java.math.BigDecimal 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other, RoundingMode mode)` 5.0

  returns the sum, difference, product, or quotient of this big decimal and `other`. To compute the quotient, you must supply a *rounding mode*. The mode `RoundingMode.HALF_UP` is the rounding mode that you learned in school: round down the digits 0 to 4, round up the digits 5 to 9. It is appropriate for routine calculations. See the API documentation for other rounding modes.
- `int compareTo(BigDecimal other)`

  returns `0` if this big decimal equals `other`, a negative result if this big decimal is less than `other`, and a positive result otherwise.
- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x, int scale)`

  returns a big decimal whose value equals `x` or `x` / $10^{\texttt{scale}}$.

3.10 Arrays

An array is a data structure that stores a collection of values of the same type. You access each individual value through an integer *index*. For example, if `a` is an array of integers, then `a[i]` is the `i`th integer in the array.

Declare an array variable by specifying the array type—which is the element type followed by `[]`—and the array variable name. For example, here is the declaration of an array `a` of integers:

```
int[] a;
```

However, this statement only declares the variable `a`. It does not yet initialize `a` with an actual array. Use the `new` operator to create the array.

```
int[] a = new int[100];
```

This statement declares and initializes an array of 100 integers.

The array length need not be a constant: `new int[n]` creates an array of length `n`.

NOTE: You can define an array variable either as

```
int[] a;
```

or as

```
int a[];
```

Most Java programmers prefer the former style because it neatly separates the type `int[]` (integer array) from the variable name.

The array elements are *numbered from 0 to 99* (and not 1 to 100). Once the array is created, you can fill the elements in an array, for example, by using a loop:

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
   a[i] = i; // fills the array with numbers 0 to 99
```

When you create an array of numbers, all elements are initialized with zero. Arrays of `boolean` are initialized with `false`. Arrays of objects are initialized with the special value `null`, which indicates that they do not (yet) hold any objects. This can be surprising for beginners. For example,

```
String[] names = new String[10];
```

creates an array of ten strings, all of which are `null`. If you want the array to hold empty strings, you must supply them:

```
for (int i = 0; i < 10; i++) names[i] = "";
```

CAUTION: If you construct an array with 100 elements and then try to access the element `a[100]` (or any other index outside the range from 0 to 99), your program will terminate with an “array index out of bounds” exception.

To find the number of elements of an array, use *array*`.length`. For example:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Once you create an array, you cannot change its size (although you can, of course, change an individual array element). If you frequently need to expand the size of an array while your program is running, you should use a different data structure called an *array list*. (See Chapter 5 for more on array lists.)

3.10.1 The "for each" Loop

Java has a powerful looping construct that allows you to loop through each element in an array (or any other collection of elements) without having to fuss with index values.

The *enhanced* `for` loop

```
for (variable : collection) statement
```

sets the given variable to each element of the collection and then executes the statement (which, of course, may be a block). The *collection* expression must be an array or an object of a class that implements the `Iterable` interface, such as `ArrayList`. We discuss array lists in Chapter 5 and the `Iterable` interface in Chapter 9.

For example,

```
for (int element : a)
   System.out.println(element);
```

prints each element of the array `a` on a separate line.

You should read this loop as "for each `element` in `a`". The designers of the Java language considered using keywords, such as `foreach` and `in`. But this loop was a late addition to the Java language, and in the end nobody wanted to break the old code that already contained methods or variables with these names (such as `System.in`).

Of course, you could achieve the same effect with a traditional `for` loop:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

However, the "for each" loop is more concise and less error-prone, as you don't have to worry about those pesky start and end index values.

NOTE: The loop variable of the "for each" loop traverses the *elements* of the array, not the index values.

The "for each" loop is a pleasant improvement over the traditional loop if you need to process all elements in a collection. However, there are still plenty of opportunities to use the traditional `for` loop. For example, you might not want to traverse the entire collection, or you may need the index value inside the loop.

TIP: There is an even easier way to print all values of an array, using the `toString` method of the `Arrays` class. The call `Arrays.toString(a)` returns a string containing the array elements, enclosed in brackets and separated by commas, such as `"[2, 3, 5, 7, 11, 13]"`. To print the array, simply call

