotteries are conducted electronically; also, state lotteries usually offer some form of "quick pick" option, letting players buy lottery tickets with the numbers selected at random by computer. So we'll write a Java program that generates our own lottery draws or quick picks.

(The next three exercises assume the DrJava development environment. Other IDEs can be used to write and compile the code in exercises 3 and 5, but exercise 4 depends on the interactive features of DrJava.)

Exercise 3: Writing a Lottery Program in Java

One of the first things we need to do, when writing any Java program, is decide what *classes* we need. In Java, classes are units of Java code, which generally perform one or more of three primary roles:

- A set of related *methods* (named procedures that perform specific tasks) that operate on variables of the basic Java data types, or object types defined by other classes.

- A set of methods to manage the execution (and termination) of Java programs, applets, etc.

- An encapsulation of the attributes and behaviors of a type of object, often corresponding to a physical or logical object related to the problem we're working on. For example, a traffic simulation program might have `Vehicle` and `TrafficSignal` classes that define variables and methods for managing the speed, location, and direction of vehicles, and the states of traffic signals.

In this case, we need a class that encapsulates the data for a lottery – i.e. the set of numbers that can be selected – with one very important behavior:

- Select a subset of the numbers at random, *without replacement* – i.e. without putting each number back after it's selected. In most lotteries, the order in which the numbers are selected is irrelevant – in fact, they're generally sorted in ascending order before they're announced. So our class will follow that example, and sort the selected subset of numbers in ascending order.

We'll begin by creating a `Lottery` class, with an *array* of integers to hold the lottery numbers, and a *constructor* that will create and fill the array. (We'll learn what those terms mean in the next few minutes.)

Create a new class file (with the **File/New** menu or the **New** button), and write the following code (the line numbers are for reference only; don't write them in your code):

```
 1 public class Lottery {
 2 
 3     private int[] numbers;
 4 
 5     public Lottery(int maximum) {
 6         numbers = new int[maximum];
 7         for (int i = 0; i < numbers.length; i++) {
 8             numbers[i] = i + 1;
 9         }
10     }
11 
12 }
```

Let's review the most important elements of the code:

- Our `class` is named `Lottery`, and it has `public` visibility (line 1).

- The left and right curly braces on lines 1 and 12 (respectively) enclose the code that makes up the implementation or *body* of the `Lottery` class.

- Inside the body of the class, we have the `private` variable `numbers` (line 3). It's a reference to an `int[]`, or an *array* of integers. An array is a variable that can hold multiple items of the same type, where each item is accessed by a numbered index – e.g. `numbers[0]`, `numbers[1]`, and `numbers[2]` are the first three integers in the `numbers` array. (In most programming languages, we start counting array elements with 0, instead of 1.)

- A **public** *constructor* begins on lines. A constructor is a special kind of method, whose job is to initialize the data used by an object of this class type. A constructor always has the same name as the class, and no return type is specified. Inside the parentheses that follow the name, we see that when this constructor is called, some additional information must be provided: an **int** called **maximum**. Later, when we're using our **Lottery** class type, we'll need to provide this additional data when we call the constructor.

  Inside the curly braces of the constructor (lines 5 and 10), we have the body of the constructor – i.e. the code that initializes our lottery numbers. First we use the **new** keyword to allocate space for the **numbers** array, with enough elements for the range of numbers from 1 to **maximum** (line 6).

  Next, we need to put our lottery numbers into the array – but remember, the indices for accessing array elements begin with 0. So we use the **for** statement (line 7) to iterate over the elements in the array, using the **int** variable **i** as an iteration counter with a starting value of 0, and continuing as long as **i** is less than **maximum**, After each iteration, we increment the value of **i** (using **i++**). In each iteration, we execute the code between the curly braces on lines 7 and 9; in that code, we place into element **i** of the array a value equal to one more than **i**; for example, we put the number 1 into element 0, 2 into element 1, etc. – all the way up to the value of **maximum**, which we put into element **maximum - 1**.

Before we go any further, make sure you've saved your work. Since the class is named **Lottery**, the file name must be `Lottery.java`. – i.e. with the exact same spelling and capitalization as the class name, and with the `.java` extension (DrJava adds this extension automatically, if you don't specify one).

Next, use the **Compile** button to compile the code you've written – making sure to fix any errors that are reported along the way, until your code compiles without error. (Ask for help from the instructors as needed.)

So far, we've taken care of the first part of the required functionality. Now, we'll write code to select a subset of the numbers at random. We'll use shuffling to mix up the numbers, but we're going to change the algorithm just a bit. Remember that in order to shuffle N items, we perform $N-1$ iterations of selecting an unshuffled item at random and swapping it with the last unshuffled item. But in this case, we don't really need to shuffle all of the numbers; we only need to shuffle enough four our subset.

In writing the code that follows, the code you already wrote is grayed-out. *Don't re-write the grayed-out code; simply add the new code to it.*

```
 1 import java.util.Arrays;
 2 import java.util.Random;
 3
 4 public class Lottery {
 5
 6     private int[] numbers;
 7     private Random rng = new Random();
 8
 9     public Lottery(int maximum) {
10         numbers = new int[maximum];
11         for (int i = 0; i < numbers.length; i++) {
12             numbers[i] = i + 1;
13         }
14     }
15
16     private void mix(int iterations) {
17         for (int i = 0; i < iterations; i++) {
18             int shuffleIndex =
19                 i + rng.nextInt(numbers.length - i);
20             int temp = numbers[shuffleIndex];
21             numbers[shuffleIndex] = numbers[i];
22             numbers[i] = temp;
23         }
24     }
25
26     public int[] pick(int numbersToPick) {
27         int[] selection;
28         mix(numbersToPick);
29         selection = Arrays.copyOf(numbers, numbersToPick);
30         Arrays.sort(selection);
31         return selection;
32     }
33
34 }
```

Let's review the additions, starting at the top:

- We added two `import` statements (lines 1-2). These tell Java that our code will use the `Arrays` and `Random` classes, both located in the `java.util` package.

- We've added the `private` variable `rng` to the class, in line 7. The type is `Random`, a class that implements the standard Java random number generator. We've also created and initialized this variable by calling the `Random` constructor, using the `new` keyword.

- Lines 16-24 contain the declaration and body of the **private** (not visible outside the class) method **mix**. When this method is called, it expects to receive an **int**, referred to by the method as **iterations**. This is the number of iterations of steps 2a-b in the Fisher-Yates shuffle that this method will execute; since each iteration shuffles one element in the **numbers** array, this is also the number of lottery numbers shuffled by this method.

  We use the **for** statement (line 17), with the counter variable **i**, to repeat the code between the curly braces on lines 17 and 23 a total of **iterations** times.

  In lines 18-19, a *local variable* (a variable that exists only inside the current block of statements) named **shuffleIndex** is given a random integer value (generated by **rng**) between **i** (inclusive) and **numbers.length** (exclusive).

  Now, we exchange two items in the **numbers** array: First, the value of the lottery number in the **shuffleIndex** element of **numbers** is assigned to **temp**. Then, we take the lottery number in the **i** element of **numbers** and put it in the **shuffleIndex** element. Finally, we take the lottery number we stored in **temp**, and put it into the **i** element of the **numbers** array. This has the effect of swapping a randomly selected unshuffled item with the first unshuffled item in the **numbers** array; this randomly selected item is now shuffled.

