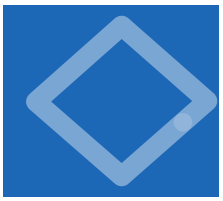
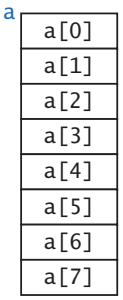


1.4 Arrays

In this section, we consider a fundamental programming construct known as the *array*. The primary purpose of an array is to facilitate storing and manipulating large quantities of data. Arrays play an essential role in many data processing tasks. They also correspond to vectors and matrices, which are widely used in science and in scientific programming. We will consider basic properties of array processing in Java, with many examples illustrating why they are useful.

An array stores a sequence of values that are all of the same type. Processing such a set of values is very common. We might have exam scores, stock prices, nucleotides in a DNA strand, or characters in a book. Each of these examples involve a large number of values that are all of the same type.

We want not only to store values but also directly access each individual value. The method that we use to refer to individual values in an array is numbering and then *indexing* them. If we have N values, we think of them as being numbered from 0 to $N-1$. Then, we can unambiguously specify one of them by referring to the ith value for any value of i from 0 to $N-1$. To refer to the `i`th value in an array `a`, we use the notation `a[i]`, pronounced *a sub i*. This Java construct is known as a *one-dimensional* array.



An array

The one-dimensional array is our first example in this book of a *data structure* (a method for organizing data). We also consider in this section a more complicated data structure known as *a two-dimensional array*. Data structures play an essential role in modern programming—Chapter 4 is largely devoted to the topic.

Typically, when we have a large amount of data to process, we first put all of the data into one or more arrays. Then we use array indexing to refer to individual values and to process the data. We consider such applications when we discuss data input in Section 1.5 and in the case study that is the subject of Section 1.6. In this section, we expose the basic properties of arrays by considering examples where our programs first populate arrays with computed values from experimental studies and then process them.

Arrays in Java Making an array in a Java program involves three distinct steps:

- Declare the array name and type.
- Create the array.
- Initialize the array values.

To declare the array, you need to specify a name and the type of data it will contain. To create it, you need to specify its size (the number of values). For example, the following code makes an array of `N` numbers of type `double`, all initialized to `0.0`:

```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

The first statement is the array declaration. It is just like a declaration of a variable of the corresponding primitive type except for the square brackets following the type name, which specify that we are declaring an array. The second statement creates the array. This action is unnecessary for variables of a primitive type (so we have not seen a similar action before), but it is needed for all other types of data in Java (see SECTION 3.1). In the code in this book, we normally keep the array length in an integer variable `N`, but any integer-valued expression will do. The `for` statement initializes the `N` array values. We refer to each value by putting its index in brackets after the array name. This code sets all of the array entries to the value `0.0`.

When you begin to write code that uses an array, you must be sure that your code declares, creates, and initializes it. Omitting one of these steps is a common programming mistake. For economy in code, we often take advantage of Java's default array initialization convention and combine all three steps into a single statement. For example, the following statement is equivalent to the code above:

```
double[] a = new double[N];
```

The code to the left of the equal sign constitutes the declaration; the code to the right constitutes the creation. The `for` loop is unnecessary in this case because the default initial value of variables of type `double` in a Java array is `0.0`, but it would be required if a nonzero value were desired. The default initial value is zero for all numbers and `false` for type `boolean`. For `String` and other non-primitive types, the default is the value `null`, which you will learn about in CHAPTER 3.

After declaring and creating an array, you can refer to any individual value anywhere you would use a variable name in a program by enclosing an integer in-

dex in braces after the array name. We refer to the `i`th item with the code `a[i]`. The explicit initialization code shown earlier is an example of such a use. The obvious advantage of using arrays is to avoid explicitly naming each variable individually. Using an array index is virtually the same as appending the index to the array name: for example, if we wanted to process eight variables of type `double`, we could declare each of them individually with the declaration