```
System.out.println(Arrays.toString(a));
```

3.10.2 Array Initializers and Anonymous Arrays

Java has a shortcut for creating an array object and supplying initial values at the same time. Here's an example of the syntax at work:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

Notice that you do not call `new` when you use this syntax.

You can even initialize an *anonymous array*:

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

This expression allocates a new array and fills it with the values inside the braces. It counts the number of initial values and sets the array size accordingly. You can use this syntax to reinitialize an array without creating a new variable. For example,

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

is shorthand for

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```

NOTE: It is legal to have arrays of length 0. Such an array can be useful if you write a method that computes an array result and the result happens to be empty. Construct an array of length 0 as

`new` *elementType*`[0]`

Note that an array of length 0 is not the same as `null`.

3.10.3 Array Copying

You can copy one array variable into another, but then *both variables refer to the same array*:

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```

Figure 3.14 shows the result. If you actually want to copy all values of one array into a new array, you use the `copyOf` method in the `Arrays` class:

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

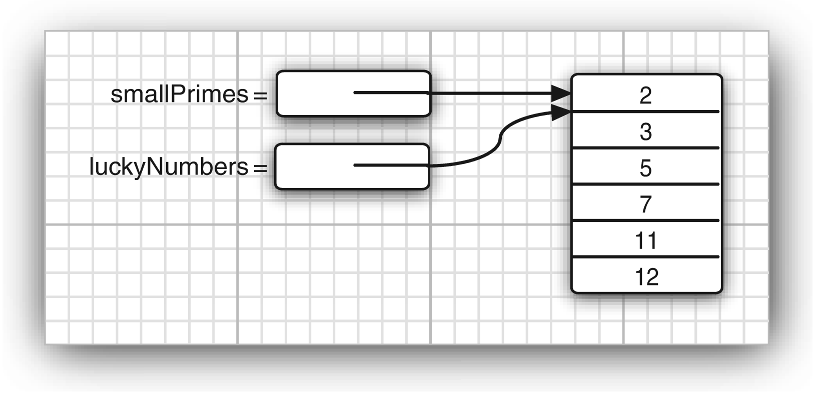

Figure 3.14 Copying an array variable

The second parameter is the length of the new array. A common use of this method is to increase the size of an array:

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

The additional elements are filled with `0` if the array contains numbers, `false` if the array contains `boolean` values. Conversely, if the length is less than the length of the original array, only the initial values are copied.

C++ NOTE: A Java array is quite different from a C++ array on the stack. It is, however, essentially the same as a pointer to an array allocated on the *heap*. That is,

```
int[] a = new int[100]; // Java
```

is not the same as

```
int a[100]; // C++
```

but rather

```
int* a = new int[100]; // C++
```

In Java, the `[]` operator is predefined to perform *bounds checking*. Furthermore, there is no pointer arithmetic—you can't increment `a` to point to the next element in the array.

3.10.4 Command-Line Parameters

You have already seen one example of a Java array repeated quite a few times. Every Java program has a `main` method with a `String[] args` parameter. This parameter indicates that the `main` method receives an array of strings—namely, the arguments specified on the command line.

For example, consider this program:

```
public class Message
{
   public static void main(String[] args)
   {
      if (args.length == 0 || args[0].equals("-h"))
         System.out.print("Hello,");
      else if (args[0].equals("-g"))
         System.out.print("Goodbye,");
      // print the other command-line arguments
      for (int i = 1; i < args.length; i++)
         System.out.print(" " + args[i]);
      System.out.println("!");
   }
}
```

If the program is called as

```
java Message -g cruel world
```

then the `args` array has the following contents:

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

The program prints the message

```
Goodbye, cruel world!
```

C++ NOTE: In the `main` method of a Java program, the name of the program is not stored in the `args` array. For example, when you start up a program as