- We've declared the **public** method **pick** (lines 26-33), which returns an **int[]**, or array of integers. When this method is called, an additional piece of data must be provided: an **int**, which this method refers to as **numbersToPick**.

  In the body of the **pick** method, we declare a local variable named **selection** (line 27). Like the **numbers** variable, **selection** refers to an array of integers.

  Next, we call the **mix** method, specifying **numbersToPick** (line 28). The **mix** method shuffles that many lottery numbers to the end of the **numbers** array.

  After **mix** does the shuffling, we copy the shuffled numbers from the **numbers** array to the **selection** array, using the **copyOf** method in the **Arrays** class (lines 29-30). In this call, we specify the array we're copying data from (**numbers**), and the number of elements (starting with element 0) to copy.

  We use another method of the **Arrays** class, **sort**, to sort the elements of the **selection** array in ascending order (line 31).

  Finally, we return the **selection** array as the result of this method (line 32).

Save and compile the `Lottery` class. If any error messages appear, fix the reported problems, then save and compile again, until you can compile without any errors.

Exercise 4: Testing the Lottery Class Interactively

Our `Lottery` class is complete for our purposes today, but it isn't a Java program. Fortunately, the **Interactions** pane of DrJava is very handy for experimenting with Java, without first having to write complete Java programs.

In the **Interactions** pane, type the following to create a new variable of the `Lottery` type, with a range of numbers of 1 through 40. Note that the greater-than sign (>) isn't part of the text you type; it's a prompt character that DrJava displays. Also, you need to press the *enter* key at the end of each line you type.

```
> Lottery lotto = new Lottery(40);
```

If you typed the line correctly, you should see the input prompt (>) again.

Now we have a variable `lotto`, of the `Lottery` type, ready to select numbers in the range from 1 to 40. Let's use the `pick` method to have it pick 6 numbers for us:

```
> lotto.pick(6)
```

Notice that there's no semicolon at the end of the line this time. Java requires semicolons at the ends of statements; however, the DrJava **Interactions** pane recognizes a statement without a semicolon as shorthand for “display the value of this expression”.

After you hit the *enter* key, did you see an array of 6 distinct numbers between 1 and 40, sorted in ascending order?

Have `lotto` pick a few more sets of 6 numbers, using the same statement as before. (You can use the arrow keys to cycle through the lines previously typed.)

We can also experiment with different kinds of lotteries. For example, in keno, the casino has a fixed draw size (usually 20 numbers from 1 to 80), but the subsets picked by a player don't have to be the same size as the casino's draw.

Use this code in the **Interactions** pane, to create a variable of the `Lottery` type with numbers ranging from 1 to 80, and then to draw and display 20 numbers:

```
> Lottery keno = new Lottery(80);
> keno.pick(20)
```

Exercise 5: Writing a Java Program That Uses the Lottery Class

So far, we've tested our `Lottery` class using the DrJava **Interactions** pane. However, most people that run Java programs don't use DrJava; they rely on the standard Java launcher to run Java programs. For now, we'll keep using DrJava, but we will write a Java program that can be launched from a Windows or Unix/Linux command line.

For this exercise, we'll write a program to generate several picks for the NM Lottery Roadrunner Cash game, in which the player picks 5 separate numbers from 1 to 37.

For a Java program, we start (again) with a class. Create a new file, and type this code:

```
 1 public class Roadrunner {
 2 
 3     private static final int MAXIMUM_NUMBER = 37;
 4     private static final int NUMBER_TO_PICK = 5;
 5     private static final int NUMBER_OF_TICKETS = 10;
 6 
 7     public static void main(String[] args) {
 8 
 9     }
10 
11 }
```

Let's take a look at the code in more detail:

- As before, we declare our class with the `class` keyword, the name of the class, and a set of curly braces (lines 1 and 11) to hold the implementation of the class. We also set the visibility of the class to `public` (line 1), so that it can be seen from other classes – and most importantly, by the Java launcher.

- Just inside the class (lines 3-5), we have three variables with an important combination of keywords. A variable that's declared `static final` is actually a *constant*: once we assign an initial value to a constant, the Java compiler won't let a different value be assigned. We've assigned initial values to all three of these constants, so we can be certain that those values won't change.

  In these constants, we've stored the upper limit of the range of numbers for this lottery (line 3), how many numbers must be selected in each pick (line 4), and how many separate sets of numbers we'll pick (line 5).

- In line 7, we see the `static` keyword again – this time with the `main` method. The most important thing to know about this is that when the Java launcher tries

to run a class as a Java program, it looks in that class for a **public static** method called **main**, which doesn't return any data when it's done (that what **void** return type means). Finally, inside the parentheses of the **main** method declaration (line 7), we see that the method expects to receive an array of **String** objects; this is also a requirement for the **main** method of a Java program. (The arguments are values that can be specified on the command line when a Java program is launched – but our program will ignore any additional information passed to it this way.)

If the Java launcher finds a method with the **public static void main(String[])** signature, it calls it; if it doesn't find it, it reports an error and terminates execution.

As with our earlier methods, we see that the **main** method has a set of curly braces (lines 7 and 9); we'll write the body of the method – i.e. the top-level steps of our Roadrunner program – between these braces.

Save the file. Be sure to save it in the same directory as your `Lottery.java` file. Also, remember that the file name must match the class name (with the same spelling and capitalization), with the addition of the `.java` extension; in this case, the file must be named `Roadrunner.java`.

Compile the file. If you get error messages, read them carefully to try to figure out what's wrong, and try to fix the problems.

Some of the last few lines of code we need to add will look familiar after the typing you did in the last exercise. Basically, we're putting the kinds of statements that you typed manually in the **Interactions** pane into our program. However, there are a couple of things we need to include in our code that the **Interactions** pane did for us automatically. First, it recognized, by our leaving a semicolon off the end of a line, that we wanted to display a value on the screen; second, it converted the array of integers we wanted to display into a *string* (a sequence of text characters, which can include letters, numbers, punctuation, etc.), using an easy-to-read format. Fortunately, the code to convert and display an array isn't difficult to write. (Remember not to retype the grayed-out code.)

```java
1  import java.util.Arrays;
2
3  public class Roadrunner {
4
5      private static final int MAXIMUM_NUMBER = 37;
6      private static final int NUMBER_TO_PICK = 5;
7      private static final int NUMBER_OF_TICKETS = 10;
8
9      public static void main(String[] args) {
10         Lottery lotto = new Lottery(MAXIMUM_NUMBER);
11         for (int i = 1; i <= NUMBER_OF_TICKETS; i++) {
12             int[] selection = lotto.pick(NUMBER_TO_PICK);
13             String selectString = Arrays.toString(selection);
14             System.out.printf("Ticket #%d: %s\n",
15                 i, selectString);
16         }
17     }
18
19 }
```

There are a few things to notice this time:

- Once again, we import the **`Arrays`** class from the **`java.util`** package (line 1).

- We're using the **`for`** keyword to iterate again (line 11, with curly braces in lines 11 and 16). This time, we're using it to count up to the desired number of "tickets" - i.e. the separate subsets of lottery numbers the program is drawing for us.