```
double a0, a1, a2, a3, a4, a5, a6, a7;
```

and then refer to them as `a0`, `a1` and so forth instead of declaring them with `double[] a = new double[8]` and referring to them as `a[0]`, `a[1]`, and so forth. But naming dozens of individual variables in this way would be cumbersome and naming millions is untenable.

As an example of code that uses arrays, consider using arrays to represent *vectors*. We consider vectors in detail in SECTION 3.3; for the moment, think of a vector as a sequence of real numbers. The *dot product* of two vectors (of the same length) is the sum of the products of their corresponding components. The dot product of two vectors that are represented as one-dimensional arrays `x[]` and `y[]` that are each of length 3 is the expression `x[0]*y[0] + x[1]*y[1] + x[2]*y[2]`. If we represent the two vectors as one-dimensional arrays `x[]` and `y[]` that are each of length `N` and of type `double`, the dot product is easy to compute:

```
double sum = 0.0;
for (int i = 0; i < N; i++)
   sum += x[i]*y[i];
```

i	x[i]	y[i]	x[i]*y[i]	sum
				0
0	.30	.50	.15	.15
1	.60	.10	.06	.21
2	.10	.40	.04	.25
				.25

Trace of dot product computation

The simplicity of coding such computations makes the use of arrays the natural choice for all kinds of applications. (Note that when we use the notation `x[]`, we are referring to the whole array, as opposed to `x[i]`, which is a reference to the `i`th entry.)

The accompanying table has many examples of array-processing code, and we will consider even more examples later in the book, because arrays play a central role in processing data in many applications. Before considering more sophisticated examples, we describe a number of important characteristics of programming with arrays.

create an array with random values	`double[] a = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < N; i++)` `  System.out.println(a[i]);`

find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < N; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < N; i++)`
	`  sum += a[i];`
	`double average = sum / N;`

copy to another array	`double[] b = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  b[i] = a[i];`

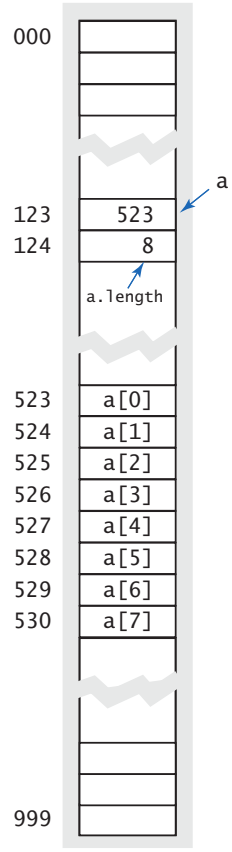
reverse the elements within an array	`for (int i = 0; i < N/2; i++)`
	`{`
	`  double temp = b[i];`
	`  b[i] = b[N-1-i];`
	`  b[N-i-1] = temp;`
	`}`

Typical array-processing code (for arrays of N `double` *values)*

Zero-based indexing. We always refer to the first element of an array as `a[0]`, the second as `a[1]`, and so forth. It might seem more natural to you to refer to the first element as `a[1]`, the second value as `a[2]`, and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages. Misunderstanding this convention often leads to *off-by one-errors* that are notoriously difficult to avoid and debug, so be careful!

Array length. Once we create an array, its size is fixed. The reason that we need to explicitly create arrays at runtime is that the Java compiler cannot know how much space to reserve for the array at compile time (as it can for primitive-type values). Our convention is to keep the size of the array in a variable `N` whose value can be set at runtime (usually it is the value of a command-line argument). Java's standard mechanism is to allow a program to refer to the length of an array `a[]` with the code `a.length`; we normally use `N` to create the array, or set the value of `N` to `a.length`. Note that the last element of an array is always `a[a.length-1]`.

Memory representation. Arrays are fundamental data structures in that they have a direct correspondence with memory systems on virtually all computers. The elements of an array are stored consecutively in memory, so that it is easy to quickly access any array value. Indeed, we can view memory itself as a giant array. On modern computers, memory is implemented in hardware as a sequence of indexed memory locations that each can be quickly accessed with an appropriate index. When referring to computer memory, we normally refer to a location's index as its *address*. It is convenient to think of the name of the array—say, `a`—as storing the memory address of the first element of the array `a[0]`. For the purposes of illustration, suppose that the computer's memory is organized as 1,000 values, with addresses from 000 to 999. (This simplified model ignores the fact that array elements can occupy differing amounts of memory depending on their type, but you can ignore such details for the moment.) Now, suppose that an array of eight elements is stored in memory locations 523 through 530. In such a situation, Java would store the memory address (index) of the first array value somewhere else in memory, along with the array length. We refer to the address as a *pointer* and think of it as *pointing to* the referenced memory location. When we specify `a[i]`, the compiler generates code that accesses the desired value by adding the index `i` to the memory address of the array `a[]`. For example, the Java code `a[4]` would generate machine code that finds the value at memory location `523 + 4 = 527`. Accessing element `i` of an array is an efficient operation because it simply requires adding two integers and then referencing memory—just two elementary operations. Extending the model to handle different-sized array elements just involves multiplying the index by the element size before adding to the array address.



Memory representation

Memory allocation. When you use `new` to create an array, Java reserves space in memory for it. This process is called *memory allocation*. The same process is required for all variables that you use in a program. We call attention to it now because it is your responsibility to use `new` to allocate memory for an array before accessing any of its elements. If you fail to adhere to this rule, you will get a compile-time *uninitialized variable* error. Java automatically initializes all of the values in an array when it is created. You should remember that the time required to create an array is proportional to its length.

Bounds checking. As already indicated, you must be careful when programming with arrays. It is your responsibility to use legal indices when accessing an array element. If you have created an array of size `N` and use an index whose value is less than `0` or greater than `N-1`, your program will terminate with an `ArrayIndexOutOfBounds` run-time exception. (In many programming languages, such *buffer overflow* conditions are not checked by the system. Such unchecked errors can and do lead to debugging nightmares, but it is also not uncommon for such an error to go unnoticed and remain in a finished program. You might be surprised to know that such a mistake can be exploited by a hacker to take control of a system, even your personal computer, to spread viruses, steal personal information, or wreak other malicious havoc.) The error messages provided by Java may seem annoying to you at first, but they are small price to pay to have a more secure program.

Setting array values at compile time. When we have a small number of literal values that we want to keep in array, we can declare and initialize it by listing the values between curly braces, separated by commas. For example, we might use the following code in a program that processes playing cards.

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

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

After creating the two arrays, we can use them to print out a random card name, such as `Queen of Clubs`, as follows:

```
int i = (int) (Math.random() * rank.length);
int j = (int) (Math.random() * suit.length);
System.out.println(rank[i] + " of " + suit[j]);
```

This code uses the idiom introduced in SECTION 1.2 to generate random indices and then uses the indices to pick strings out of the arrays. Whenever the values of all array entries are known at compile time (and the size of the array is not too large) it makes sense to use this method of initializing the array—just put all the values in braces on the right hand side of an assignment in the array declaration. Doing so implies array creation, so the `new` keyword is not needed.

Setting array values at runtime. A more typical situation is when we wish to compute the values to be stored in an array. In this case, we can use array names with indices in the same way we use variable names on the left side of assignment statements. For example, we might use the following code to initialize an array of size 52 that represents a deck of playing cards, using the two arrays just defined:

```
String[] deck = new String[suit.length * rank.length];
for (int i = 0; i < suit.length; i++)
   for (int j = 0; j < rank.length; j++)
      deck[rank.length*i + j] = rank[i] + " of " + suit[j];
```

After this code has been executed, if you were to print out the contents of deck in order from `deck[0]` through `deck[51]` using `System.out.println()`, you would get the sequence

```
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
...
Ace of Hearts
Ace of Spades
```

Exchange. Frequently, we wish to exchange two values in an array. Continuing our example with playing cards, the following code exchanges the cards at position `i` and `j` using the same idiom that we traced as our first example of the use of assignment statements in SECTION 1.2:

```
String t = deck[i];
deck[i] = deck[j];
deck[j] = t;
```

When we use this code, we are assured that we are perhaps changing the *order* of the values in the array but not the *set* of values in the array. When `i` and `j` are equal, the array is unchanged. When `i` and `j` are not equal, the values `a[i]` and `a[j]` are found in different places in the array. For example, if we were to use this code with `i` equal to `1` and `j` equal to `4` in the `deck` array of the previous example, it would leave `3 of Clubs` in `deck[1]` and `2 of Diamonds` in `deck[4]`.

Shuffle. The following code shuffles our deck of cards:

```
int N = deck.length;
for (int i = 0; i < N; i++)
{
   int r = i + (int) (Math.random() * (N-i));
   String t = deck[i];
   deck[i] = deck[r];
   deck[r] = t;
}
```

Proceeding from left to right, we pick a random card from `deck[i]` through `deck[N-1]` (each card equally likely) and exchange it with `deck[i]`. This code is more sophisticated than it might seem: First, we ensure that the cards in the deck after the shuffle are the same as the cards in the deck before the shuffle by using the exchange idiom. Second, we ensure that the shuffle is random by choosing uniformly from the cards not yet chosen.

Sampling without replacement. In many situations, we want to draw a random sample from a set such that each member of the set appears at most once in the sample. Drawing numbered ping-pong balls from a basket for a lottery is an example of this kind of sample, as is dealing a hand from a deck of cards. `Sample` (PROGRAM 1.4.1) illustrates how to sample, using the basic operation underlying shuffling. It takes command-line arguments `M` and `N` and creates a *permutation* of size `N` (a rearrangement of the integers from `0` to `N-1`) whose first `M` entries com-

i	r	perm														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	9	9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15
1	5	9	5	2	3	4	1	6	7	8	0	10	11	12	13	14	15
2	13	9	5	13	3	4	1	6	7	8	0	10	11	12	2	14	15
3	5	9	5	13	1	4	3	6	7	8	0	10	11	12	2	14	15
4	11	9	5	13	1	11	3	6	7	8	0	10	4	12	2	14	15
5	8	9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15
		9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15

Trace of `java Sample 6 16`

Program 1.4.1 Sampling without replacement

```
public class Sample
{
   public static void main(String[] args)
   {  // Print a random sample of M integers
      // from 0 ... N-1 (no duplicates).
      int M = Integer.parseInt(args[0]);
      int N = Integer.parseInt(args[1]);
      int[] perm = new int[N];

      // Initialize perm[].
      for (int j = 0; j < N; j++)
          perm[j] = j;

      // Take sample.
      for (int i = 0; i < M; i++)
      {  // Exchange perm[i] with a random element to its right.
         int r = i + (int) (Math.random() * (N-i));
         int t = perm[r];
         perm[r] = perm[i];
         perm[i] = t;
      }

      // Print sample.
      for (int i = 0; i < M; i++)
         System.out.print(perm[i] + " ");
      System.out.println();
   }
}
```

M	*sample size*
N	*range*
perm[]	*permutation of 0 to N-1*

This program takes two command-line arguments M *and* N *and produces a sample of* M *of the integers from* 0 *to* N-1*. This process is useful, not just in state and local lotteries, but in scientific applications of all sorts. If the first argument is equal to the second, the result is a random permutation of the integers from* 0 *to* N-1*. If the first argument is greater than the second, the program will terminate with an* ArrayOutOfBounds *exception.*