```
java Message -h world
```

from the command line, then `args[0]` will be `"-h"` and not `"Message"` or `"java"`.

3.10.5 Array Sorting

To sort an array of numbers, you can use one of the `sort` methods in the `Arrays` class:

```
int[] a = new int[10000];
 . . .
Arrays.sort(a)
```

This method uses a tuned version of the QuickSort algorithm that is claimed to be very efficient on most data sets. The `Arrays` class provides several other convenience methods for arrays that are included in the API notes at the end of this section.

The program in Listing 3.7 puts arrays to work. This program draws a random combination of numbers for a lottery game. For example, if you play a "choose 6 numbers from 49" lottery, the program might print this:

```
Bet the following combination. It'll make you rich!
   4
   7
   8
   19
   30
   44
```

To select such a random set of numbers, we first fill an array `numbers` with the values `1, 2, . . ., n`:

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
   numbers[i] = i + 1;
```

A second array holds the numbers to be drawn:

```
int[] result = new int[k];
```

Now we draw `k` numbers. The `Math.random` method returns a random floating-point number that is between `0` (inclusive) and `1` (exclusive). By multiplying the result with `n`, we obtain a random number between `0` and `n - 1`.

```
int r = (int) (Math.random() * n);
```

We set the `i`th result to be the number at that index. Initially, that is just `r + 1`, but as you'll see presently, the contents of the `numbers` array are changed after each draw.

```
result[i] = numbers[r];
```

Now we must be sure never to draw that number again—all lottery numbers must be distinct. Therefore, we overwrite `numbers[r]` with the *last* number in the array and reduce `n` by `1`.

```
numbers[r] = numbers[n - 1];
n--;
```

The point is that in each draw we pick an *index*, not the actual value. The index points into an array that contains the values that have not yet been drawn.

After drawing `k` lottery numbers, we sort the `result` array for a more pleasing output:

```
Arrays.sort(result);
for (int r : result)
   System.out.println(r);
```

Listing 3.7 `LotteryDrawing/LotteryDrawing.java`

```
import java.util.*;

/**
 * This program demonstrates array manipulation.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryDrawing
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("How many numbers do you need to draw? ");
      int k = in.nextInt();

      System.out.print("What is the highest number you can draw? ");
      int n = in.nextInt();

      // fill an array with numbers 1 2 3 . . . n
      int[] numbers = new int[n];
      for (int i = 0; i < numbers.length; i++)
         numbers[i] = i + 1;

      // draw k numbers and put them into a second array
      int[] result = new int[k];
      for (int i = 0; i < result.length; i++)
      {
         // make a random index between 0 and n - 1
         int r = (int) (Math.random() * n);

         // pick the element at the random location
         result[i] = numbers[r];

         // move the last element into the random location
         numbers[r] = numbers[n - 1];
         n--;
      }

      // print the sorted array
      Arrays.sort(result);
      System.out.println("Bet the following combination. It'll make you rich!");
      for (int r : result)
         System.out.println(r);
   }
}
```

java.util.Arrays 1.2

- `static String toString(`*type*`[] a)` 5.0

  returns a string with the elements of `a`, enclosed in brackets and delimited by commas.

Parameters:	`a`	An array of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

- `static `*type*`[] copyOf(`*type*`[] a, int length)` 6
- `static `*type*`[] copyOfRange(`*type*`[] a, int start, int end)` 6

  returns an array of the same type as `a`, of length either `length` or `end - start`, filled with the values of `a`.

Parameters:	`a`	An array of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.
	`start`	The starting index (inclusive).
	`end`	The ending index (exclusive). May be larger than `a.length`, in which case the result is padded with `0` or `false` values.
	`length`	The length of the copy. If `length` is larger than `a.length`, the result is padded with `0` or `false` values. Otherwise, only the initial `length` values are copied.

- `static void sort(`*type*`[] a)`

  sorts the array, using a tuned QuickSort algorithm.

Parameters:	`a`	An array of type `int`, `long`, `short`, `char`, `byte`, `float`, or `double`.