- For each ticket, numbers are drawn using the **`pick`** method (line 12), and the result is assigned to the local variable **`selection`**.

- In line 13, we use the **`Arrays.toString`** method to convert the **`int[]`** with the selected subset of numbers to a **`String`**, storing the result in the **`selectString`** variable for later display.

- One piece of code that might look unfamiliar is the **`System.out.printf`** method call in lines 14-15. This method writes data to *standard output*, a text-based console display. Primitive as it is, this is one of the easiest ways to display output from simple programs.

  The **`printf`** method is used to format and write data values to standard output, or to a file. With this method, we first specify a format string, which may contain

static text along with one or more placeholders (you can spot these placeholders easily: they start with the percent sign); then, we pass the data to be displayed, as additional parameters in the method call. Before the text is written out, the placeholders in the format string are replaced by the data values.

In this case, the format string is: "`Ticket #%d: %s\n`". The placeholders are `%d` (for an integer value displayed as a decimal number) and `%s` (for a string value); the `\n` is the code to go to a new line. The first placeholder is replaced by the value of `i` (the variable we're using to count up the tickets), and the second placeholder is replaced by the value of `selectString`.

Save, compile, and run your program. Is the result what you expected to see?

Exercise 6: Additional Questions and Tasks

I. In all, how many different Roadrunner picks are possible? In other words, how many *combinations* (distinct subsets, without regard to the order of items in a subset) of 5 numbers from the set {1, 2, …, 37} are possible?

II. Can you think of a way to modify your `Roadrunner` class, so that you could use it to check whether the method we're using generates all possible combinations, in approximately equal proportions?

III. Modify your program (or create a new `Keno` class with a `main` method, that can be executed as a Java program) so that 20 numbers in the range from 1 to 80 are drawn, for an automated keno game.

IV. Modify your program (or create a new Java program class) to generate quick pick numbers for PowerBall, where 5 numbers from 1 to 59 are selected without replacement, and then a 6th number from 1 to 39 is selected.

Random and Pseudo-random Numbers

The Need for Randomness

In previous exercises, we used Java's random number generator class (`Random`) to generate random values that we used to shuffle lottery numbers. Many different kinds of computer programs benefit from random numbers; when we're working on computational science projects, and building scientific simulations, the need for random numbers becomes even more critical.

But there's a wrinkle: contrary to what many might think, very little that happens in most computers – at least in an easily observed fashion – is truly random. Fortunately for us, we often don't need true randomness: numbers that appear random, even though they're not, are good enough in many cases. Numbers of this type are called *pseudo-random numbers*.

The Appearance of Randomness

What can we observe in this portion of a sequence of numbers?

$$S = 0, 5, 6, 3, 12, 1, 2, 15, 8, 13, 14, 11, 4, 9, 10, 7, \ldots$$

At first, we might see some apparently random numbers, all less than 16. We might notice that none of the numbers is repeated – but we could chalk that up to having such a short portion of the sequence (or, we might speculate that the numbers are being selected by sampling without replacement). Looking closer, we might suspect some patterns in the differences between successive numbers; maybe that's just a coincidence. Finally, we might notice that the sequence alternates between odd and even values; that seems very unlikely to be a coincidence.

Let's look at more of the sequence:

$$S = 0,5,6,3,12,1,2,15,8,13,14,11,4,9,10,7,0,5,6,3,12,1,2,15,8,13,14,11,4,9,10,7,\ldots$$

Now we see a complete duplication: the sequence seems to restart with the 17th number and repeat identically. Certainly, the sequence is looking less and less likely to be random. But for some very simple purposes (certainly not involving scientific simulations), it might appear sufficiently unpredictable (at least, to the casual observer, who's probably less picky about these things than we are) to be of use.

In fact, this sequence is an example of a *pseudo-random number sequence* – a sequence of numbers generated by mathematical formulæ (called a *pseudo-random number generator*, or *PRNG*), where each number in the sequence is computed from one or more previous numbers in the sequence, and where the overall result is intended to appear random.

The PRNG that generated the sequence is quite simple:

$$s_0 = 0$$
$$s_{i+1} = (13\,s_i + 5) \bmod 16$$

The sequence has a *seed* (initial value – note that we usually call the initial value in a sequence s_0, instead of s_1) of 0, and each term in the sequence is generated by multiplying the previous term by 13, then adding 5, and finally taking the remainder after dividing by 16.

We can keep computing the terms of this sequence indefinitely, and the pattern keeps repeating, as shown in figure 1. The number of iterations required before the sequence repeats is the *period* of a PRNG; here, the period is 16.

Figure 1: LCG with $s_{i+1} = (13s_i + 5) \bmod 16$, $s_0 = 0$

We can also use a different value for s_0 (i.e. a different seed); we'll get the same repeating sequence, but starting at a different point. For example, with $s_0 = 2$, we have:

$$S = 2, 15, 8, 13, 14, 11, 4, 9, 10, 7, 0, 5, 6, 3, 12, 1, 2, 15, 8, 13, 14, 11, 4, 9, 10, 7, 0, 5, 6, 3, 12, 1, \ldots$$

We might conclude that this isn't a good way to generate pseudo-random numbers, for the simple reason that the results aren't unpredictable enough. But what if our sequence formula used values other than 13, 5, and 16? It turns out that with careful selection of these values, we can get much better results.

Linear Congruential Generator

The PRNG we just looked at is an example of a *linear congruential generator*, or *LCG*. An LCG generates a sequence of pseudo-random numbers with the equation:

$$s_{i+1} = (a\,s_i + c) \bmod m$$

LCGs are included in the standard libraries of many different programming languages. In most cases, only a portion of each s_i is returned as the pseudo-random value. For example, the standard Java library generates pseudo-random integers as follows (S is the sequence maintained internally; X is the sequence of values returned):

$$s_{i+1} = (25214903917\,s_i + 11) \bmod 2^{48}$$
$$x_i = \left\lfloor \frac{s_i}{2^{16}} \right\rfloor$$

With the division by 2^{16}, the most obviously non-random aspects of the sequence (e.g. the alternating odd/even pattern, characteristic of LCGs with even moduli) are minimized. However, there are other issues with LCGs, primarily having to do with the correlation between successive LCG values. For example, in figure 2 we see what happens when we group the terms of our original sequence in pairs, and treat them as a sequence of points in two dimensions:

$$\boldsymbol{S} = (0,5),\ (6,3),\ (12,1),\ (2,15),\ (8,13),\ (14,11),\ (4,9),\ (10,7),\ \ldots$$

Not only do the points lie on a small number of straight lines, but when we view the graph as a torus (an LCG “wraps around” at the modulus value – in this case, 16), we see that we can draw a single line from (15, 0), through the point at (12, 1), and continuing (wrapping around as appropriate) through all of the other points!

With a well chosen LCG, with a long period, this effect isn't nearly as visible (especially with a small number of dimensions), but it's still present.