```
% java Sample 6 16
9 5 13 1 11 8

% java Sample 10 1000
656 488 298 534 811 97 813 156 424 109

% java Sample 20 20
6 12 9 8 13 19 0 2 4 5 18 1 14 16 17 3 7 11 10 15
```

prise a random sample. The accompanying trace of the contents of the `perm[]` array at the end of each iteration of the main loop (for a run where the values of M and N are 6 and 16, respectively) illustrates the process.

If the values of `r` are chosen such that each value in the given range is equally likely, then `perm[0]` through `perm[M-1]` are a random sample at the end of the process (even though some elements might move multiple times) because each element in the sample is chosen by taking each item not yet sampled, with equal probability for each choice. One important reason to explicitly compute the permutation is that we can use it to print out a random sample of *any* array by using the elements of the permutation as indices into the array. Doing so is often an attractive alternative to actually rearranging the array because it may need to be in order for some other reason (for instance, a company might wish to draw a random sample from a list of customers that is kept in alphabetical order). To see how this trick works, suppose that we wish to draw a random poker hand from our `deck[]` array, constructed as just described. We use the code in `Sample` with $N = 52$ and $M = 5$ and replace `perm[i]` with `deck[perm[i]]` in the `System.out.print()` statement (and change it to `println()`), resulting in output such as the following:

```
3 of Clubs
Jack of Hearts
6 of Spades
Ace of Clubs
10 of Diamonds
```

Sampling like this is widely used as the basis for statistical studies in polling, scientific research, and many other applications, whenever we want to draw conclusions about a large population by analyzing a small random sample.

Precomputed values. One simple application of arrays is to save values that you have computed, for later use. As an example, suppose that you are writing a program that performs calculations using small values of the harmonic numbers (see PROGRAM 1.3.5). An efficient approach is to save the values in an array, as follows:

```
double[] H = new double[N];
for (int i = 1; i < N; i++)
   H[i] = H[i-1] + 1.0/i;
```

Then you can just use the code `H[i]` to refer to any of the values. Precomputing values in this way is an example of a *space-time tradeoff*: by investing in space (to save

the values) we save time (since we do not need to recompute them). This method is not effective if we need values for huge N, but it is very effective if we need values for small N many different times.

Simplifying repetitive code. As an example of another simple application of arrays, consider the following code fragment, which prints out the name of a month given its number (1 for January, 2 for February, and so forth):

```
if      (m ==  1) System.out.println("Jan");
else if (m ==  2) System.out.println("Feb");
else if (m ==  3) System.out.println("Mar");
else if (m ==  4) System.out.println("Apr");
else if (m ==  5) System.out.println("May");
else if (m ==  6) System.out.println("Jun");
else if (m ==  7) System.out.println("Jul");
else if (m ==  8) System.out.println("Aug");
else if (m ==  9) System.out.println("Sep");
else if (m == 10) System.out.println("Oct");
else if (m == 11) System.out.println("Nov");
else if (m == 12) System.out.println("Dec");
```

We could also use a `switch` statement, but a much more compact alternative is to use a `String` array consisting of the names of each month:

```
String[] months =
{
   "", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
       "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
System.out.println(months[m]);
```

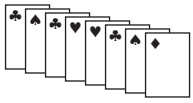
This technique would be especially useful if you needed to access the name of a month by its number in several different places in your program. Note that we intentionally waste one slot in the array (element `0`) to make `months[1]` correspond to January, as required.

Assignments and equality tests. Suppose that you have created the two arrays `a[]` and `b[]`. What does it mean to assign one to the other with the code `a = b;`? Similarly, what does it mean to test whether the two arrays are equal with the code `(a == b)`? The answers to these questions may not be what you first assume, but if you think about the array memory representation, you will see that Java's interpretation

of these operations makes sense: An assignment makes the names `a` and `b` refer to the same array. The alternative would be to have an implied loop that assigns each value in `b` to the corresponding value in `a`. Similarly, an equality test checks whether the two names refer to the same array. The alternative would be to have an implied loop that tests whether each value in one array is equal to the corresponding value in the other array. In both cases, the implementation in Java is very simple: it just performs the standard operation as if the array name were a variable whose value is the memory address of the array. Note that there are many other operations that you might want to perform on arrays: for example, it would be nice in some applications to say `a = a + b` and have it mean "add the corresponding element in `b[]` to each element in `a[]`," but that statement is not legal in Java. Instead, we write an explicit loop to perform all the additions. We will consider in detail Java's mechanism for satisfying such higher-level programming needs in SECTION 3.2. In typical applications, we use this mechanism, so we rarely need to use Java's assignments and equality tests with arrays.

WITH THESE BASIC DEFINITIONS AND EXAMPLES out of the way, we can now consider two applications that both address interesting classical problems and illustrate the fundamental importance of arrays in efficient computation. In both cases, the idea of using data to index into an array plays a central role and enables a computation that would not otherwise be feasible.

Coupon collector Suppose that you have a shuffled deck of cards and you turn them face up, one by one. How many cards do you need to turn up before you have seen one of each suit? How many cards do you need to turn up before seeing one of each value? These are examples of the famous *coupon collector* problem. In general, suppose that a trading card company issues trading cards with N different possible cards: how many do you have to collect before you have all N possibilities, assuming that each possibility is equally likely for each card that you collect?



Coupon collection

Coupon collecting is no toy problem. For example, it is very often the case that scientists want to know whether a sequence that arises in nature has the same characteristics as a random sequence. If so, that fact might be of interest; if not, further investigation may be warranted to look for patterns that might be of importance. For example, such tests are used by scientists to decide which parts of genomes are worth studying. One effective test for whether a sequence is truly random is

Program 1.4.2 Coupon collector simulation

```
public class CouponCollector
{
   public static void main(String[] args)
   {  // Generate random values in (0..N] until finding each one.
      int N = Integer.parseInt(args[0]);
      boolean[] found = new boolean[N];
      int cardcnt = 0, valcnt = 0;
      while (valcnt < N)
      {  // Generate another value.
         int val = (int) (Math.random() * N);
         cardcnt++;
         if (!found[val])
         {
            valcnt++;
            found[val] = true;
         }
      }  // N different values found.
      System.out.println(cardcnt);
   }
}
```

N	*range*
cardcnt	*values generated*
valcnt	*different values found*
found[]	*table of found values*

This program simulates coupon collection by taking a command-line argument N *and generating random numbers between* 0 *and* N-1 *until getting every possible value.*

```
% java CouponCollector 1000
6583
% java CouponCollector 1000
6477
% java CouponCollector 1000000
12782673
```

the *coupon collector test*: compare the number of elements that need to be examined before all values are found against the corresponding number for a uniformly random sequence. `CouponCollector` (PROGRAM 1.4.2) is an example program that simulates this process and illustrates the utility of arrays. It takes the value of *N* from the command line and generates a sequence of random integer values between 0

and $N-1$ using the code `(int) (Math.random() * N)` (see PROGRAM 1.2.5). Each value represents a card: for each card, we want to know if we have seen that value before. To maintain that knowledge, we use an array `found[]`, which uses the card value as an index: `found[i]` is `true` if we have seen a card with value `i` and `false` if we have not. When we get a new card that is represented by the integer `val`, we check whether we have seen its value before simply by accessing `found[val]`. The computation consists of keeping count of the number of distinct values seen and the number of cards generated and printing the latter when the former gets to `N`.

val	found 0	1	2	3	4	5	valcnt	cardcnt
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

Trace for a typical run of `java CouponCollector 6`

As usual, the best way to understand a program is to consider a trace of the values of its variables for a typical run. It is easy to add code to `CouponCollector` that produces a trace that gives the values of the variables at the end of the `while` loop for a typical run. In the accompanying figure, we use `F` for the value `false` and `T` for the value `true` to make the trace easier to follow. Tracing programs that use large arrays can be a challenge: when you have an array of size N in your program, it represents N variables, so you have to list them all. Tracing programs that use `Math.random()` also can be a challenge because you get a different trace every time you run the program. Accordingly, we check relationships among variables carefully. Here, note that `valcnt` always is equal to the number of `true` values in `found[]`.

Without arrays, we could not contemplate simulating the coupon collector process for huge N; with arrays it is easy to do so. We will see many examples of such processes throughout the book.

Sieve of Eratosthenes Prime numbers play an important role in mathematics and computation, including cryptography. A *prime number* is an integer greater than one whose only positive divisors are one and itself. The prime counting function $\pi(N)$ is the number of primes less than or equal to N. For example, $\pi(25) = 9$ since the first nine primes are 2, 3, 5, 7, 11, 13, 17, 19, and 23. This function plays a central role in number theory.

Program 1.4.3 Sieve of Eratosthenes