(Continues)

`java.util.Arrays` 1.2 *(Continued)*

- `static int binarySearch(`*type*`[] a, `*type*` v)`
- `static int binarySearch(`*type*`[] a, int start, int end, `*type*` v)` 6

  Uses the binary search algorithm to search for the value `v`. If it is found, its index is returned. Otherwise, a negative value `r` is returned; `-r - 1` is the spot at which `v` should be inserted to keep `a` sorted.

Parameters:	`a`	a *sorted* array of type `int`, `long`, `short`, `char`, `byte`, `float`, or `double`.
	`start`	The starting index (inclusive).
	`end`	The ending index (exclusive).
	`v`	A value of the same type as the elements of `a`.

- `static void fill(`*type*`[] a, `*type*` v)`

  Sets all elements of the array to `v`.

Parameters:	`a`	An array of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.
	`v`	A value of the same type as the elements of `a`.

- `static boolean equals(`*type*`[] a, `*type*`[] b)`

  Returns `true` if the arrays have the same length and if the elements in corresponding indexes match.

Parameters:	`a, b`	Arrays of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

3.10.6 Multidimensional Arrays

Multidimensional arrays use more than one index to access array elements. They are used for tables and other more complex arrangements. You can safely skip this section until you have a need for this storage mechanism.

Suppose you want to make a table of numbers that shows how much an investment of $10,000 will grow under different interest rate scenarios in which interest is paid annually and reinvested (Table 3.8).

You can store this information in a two-dimensional array (matrix), which we call `balances`.

Declaring a two-dimensional array in Java is simple enough. For example:

```
double[][] balances;
```

Table 3.8 Growth of an Investment at Different Interest Rates

10%	11%	12%	13%	14%	15%
10,000.00	10,000.00	10,000.00	10,000.00	10,000.00	10,000.00
11,000.00	11,100.00	11,200.00	11,300.00	11,400.00	11,500.00
12,100.00	12,321.00	12,544.00	12,769.00	12,996.00	13,225.00
13,310.00	13,676.31	14,049.28	14,428.97	14,815.44	15,208.75
14,641.00	15,180.70	15,735.19	16,304.74	16,889.60	17,490.06
16,105.10	16,850.58	17,623.42	18,424.35	19,254.15	20,113.57
17,715.61	18,704.15	19,738.23	20,819.52	21,949.73	23,130.61
19,487.17	20,761.60	22,106.81	23,526.05	25,022.69	26,600.20
21,435.89	23,045.38	24,759.63	26,584.44	28,525.86	30,590.23
23,579.48	25,580.37	27,730.79	30,040.42	32,519.49	35,178.76

You cannot use the array until you initialize it. In this case, you can do the initialization as follows:

```
balances = new double[NYEARS][NRATES];
```

In other cases, if you know the array elements, you can use a shorthand notation for initializing a multidimensional array without a call to `new`. For example:

```
int[][] magicSquare =
   {
      {16, 3, 2, 13},
      {5, 10, 11, 8},
      {9, 6, 7, 12},
      {4, 15, 14, 1}
   };
```

Once the array is initialized, you can access individual elements by supplying two pairs of brackets—for example, `balances[i][j]`.

The example program stores a one-dimensional array `interest` of interest rates and a two-dimensional array `balances` of account balances, one for each year and interest rate. We initialize the first row of the array with the initial balance:

```
for (int j = 0; j < balances[0].length; j++)
   balances[0][j] = 10000;
```

Then we compute the other rows, as follows:

```
for (int i = 1; i < balances.length; i++)
{
   for (int j = 0; j < balances[i].length; j++)
   {
      double oldBalance = balances[i - 1][j];
      double interest = . . .;
      balances[i][j] = oldBalance + interest;
   }
}
```

Listing 3.8 shows the full program.