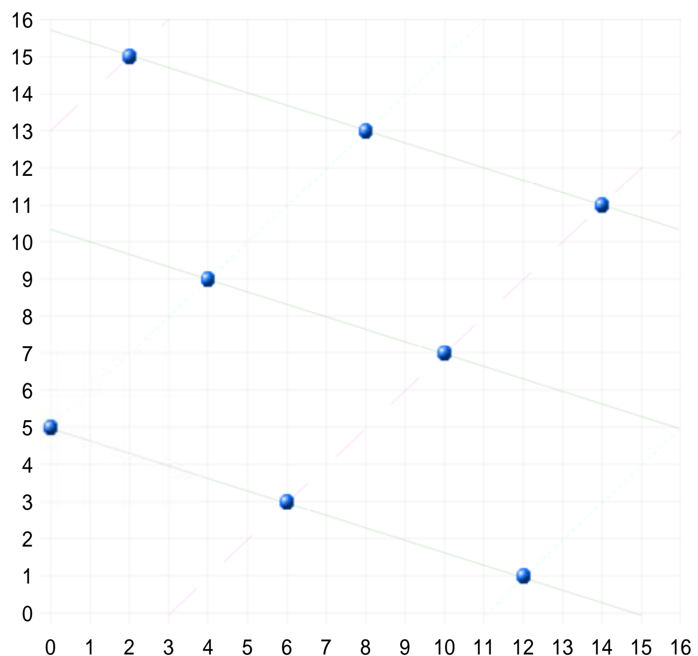

Figure 2: 4-bit LCG in Two Dimensions

Exercise 7: Another LCG

Given the LCG:

$s_0 = \text{day of the month you were born} \in \{1, 2, \ldots, 31\}$
$s_{i+1} = (5 s_i + 13) \bmod 32$

Compute the terms $s_{1,}, s_{2,} \ldots, s_{10}$.

Remember that $a \bmod b =$ the remainder when a is divided by b. For example:

$11 \bmod 4 = 3$

$37 \bmod 5 = 2$

$89 \bmod 32 = 25$

Based on your calculations, and on the parameters of this LCG:

- How random does the result appear?
- What patterns do you see?
- How many distinct values of s_i are possible?
- What is the period of the sequence S?

The Mersenne Twister and Other PRNGs

The *Mersenne twister* is a PRNG that uses a matrix linear recurrence to generate its pseudo-random sequence. It is named for the fact that key parameters of the generating functions form the exponent of a *Mersenne prime* (a prime number of the form $2^p - 1$).

Over the last decade, the Mersenne Twister has become a popular alternative to the LCGs provided in many standard libraries. It's now included as the standard PRNG in Python, Ruby, R, MatLab, and Maple. Its key advantages are a very long period (equal to the Mersenne prime $2^{19937} - 1$) and a lack of significant serial correlation.

There are many other PRNGs as well, each with its strengths and weaknesses. Statistical tests can help us assess the quality of the the generated sequences; other factors in the selection of a PRNG include ease of implementation and runtime efficiency. But in the end, the selection of a PRNG has much to do with context: What's the intended use, how much apparent randomness is required, and what's the cost of switching PRNGs?

When Pseudo-random Isn't Good Enough

Sometimes, there's no acceptable substitute for truly random numbers – at least as seed values for PRNGs. As noted above, most computers aren't very good at genuine randomness, but there are any number of real-world processes that have truly random behavior. Even better, some of these processes, while random, have well-understood statistical behavior. If we can measure one or more of these processes over time, that may give us a good source of useful random numbers.

In fact, processes such as the radioactive decay of certain elements, photon emissions by semiconductors, atmospheric or thermal noise – even human users' mouse movements and keyboard activity – serve these purposes very well. Devices using these phenomena to generate random numbers are now available; some of these are used in services that make the resulting number sequences accessible via the Internet.

Shuffling with Java: Cards

Introduction

In previous exercises, we shuffled and picked numbers for a lottery. This time, we'll use the Fisher-Yates algorithm (see appendix A) to shuffle and deal a deck of playing cards. At first, it might seem more complicated to shuffle cards, since we're dealing with the concepts of suit and rank. However, 52 distinct items are 52 distinct items, whether they're the numbers 1 through 52, or the combinations of ranks (A, 2, 3, …, King) with suits (Clubs, Diamonds, Hearts, Spades). In any event, the type of things being shuffled is irrelevant to the shuffling algorithm: we simply exchange one thing for another, repeatedly. So shuffling cards won't be much different from shuffling numbers.

The number of possible combinations of five hands of five cards each, from a 52-card deck, is several orders of magnitude larger than the number of possible combinations in a typical lottery (with the exception of keno). For this reason, we're going to use the Mersenne twister as our PRNG in these exercises, since it has a much larger number of states (i.e. longer period) than the LCG used by the `Random` class.

We're also going to switch development environments for these exercises. We used DrJava in the earlier exercises, to take advantage of the **Interactions** pane. This time, we'll use NetBeans; it doesn't have the interactive execution features, but it has many other strengths that make it well suited to a wide variety of development projects.

The Shuffle Project in NetBeans

Let's start by launching NetBeans. (NetBeans is a pure Java program, and it does a lot of housekeeping tasks when it starts up, so it can seem a bit slow in starting.) Once NetBeans has completed its startup processing, use the **File/Open Project…** menu command, navigate to where the Shuffle project is located (preferably on your own USB flash drive, but the project is probably also located on the system desktop as well), and click the **Open Project** button.

You should now see the Shuffle project in the **Project** panel, on the left-hand side of the NetBeans window. In this panel, each project is viewed as a tree; click the "**+**" or "**–**" symbols next to the project name, or one of its components, to expand or collapse that portion of the tree.

We'll be writing Java source code, so we'll need to work in **Source Packages** portion of the project. Expand the Shuffle project itself, then **Source Packages**, and finally the

`org.nm.challenge.examples.cards` package. (You might remember that a Java package usually contains one or more related classes.) All of the new Java classes we create will be in this package.

The Card Class

One of the classes we'll use has already been built for us: you should be able to see the `Card.java` file in the `org.nm.challenge.examples.cards` package; that file contains the Card class, along with some additional supporting classes.

This class encapsulates the attributes (rank and suit) of a playing card, along with the simple behavior of converting itself to a string suitable for display. It also has a `static` method called `newDeck` that returns a complete, unshuffled deck as a `Card[]` (array of `Card` objects); we'll call that method from our own code, to start with a fresh deck of cards. Finally, it has methods for sorting a `Card[]`, in either rank or suit order; we'll use one of those methods to sort the dealt hands before displaying them.

Exercise 8: The Dealer Class

What are the basic, required attributes and behaviors of the dealer in a card game? Of course, a dealer has a deck of cards. The dealer must be able to shuffle those cards, and then deal them out. Of course, dealing is done differently in different types of card games; also, what the dealer might be required to do after the deal varies by the type of game as well. For now, we'll focus just on the initial deal of some number of cards to each of some number of players.

Based on this description of the basic attributes and behaviors of a dealer, we'll create a `Dealer` class, with (at least) a `shuffle` method and a `deal` method. Creating a new class is quite easy in NetBeans:

1. Right-click on the `org.nm.challenge.examples.cards` package in the **Projects** panel, and select **New/Java Class…** from the pop-up menu.
2. Specify “Dealer” (starting with an uppercase “D”; don't type the quotes) for **Class Name**; leave the other fields unchanged.
3. Click the **Finish** button.