NOTE: A "for each" loop does not automatically loop through all elements in a two-dimensional array. Instead, it loops through the rows, which are themselves one-dimensional arrays. To visit all elements of a two-dimensional array `a`, nest two loops, like this:

```
for (double[] row : a)
   for (double value : row)
      do something with value
```

TIP: To print out a quick-and-dirty list of the elements of a two-dimensional array, call

```
System.out.println(Arrays.deepToString(a));
```

The output is formatted like this:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

Listing 3.8 `CompoundInterest/CompoundInterest.java`

```
/**
 * This program shows how to store tabular data in a 2D array.
 * @version 1.40 2004-02-10
 * @author Cay Horstmann
 */
public class CompoundInterest
{
   public static void main(String[] args)
   {
      final double STARTRATE = 10;
      final int NRATES = 6;
      final int NYEARS = 10;

      // set interest rates to 10 . . . 15%
      double[] interestRate = new double[NRATES];
      for (int j = 0; j < interestRate.length; j++)
         interestRate[j] = (STARTRATE + j) / 100.0;

      double[][] balances = new double[NYEARS][NRATES];

      // set initial balances to 10000
      for (int j = 0; j < balances[0].length; j++)
         balances[0][j] = 10000;

      // compute interest for future years
      for (int i = 1; i < balances.length; i++)
      {
         for (int j = 0; j < balances[i].length; j++)
         {
            // get last year's balances from previous row
            double oldBalance = balances[i - 1][j];

            // compute interest
            double interest = oldBalance * interestRate[j];

            // compute this year's balances
            balances[i][j] = oldBalance + interest;
         }
      }

      // print one row of interest rates
      for (int j = 0; j < interestRate.length; j++)
         System.out.printf("%9.0f%%", 100 * interestRate[j]);

      System.out.println();

      // print balance table
      for (double[] row : balances)
      {
         // print table row
         for (double b : row)
            System.out.printf("%10.2f", b);

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

3.10.7 Ragged Arrays

So far, what you have seen is not too different from other programming languages. But there is actually something subtle going on behind the scenes that you can sometimes turn to your advantage: Java has *no* multidimensional arrays at all, only one-dimensional arrays. Multidimensional arrays are faked as "arrays of arrays."

For example, the `balances` array in the preceding example is actually an array that contains ten elements, each of which is an array of six floating-point numbers (Figure 3.15).

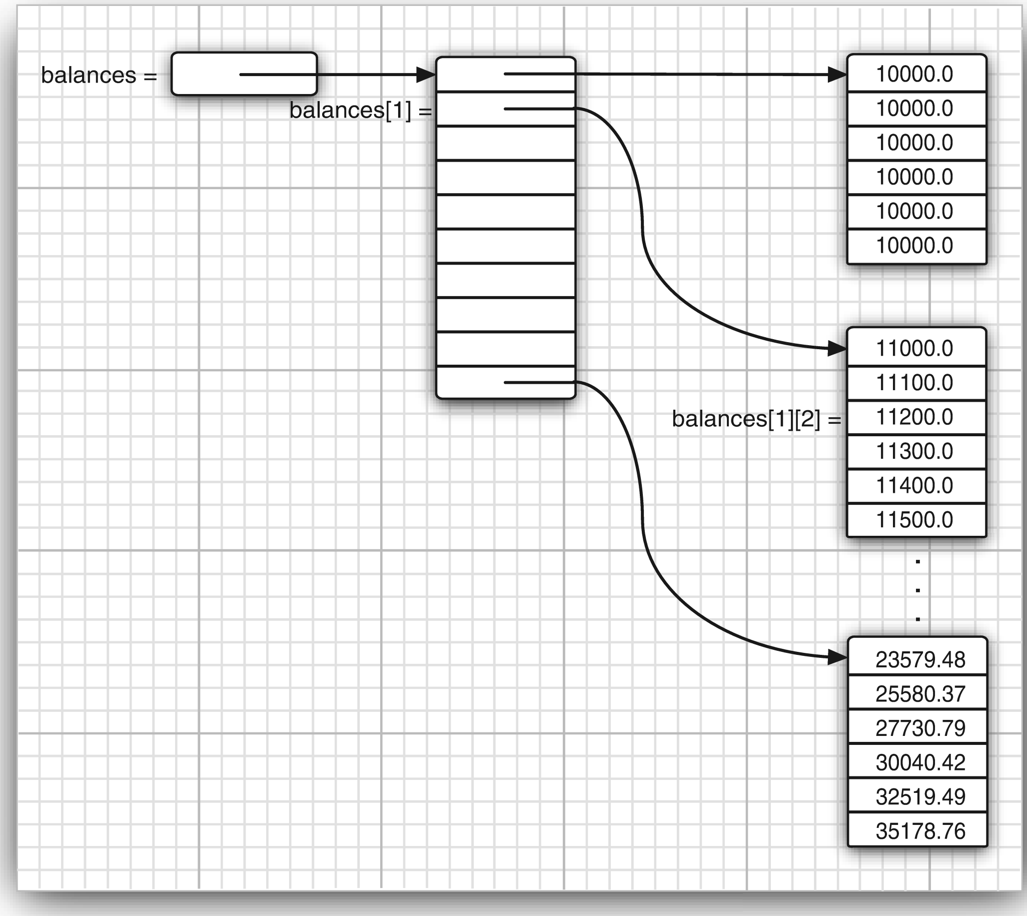

Figure 3.15 A two-dimensional array

The expression `balances[i]` refers to the `i`th subarray—that is, the `i`th row of the table. It is itself an array, and `balances[i][j]` refers to the `j`th element of that array.

Since rows of arrays are individually accessible, you can actually swap them!

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

It is also easy to make "ragged" arrays—that is, arrays in which different rows have different lengths. Here is the standard example. Let us make an array in which the element at row `i` and column `j` equals the number of possible outcomes of a "choose `j` numbers from `i` numbers" lottery.

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10  5 1
1  6 15 20 15 6 1
```

As `j` can never be larger than `i`, the matrix is triangular. The `i`th row has `i + 1` elements. (We allow choosing 0 elements; there is one way to make such a choice.) To build this ragged array, first allocate the array holding the rows.

```
int[][] odds = new int[NMAX + 1][];
```

Next, allocate the rows.

```
for (int n = 0; n <= NMAX; n++)
   odds[n] = new int[n + 1];
```

Now that the array is allocated, we can access the elements in the normal way, provided we do not overstep the bounds.

```
for (int n = 0; n < odds.length; n++)
   for (int k = 0; k < odds[n].length; k++)
   {
      // compute lotteryOdds
      . . .
      odds[n][k] = lotteryOdds;
   }
```

Listing 3.9 gives the complete program.

C++ NOTE: In C++, the Java declaration

```
double[][] balances = new double[10][6]; // Java
```

is not the same as

```
double balances[10][6]; // C++
```

or even

```
double (*balances)[6] = new double[10][6]; // C++
```

Instead, an array of ten pointers is allocated:

```
double** balances = new double*[10]; // C++
```

Then, each element in the pointer array is filled with an array of six numbers:

```
for (i = 0; i < 10; i++)
   balances[i] = new double[6];
```

Mercifully, this loop is automatic when you ask for a `new double[10][6]`. When you want ragged arrays, you allocate the row arrays separately.

Listing 3.9 `LotteryArray/LotteryArray.java`

```java
/**
 * This program demonstrates a triangular array.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryArray
{
   public static void main(String[] args)
   {
      final int NMAX = 10;

      // allocate triangular array
      int[][] odds = new int[NMAX + 1][];
      for (int n = 0; n <= NMAX; n++)
         odds[n] = new int[n + 1];

      // fill triangular array
      for (int n = 0; n < odds.length; n++)
         for (int k = 0; k < odds[n].length; k++)
         {
            /*
             * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
             */
            int lotteryOdds = 1;
            for (int i = 1; i <= k; i++)
               lotteryOdds = lotteryOdds * (n - i + 1) / i;

            odds[n][k] = lotteryOdds;
         }

      // print triangular array
      for (int[] row : odds)
      {
         for (int odd : row)
            System.out.printf("%4d", odd);
         System.out.println();
      }
   }
}
```

You have now seen the fundamental programming structures of the Java language. The next chapter covers object-oriented programming in Java.