You should now see a NetBeans-created skeleton `Dealer` class, based on the information you provided:[1]

```
1 package org.nm.challenge.examples.cards;
2
3 public class Dealer {
4
5 }
```

Let's add to the class, starting with a `Card[]` to hold the deck of cards, and an instance of the Mersenne twister for generating random numbers. (As in the previous exercises, the code you've already written is grayed out. Don't rewrite this code; simply add to it.)

```
 1 package org.nm.challenge.examples.cards;
 2
 3 public class Dealer {
 4
 5     private MersenneTwister rng;
 6     private Card[] deck;
 7
 8     public Dealer() {
 9         rng = new MersenneTwister();
10         deck = Card.newDeck();
11     }
12
13 }
```

Even if you type all of the above correctly, you'll see some red lines under the `MersenneTwister`, in lines 5 and 9. Unlike DrJava, NetBeans analyzes code as we type, looking for problems that would prevent successful compilation. When such problems are found, they're underlined, and we can read the error message.

If you hold your mouse over one of the red lines (or over one of the error indicators in the left margin), you'll see that NetBeans is unable to find the `MersenneTwister` class. Since that class is in a different package than the one where our Dealer class is located, we have to use an `import` statement to let Java know in which package the `MersenneTwister` class is located.

We could go ahead and write the `import` statement, but let's see one of the convenient features of NetBeans instead: right-click anywhere in the white space of the file you're

---

1 The default NetBeans configuration includes boilerplate comment sections in the Java class template. In the interests of clarity, those comments have been removed from the NetBeans-generated code here.

editing, and select **Fix Imports** from the pop-up menu. The red lines should disappear (if they don't, check your spelling), and you should now see this (note the **`import`** statement in line 3, added automatically by NetBeans):

```
 1 package org.nm.challenge.examples.cards;
 2
 3 import org.apache.commons.math.random.MersenneTwister;
 4
 5 public class Dealer {
 6
 7     private MersenneTwister rng;
 8     private Card[] deck;
 9
10     public Dealer() {
11         rng = new MersenneTwister();
12         deck = Card.newDeck();
13     }
14
15 }
```

If you have other red underlines in your code, indicating more errors, try to eliminate them by checking the error message and correcting the reported condition. (The large majority of these problems are due to missing semicolons, incorrect or inconsistent spelling or letter case, and mismatched curly braces.) When you've done that, type Ctrl-S (or select the **File/Save** menu command) to save `Dealer.java`. (You don't need to specify a file name; when you typed "Dealer" for the class name, NetBeans automatically created the `Dealer.java` file for the **`Dealer`** class.) Each time you save a Java file in NetBeans, it's compiled automatically – as long as there aren't any errors.

Let's review the important points of the code so far:

- The **`package`** statement (line 1) tells the Java compiler that this class will reside in the **`org.nm.challenge.examples.cards`** package. (There's a direct relationship between package names and the directory structures for source files and compiled class files; fortunately, NetBeans handles this for us automatically.)

- The **`import`** statement (line 3) tells the Java compiler that our code will be using the **`MersenneTwister`** class, from the **`org.apache.commons.math.random`** package. (That package is part of the Apache Commons Math library, which has already been included as part of this NetBeans project.)

- As always, a class is declared with the **`class`** keyword and the class name (line 5). The set of curly braces that follows (lines 5 and 15) contains the class body.

- We've declared two private variables to hold the state data for Dealer objects: **`rng`**, which is a **`MersenneTwister`** object (line 7), and **`deck`**, which is a **`Card[]`**, or array of **`Card`** objects (line 8).

- We've defined a **`public`** constructor for the **`Dealer`** class (lines 10-13); like all constructors, this one has the same name as the class. The purpose of this constructor is to initialize the data of a **`Dealer`** object – to do that, it creates and initializes the **`rng`** variable by calling the **`MersenneTwister`** constructor, and creates and initializes the **`deck`** variable by calling the **`Card.newDeck`** method.

Now, let's add the **`shuffle`** method. Unlike the shuffle code we wrote in the lottery exercises, this code will implement a full shuffle of the deck.

```
 1 package org.nm.challenge.examples.cards;
 2 
 3 import org.apache.commons.math.random.MersenneTwister;
 4 
 5 public class Dealer {
 6 
 7     private MersenneTwister rng;
 8     private Card[] deck;
 9 
10     public Dealer() {
11         rng = new MersenneTwister();
12         deck = Card.newDeck();
13     }
14 
15     public void shuffle() {
16         for (int i = 0; i < deck.length - 1; i++) {
17             int selection = i + rng.nextInt(deck.length - i);
18             Card temp = deck[selection];
19             deck[selection] = deck[i];
20             deck[i] = temp;
21         }
22     }
23 
24 }
```

Once again, you might see red underlines, indicating that NetBeans has detected one or more problems that will prevent Java from compiling the code. Fix any such errors (requesting help as needed), and save your file.

Let's look at what the `shuffle` method is doing (lines 15-22):

- We use a `for` statement to iterate over values of `i` from 0 (inclusive) to `deck.length - 1` (exclusive). This results in the code between the curly braces (lines 16 and 21) executing `deck.length - 1` times, which will shuffle the `deck.length` elements of the `deck` array completely.

- In each iteration, we generate a pseudo-random number (line 17) between `i` (inclusive) and `deck.length` (exclusive). This is the index of our randomly selected `Card` in the `deck` array.

- In lines 18-20, we exchange element `selection` in the `deck` array with element `i`. This moves our randomly selected `Card` to its shuffled position.

Now that we've shuffled the deck of cards, we need a way to deal them out. To do this, we need to answer a couple of questions first:

- In what order will we deal the cards? Traditionally, the dealer deals one card to each player, until all players have the required number of cards; we'll follow this convention in our code.

- What's the right Java structure for holding multiple hands of cards? So far, we've been using arrays to hold multiple items of the same type, but we've only used them for one-dimensional structures. Fortunately, we can also have arrays with two (or more) dimensions; that's the approach we'll take.

  What does a two-dimensional array look like? Imagine that we have the following declaration and assignment statement, for a two-dimensional array of `Card` objects:

  ```
  Card[][] cards = new Card[4][5];
  ```

  We can visualize the elements of the array with the arrangement shown in table 4. (Can you see the pattern in the indices?)

cards[0][0]	cards[0][1]	cards[0][2]	cards[0][3]	cards[0][4]
cards[1][0]	cards[1][1]	cards[1][2]	cards[1][3]	cards[1][4]
cards[2][0]	cards[2][1]	cards[2][2]	cards[2][3]	cards[2][4]
cards[3][0]	cards[3][1]	cards[3][2]	cards[3][3]	cards[3][4]

Table 4: Two-dimensional Array

With those decisions out of the way, let's write the **deal** method.

```
1 package org.nm.challenge.examples.cards;
2
3 import org.apache.commons.math.random.MersenneTwister;
4
5 public class Dealer {
6
7     private MersenneTwister rng;
8     private Card[] deck;
9
10    public Dealer() {
11        rng = new MersenneTwister();
12        deck = Card.newDeck();
13    }
14
15    public void shuffle() {
16        for (int i = 0; i < deck.length - 1; i++) {
17            int selection = i + rng.nextInt(deck.length - i);
18            Card temp = deck[selection];
19            deck[selection] = deck[i];
20            deck[i] = temp;
21        }
22    }
23
24    public Card[][] deal(int numberOfHands, int cardsPerHand) {
25        Card[][] cards = new Card[numberOfHands][cardsPerHand];
26        for (int i = 0; i < cardsPerHand; i++) {
27            for (int j = 0; j < numberOfHands; j++) {
28                cards[j][i] = deck[i * numberOfHands + j];
29            }
30        }
31        return cards;
32    }
33
34 }
```

Do your best to fix any errors marked with red underlines, then save your file.

The **deal** method has a few things we haven't seen before, so let's review it carefully:

- The return type is **Card[][]** (line 24). As we discussed, this indicates a two-dimensional array, where each element contains a **Card**.
- This method requires two additional pieces of information to do its job. One is an **int** called **numberOfHands**; the other is an **int** called **cardsPerHand** (line 24).

These represent the number of player hands the method will deal, and the number of cards to be dealt to each hand, respectively.

- We've declared and initialized a local variable called **`cards`**, which is a two-dimensional array of **`Card`** objects (line 25). We'll put the cards we deal into this array, and return it as the result of the **`deal`** method.

- Since we're going to deal the first card to every player's hand, then the second card to every player's hand, etc., we need two different iteration loops, one nested inside the other. If this is confusing, imagine yourself as the dealer; now imagine that you don't have much experience dealing cards, so you're talking to yourself, to keep everything straight: "First card: player 1, player 2, player 3, player 4, player 5. Second card: player 1, player 2, player 3, player 4, player 5. Third card: player 1, player 2, …" Notice that you're keeping two separate counts: one for the card number, and one for the player number. Every time you move to the next card number, you start counting the players over again. That's exactly how the two **`for`** statements work (line 26-27): the first is using **`i`** as a variable to count the current card being dealt; the second is using **`j`** as a variable to count the current hand to which a card is being dealt. With **`i`**, we're counting from 0 (inclusive) to **`cardsPerHand`** (exclusive); for each of the **`i`** values, we're counting from 0 (inclusive) to **`numberOfHands`** (exclusive) with **`j`**.

- For every pair of **`i`** and **`j`** values, we put a card from **`deck`** into **`cards[j][i]`** (line 28). But how do we know which element of **`deck`** should go into each element of **`cards`**? With a little bit of thought, we notice that each time we deal one card to all of the player hands, we go through **`numberOfHands`** cards in **`deck`**. So if we multiply the card counter (**`i`**) by **`numberOfHands`**, and then add the hand counter (**`j`**), the result is the total number of cards dealt to this point – which is the index of the element in **`deck`** that we should deal.

- After we've finished filling the two-dimensional **`cards`** array with items from the **`deck`** array, we return the **`cards`** array as the result of the method (line 31).

Fix any errors that appear, and save your file. You should now have a completed **`Dealer`** class (at least, the lack of errors tells us that it's syntactically correct) – but you don't yet have a way to test it, to make sure it does what it's supposed to do. (NetBeans has built-in support for JUnit, a toolkit used for building and running tests of each of the classes in a project; however, JUnit is outside the scope of this lesson.) So the next step is to build a Java program that uses the **`Dealer`** class to deal a deck of cards, and then displays the results.

Exercise 9: Writing a Java Program that Uses the Dealer Class

In a simplistic way, our Java program will act as the host for a card game. The host brings a dealer in, and tells the dealer how many cards should be dealt, to how many players. At this point, that's where the analogy ends: our program will then turn over all the cards, so everyone can see them, and that's that – but it's enough for now.

Right-click on `org.nm.challenge.examples.cards`, under **Source Packages** in the **Projects** panel, and select **New/Java Main Class…** from the pop-up menu. This time, type "Host" in the **Class Name** field, leave the other fields unchanged, and click the **Finish** button. NetBeans creates and opens the `Host.java` file, as follows:

```
1 package org.nm.challenge.examples.cards;
2
3 public class Host {
4
5     public static void main(String[] args) {
6
7     }
8
9 }
```

There aren't really any surprises here. The most important point to remember is that a Java program class must have a `main` method, with `public` visibility, with the `void` return type (i.e. no data will be returned), and with one `String[]` parameter.

Now, let's go ahead and fill in the first part of the `main` method (along with some constants), to create a `Dealer` object, then have it shuffle and deal the cards.

```
 1 package org.nm.challenge.examples.cards;
 2
 3 public class Host {
 4
 5     private static final int NUMBER_OF_HANDS = 5;
 6     private static final int CARDS_PER_HAND = 5;
 7
 8     public static void main(String[] args) {
 9         Card[][] hands;
10         Dealer dealer = new Dealer();
11         dealer.shuffle();
12         hands = dealer.deal(NUMBER_OF_HANDS, CARDS_PER_HAND);
13     }
14
15 }
```

Fix any errors, and save your code. When you've done that, let's review the additions:

- We've declared two **`static final`** variables (lines 5-6). Remember that this combination of keywords identifies a constant – a variable whose value can't be changed after the initial assignment. In this case, one constant (**`NUMBER_OF_HANDS`**) is the number of different player hands that will be dealt; the other (**`CARDS_PER_HAND`**) is the number of cards to deal to each player.

- The **`main`** method starts by declaring the local variable **`hands`** (line 9), which we'll use to store the hands dealt by our **`Dealer`** object.

- In lines 10-11, we create and initialize the **`dealer`** variable (an object of the **`Dealer`** class type), and call its **`shuffle`** method.

- Next, we call the **`dealer.deal`** method, using the constants declared above to specify the number of player hands to deal, and the number of cards to deal to each (line 12). That method returns a two-dimensional array of **`Card`**, which we store in **`hands`**.

The last task in the exercise is to sort and display the cards in each hand. This could potentially be somewhat tricky. After all, in our lottery exercises, we saw that the **`Arrays`** class had very convenient methods for sorting the elements of numeric arrays and converting them to strings. But does the **`Array`** class know how to sort arrays of **`Card`** objects, or convert them to strings?

In general, **`Arrays.sort`** can't automatically infer the correct ordering between two objects of a custom class type. Further, while all classes have a **`toString`** method, which is called automatically by **`Arrays.toString`**, the default implementation isn't very good for display custom classes.

Fortunately, there are solutions. For sorting, a class can implement the **`Comparable`** *interface* (an interface is similar to a class, but it doesn't have implementations of its declared methods; a class implementing an interface must implement all its methods), which declares the **`compareTo`** method. In an implementation of that method, an object compares itself to another, returning a value for the ordering between the two. With that additional information, **`Arrays.sort`** can sort objects of that class type in arrays.

Similarly, a class can override the default implementation of the **`toString`** method. Any method that converts objects to strings (e.g. **`Arrays.toString`**) will automatically take advantage of this overridden method.

In fact, the Card class includes both of these features: it implements **compareTo** and overrides **toString**. Because of this, sorting and displaying the contents of a **Card[]** is as easy as sorting and displaying the contents of an **int[]**.

Now, let's add the code to sort and display the cards dealt to us by the **Dealer** object, by iterating over the hands, sorting and displaying each one in turn.

```
 1 package org.nm.challenge.examples.cards;
 2
 3 import java.util.Arrays;
 4
 5 public class Host {
 6
 7     private static final int NUMBER_OF_HANDS = 5;
 8     private static final int CARDS_PER_HAND = 5;
 9
10     public static void main(String[] args) {
11         Card[][] hands;
12         Dealer dealer = new Dealer();
13         dealer.shuffle();
14         hands = dealer.deal(NUMBER_OF_HANDS, CARDS_PER_HAND);
15         for (int i = 0; i < NUMBER_OF_HANDS; i++) {
16             Arrays.sort(hands[i]);
17             System.out.printf("Hand #%d: %s\n",
18                 i + 1, Arrays.toString(hands[i]));
19         }
20     }

}
```

With the mixture of parentheses, square brackets, and curly braces in the new code, typographic errors are all too easy to commit. Be sure to read all error messages for code underlined in red, and then save your file.

The new code uses some notation that might be confusing at first glance. But let's start at the first line of the additions, and deal with the any confusion when we get to it:

- Because we're using methods of the **Arrays** class in the new code, we need an import statement (line 3). We can either write this line by hand, or use the **Fix Imports** feature, after we've added code that refers to the **Arrays** class.

- Once again, we use the **for** statement – this time using the variable **i** to count from 0 (inclusive) to **NUMBER_OF_HANDS** (exclusive). For each value of **i**, Java will execute the statements between the curly braces on lines 13 and 17.

- In line 14, **`Arrays.sort`** is being used to sort **`hands[i]`** – but what is **`hands[i]`**? We know that **`hands`** is a two-dimensional array, and that an expression like **`hands[a][b]`** refers to a single element in the **`hands`** array. So what does **`hands[i]`** – with only one index specified – mean?

  If we look again at table 4, we remember that the first index specifies the row of the two-dimensional array, and the second specifies the column. If only the first index is specified, then we only know the row, and not the column. In fact, such an expression refers to the entire row as an array. So **`hands[i]`** refers to the array containing all of the cards in the player hand stored in row **`i`** of **`hands`**. So **`Arrays.sort(hands[i])`** sorts the cards in a single player hand. (Incidentally, there's not an equivalent for referring to an entire column of a two-dimensional array at once.)

- We use the same notation in line 16, where we call **`Arrays.toString`** to convert the **`hands[i]`** array (i.e. the array containing all of the cards in a single player hand) to a string. The result of this conversion is substituted for the second placeholder ("**`%s`**") in the first parameter of the **`System.out.printf`** method call, in lines 15-16; the first placeholder ("**`%d`**") is replaced by the current player hand counter – actually, **`i + 1`**, since we start counting with 0, but want the display to start at 1. After the substitution of values for placeholders is done, the result is written to standard output (i.e. the console display), followed by a newline character (so the cards in the next hand are written on the next line).

A NetBeans project can contain many Java program classes. Even a single executable Java archive (a ZIP file, but with a `.jar` extension, containing compiled Java files and other resources) could have multiple Java program classes. So the last thing we need to do, before running our program, is tell NetBeans that it should treat the **`Host`** class as the "main" class, the one where Java will look for the **`main`** method. To do this, follow these steps:

1. Right-click on the Shuffle project in the **Projects** panel.

2. Select **Properties** from the pop-up menu.

3. In the **Categories** panel of the **Project Properties** window, click **Run**.

4. Click the **Browse…** button to the right of the **Main Class** field.

5. If your Host class has been written correctly, `org.nm.challenge.examples.cards.Host` will appear in the list of main classes. Click on it to highlight it, and then click the **Select Main Class** button.

6. Click the **OK** button in the **Project Properties** window.

Now we're ready to run the program. Select the **Run/Run Main Project** menu command, or click the green triangle icon in the toolbar near the top of the NetBeans window.

In the Output panel at the bottom of the screen, you should see something like this (though with different cards):

```
Hand #1: [3♣, 8♢, 10♢, 10♠, J♣]
Hand #2: [4♠, 5♢, 7♡, 8♠, J♢]
Hand #3: [2♠, 4♡, 6♠, 9♣, 9♢]
Hand #4: [3♠, 7♠, 8♡, J♡, A♢]
Hand #5: [2♢, 6♢, K♣, K♢, A♡]
```

Check to make sure that each of the hands are sorted (with the ace high), and that no card appears simultaneously in more than one hand.

Run the program a few more times, and check the results. Is it doing what you expected it to do? (Remember that each time you run it, you're starting with a fresh deck of cards and shuffling it; cards that appear in hands for one run may appear in other runs as well.)

Exercise 10: Additional Questions and Tasks

I. Modify your **`Host`** class, so that it deals 4 hands of 13 cards each (a bridge deal).

II. How many combinations of cards, with five hands of five cards each, can be dealt? Assume that order of cards within a hand is not significant, but the order of the hands themselves is significant. For example,

```
Hand #1: [3♣, 8♢, 10♢, 10♠, J♣]
Hand #2: [4♠, 5♢, 7♡, 8♠, J♢]
Hand #3: [2♠, 4♡, 6♠, 9♣, 9♢]
Hand #4: [3♠, 7♠, 8♡, J♡, A♢]
Hand #5: [2♢, 6♢, K♣, K♢, A♡]
```

is different from

```
Hand #1: [4♠, 5♢, 7♡, 8♠, J♢]
Hand #2: [3♣, 8♢, 10♢, 10♠, J♣]
Hand #3: [2♠, 4♡, 6♠, 9♣, 9♢]
Hand #4: [3♠, 7♠, 8♡, J♡, A♢]
Hand #5: [2♢, 6♢, K♣, K♢, A♡]
```

even though they are identical, save for hands 1 and 2 switching places.

III. If we don't specify a seed value when creating a **`MersenneTwister`** object, it takes the current system (in elapsed milliseconds since midnight on 1 January, 1970) as a seed. There is a theoretical maximum of 2^{64} values of this timer; however, in any given 10 year period (e.g. the 10 years starting now), there will be fewer than 2^{36} distinct values. In our program, distinct shuffle and deal outcomes can only be produced with different seed values for the PRNG. Thus, no matter how many times we run the current program over the next 10 years, the maximum number of distinct outcomes will be less than 2^{36}. Based on this, and based on your answer to the previous question, do you think the results of the shuffle in the current program can be considered fair?

IV. Modify the Dealer class, so that a **`long`** or **`int[]`** value can be value can be specified as a seed in the constructor. Use the **`setSeed`** method of the MersenneTwister class, or one of the constructors that takes a seed value as a parameter, to initialize the PRNG with the specified seed.

V. Without worrying about the specific code required to do so, can you think of a way your program might be modified, so that it would start with a seed that had at least 2^{64} (and preferably more) actual possible values?

Appendix A: Fisher-Yates Shuffle

Durstenfeld's version of Fisher-Yates shuffle (aka Knuth shuffle)

1. Begin with the list of items X, containing the N items $x_{1,}\ x_{2,}\ \ldots,\ x_N$.
2. For each integer value j, starting at 1 and ending with $N-1$, do the following:
    a. Generate a random value k, which can be any one of $\{j,\ j+1,\ \ldots,\ N\}$; each of the values must be equally likely to be selected.
    b. Exchange the positions of x_k and x_j in the list. Note that it's possible that $k = j$; obviously, no exchange is needed when that's the case.
3. Stop. The list is now shuffled.

Appendix B: Fisher-Yates Shuffle for Six Items and a Six-Sided Die

Algorithm

1. Write the names of the six items to be shuffled in columns 1 through the 6, under **Position (*i*)**, in the *Initial* row.

2. Start on row 1 (i.e. the row where the number 1 appears in the **Iteration (*j*)** column).

3. Do the following for rows 1 through 5, in order:

   a. Set j equal to the row number.

   b. Throw a single die to get a random value k, repeating as necessary to get a value between j and N (in this case, 6), inclusive.

   c. Write the k value in the **Roll (*k*)** column of the current row.

   d. Copy the items from the previous row to the current row, exchanging the items in position k and position j.

4. Stop. Row 5 contains the fully shuffled items.

Working Table

		Items (*N* = Number of Items = 6)					
		Position (*i*)					
Iteration (*j*)	**Roll (*k*)**	**1**	**2**	**3**	**4**	**5**	**6**
Initial							
1							
2							
3							
4							
5							

Appendix C: Lottery Class (Lottery.java)

```
1  import java.util.Arrays;
2  import java.util.Random;
3
4  public class Lottery {
5
6      private int[] numbers;
7      private Random rng = new Random();
8
9      public Lottery(int maximum) {
10         numbers = new int[maximum];
11         for (int i = 0; i < numbers.length; i++) {
12             numbers[i] = i + 1;
13         }
14     }
15
16     private void mix(int iterations) {
17         for (int i = 0; i < iterations; i++) {
18             int shuffleIndex =
19                 i + rng.nextInt(numbers.length - i);
20             int temp = numbers[shuffleIndex];
21             numbers[shuffleIndex] = numbers[i];
22             numbers[i] = temp;
23         }
24     }
25
26     public int[] pick(int numbersToPick) {
27         int[] selection;
28         mix(numbersToPick);
29         selection = Arrays.copyOf(numbers, numbersToPick);
30         Arrays.sort(selection);
31         return selection;
32     }
33
34 }
```

Appendix D: Roadrunner Lottery Program (Roadrunner.java)

```java
 1 import java.util.Arrays;
 2 
 3 public class Roadrunner {
 4 
 5     private static final int MAXIMUM_NUMBER = 37;
 6     private static final int NUMBER_TO_PICK = 5;
 7     private static final int NUMBER_OF_TICKETS = 10;
 8 
 9     public static void main(String[] args) {
10         Lottery lotto = new Lottery(MAXIMUM_NUMBER);
11         for (int i = 1; i <= NUMBER_OF_TICKETS; i++) {
12             int[] selection = lotto.pick(NUMBER_TO_PICK);
13             String selectString = Arrays.toString(selection);
14             System.out.printf("Ticket #%d: %s\n",
15                 i, selectString);
16         }
17     }
18 
19 }
```

Appendix E: Card Class (Card.java)

```
1 package org.nm.challenge.examples.cards;
2
3 import java.util.Arrays;
4
5 public class Card implements Comparable {
6
7     private static final String TO_STRING_PATTERN = "%s%s";
8
9     private static Card[] prototypeDeck;
10
11     private Rank rank;
12     private Suit suit;
13
14     static {
15         prototypeDeck = new Card[Rank.values().length
16             * Suit.values().length];
17         int position = 0;
18         for (Suit suit : Suit.values()) {
19             for (Rank rank : Rank.values()) {
20                 prototypeDeck[position++] =
21                     new Card(rank, suit);
22             }
23         }
24     }
25
26     public Card(Rank rank, Suit suit) {
27         this.rank = rank;
28         this.suit = suit;
29     }
30
31     public Rank getRank() {
32         return this.rank;
33     }
34
35     public Suit getSuit() {
36         return this.suit;
37     }
38
39     public static Card[] newDeck() {
40         return Arrays.copyOf(prototypeDeck,
41             prototypeDeck.length);
42     }
43
```

```
44      public String toString() {
45          return String.format(TO_STRING_PATTERN,
46              this.rank, this.suit);
47      }
48
49      public int compareTo(Object o) {
50          int result = 0;
51          if (o instanceof Card) {
52              Card other = (Card) o;
53              result = getRank().compareTo(other.getRank());
54              if (result == 0) {
55                  result =
56                      getSuit().compareTo(other.getSuit());
57              }
58          }
59          else {
60              result = toString().compareTo(o.toString());
61          }
62          return result;
63      }
64
65      public enum Rank {
66
67          TWO("2"),
68          THREE("3"),
69          FOUR("4"),
70          FIVE("5"),
71          SIX("6"),
72          SEVEN("7"),
73          EIGHT("8"),
74          NINE("9"),
75          TEN("10"),
76          JACK("J"),
77          QUEEN("Q"),
78          KING("K"),
79          ACE("A");
80
81          private String display;
82
83          Rank(String display) {
84              this.display = display;
85          }
86
```

```
 86         public String toString() {
 87             return display;
 88         }
 89 
 90     }
 91 
 92     public enum Suit {
 93 
 94         CLUBS((char) 0x2663),
 95         DIAMONDS((char) 0x2662),
 96         HEARTS((char) 0x2661),
 97         SPADES((char) 0x2660);
 98 
 99         private char display;
100 
101         Suit(char display) {
102             this.display = display;
103         }
104 
105         public String toString() {
106             return Character.toString(display);
107         }
108 
109     }
110 
111 }
```

Appendix E: Card Dealer Class (Dealer.java)

```
1 package org.nm.challenge.examples.cards;
2
3 import org.apache.commons.math.random.MersenneTwister;
4
5 public class Dealer {
6
7     private MersenneTwister rng;
8     private Card[] deck;
9
10     public Dealer() {
11         rng = new MersenneTwister();
12         deck = Card.newDeck();
13     }
14
15     public void shuffle() {
16         for (int i = 0; i < deck.length - 1; i++) {
17             int selection = i + rng.nextInt(deck.length - i);
18             Card temp = deck[selection];
19             deck[selection] = deck[i];
20             deck[i] = temp;
21         }
22     }
23
24     public Card[][] deal(int numberOfHands, int cardsPerHand) {
25         Card[][] cards = new Card[numberOfHands][cardsPerHand];
26         for (int i = 0; i < cardsPerHand; i++) {
27             for (int j = 0; j < numberOfHands; j++) {
28                 cards[j][i] = deck[i * numberOfHands + j];
29             }
30         }
31         return cards;
32     }
33
34 }
```

Appendix F: Dealer Test Class (Host.java)

```java
1 package org.nm.challenge.examples.cards;
2
3 import java.util.Arrays;
4
5 public class Host {
6
7     private static final int NUMBER_OF_HANDS = 5;
8     private static final int CARDS_PER_HAND = 5;
9
10    public static void main(String[] args) {
11        Card[][] hands;
12        Dealer dealer = new Dealer();
13        dealer.shuffle();
14        hands = dealer.deal(NUMBER_OF_HANDS, CARDS_PER_HAND);
15        for (int i = 0; i < NUMBER_OF_HANDS; i++) {
16            Arrays.sort(hands[i]);
17            System.out.printf("Hand #%d: %s\n",
18                i + 1, Arrays.toString(hands[i]));
19        }
20    }
21
22 }
```