```
public class PrimeSieve
{
   public static void main(String[] args)
   {  // Print the number of primes <= N.
      int N = Integer.parseInt(args[0]);
      boolean[] isPrime = new boolean[N+1];
      for (int i = 2; i <= N; i++)
         isPrime[i] = true;

      for (int i = 2; i <= N/i; i++)
      {  if (isPrime[i])
         {  // Mark multiples of i as nonprime.
            for (int j = i; j <= N/i; j++)
               isPrime[i * j] = false;
         }
      }

      // Count the primes.
      int primes = 0;
      for (int i = 2; i <= N; i++)
         if (isPrime[i]) primes++;
      System.out.println(primes);
   }
}
```

N	*argument*
isPrime[i]	*is* i *prime?*
primes	*prime counter*

This program takes a command-line argument N *and computes the number of primes less than or equal to* N*. To do so, it computes an array of boolean values with* isPrime[i] *set to* true *if* i *is prime, and to* false *otherwise. First, it sets to* true *all array elements in order to indicate that no numbers are initially known to be nonprime. Then it sets to* false *array elements corresponding to indices that are known to be nonprime (multiples of known primes). If* a[i] *is still* true *after all multiples of smaller primes have been set to* false*, then we know* i *to be prime. The termination test in the second* for *loop is* i <= N/i *instead of the naive* i <= N *because any number with no factor less than* N/i *has no factor greater than* N/i*, so we do not have to look for such factors. This improvement makes it possible to run the program for large* N*.*

```
% java PrimeSieve 25
9
% java PrimeSieve 100
25
% java PrimeSieve 1000000000
50847534
```

i	isPrime																							
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
3	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
5	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F
	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

Trace of `java PrimeSieve 25`

One approach to counting primes is to use a program like `Factors` (PROGRAM 1.3.9). Specifically, we could modify the code in `Factors` to set a boolean value to be `true` if a given number is prime and `false` otherwise (instead of printing out factors), then enclose that code in a loop that increments a counter for each prime number. This approach is effective for small N, but becomes too slow as N grows.

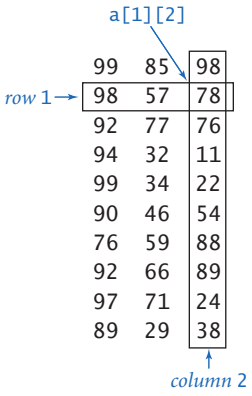
`PrimeSieve` (PROGRAM 1.4.3) takes a command-line integer `N` and computes the prime count using a technique known as the *Sieve of Eratosthenes*. The program uses a `boolean` array `isPrime[]` to record which integers are prime. The goal is to set `isPrime[i]` to `true` if `i` is prime, and to `false` otherwise. The sieve works as follows: Initially, set all array elements to `true`, indicating that no factors of any integer have yet been found. Then, repeat the following steps as long as `i <= N/i`:

- Find the next smallest `i` for which no factors have been found.
- Leave `isPrime[i]` as `true` since `i` has no smaller factors.
- Set the `isPrime[]` entries for all multiples of `i` to be `false`.

When the nested `for` loop ends, we have set the `isPrime[]` entries for all nonprimes to be `false` and have left the `isPrime[]` entries for all primes as `true`. With one more pass through the array, we can count the number of primes less than or equal to `N`. As usual, it is easy to add code to print a trace. For programs such as `PrimeSieve`, you have to be a bit careful—it contains a nested `for-if-for`, so you have to pay attention to the braces in order to put the print code in the correct place. Note that we stop when `i > N/i`, just as we did for `Factors`.

With `PrimeSieve`, we can compute $\pi(N)$ for large N, limited primarily by the maximum array size allowed by Java. This is another example of a space-time tradeoff. Programs like `PrimeSieve` play an important role in helping mathematicians to develop the theory of numbers, which has many important applications.

Two-dimensional arrays In many applications, a convenient way to store information is to use a table of numbers organized in a rectangular table and refer to *rows* and *columns* in the table. For example, a teacher might need to maintain a table with a row corresponding to each student and a column corresponding to each assignment, a scientist might need to maintain a table of experimental data with rows corresponding to experiments and columns corresponding to various outcomes, or a programmer might want to prepare an image for display by setting a table of pixels to various grayscale values or colors.



Anatomy of a two-dimensional array

The mathematical abstraction corresponding to such tables is a *matrix*; the corresponding Java construct is a *two-dimensional array*. You are likely to have already encountered many applications of matrices and two-dimensional arrays, and you will certainly encounter many others in science, in engineering, and in computing applications, as we will demonstrate with examples throughout this book. As with vectors and one-dimensional arrays, many of the most important applications involve processing large amounts of data, and we defer considering those applications until we consider input and output, in SECTION 1.5.

Extending Java array constructs to handle two-dimensional arrays is straightforward. To refer to the element in row `i` and column `j` of a two-dimensional array `a[][]`, we use the notation `a[i][j]`; to declare a two-dimensional array, we add another pair of brackets; and to create the array, we specify the number of rows followed by the number of columns after the type name (both within brackets), as follows:

```
double[][] a = new double[M][N];
```

We refer to such an array as an *M*-by-*N* array. By convention, the first dimension is the number of rows and the second is the number of columns. As with one-dimensional arrays, Java initializes all entries in arrays of numbers to zero and in arrays of boolean values to `false`.

Initialization. Default initialization of two-dimensional arrays is useful because it masks more code than for one-dimensional arrays. The following code is equivalent to the single-line create-and-initialize idiom that we just considered:

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
{  // Initialize the ith row.
   for (int j = 0; j < N; j++)
      a[i][j] = 0.0;
}
```

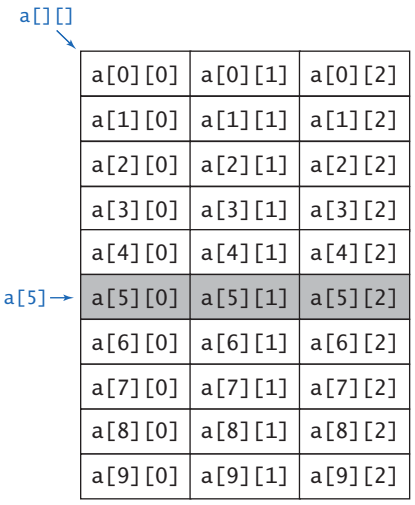
This code is superfluous when initializing to zero, but the nested `for` loops are needed to initialize to some other value(s). As you will see, this code is a model for the code that we use to access or modify each element of a two-dimensional array.

Output. We use nested `for` loops for many array-processing operations. For example, to print an M-by-N array in the familiar tabular format, we would use the following code

```
for (int i = 0; i < M; i++)
{  // Print the ith row.
   for (int j = 0; j < N; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

regardless of the array elements' type. If desired, we could add code to embellish the output with row and column numbers (see EXERCISE 1.4.6), but Java programmers typically tabulate arrays with row numbers running top to bottom from 0 and column number running left to right from 0. Generally, we also do so and do not bother to use labels.

Memory representation. Java represents a two-dimensional array as an array of arrays. A matrix with M rows and N columns is actually an array of length M, each entry of which is an array of length N. In a two-dimensional Java array `a[][]`, we can use the code `a[i]` to refer to the `i`th row (which is a one-dimensional array), but we have no corresponding way to refer to a column.



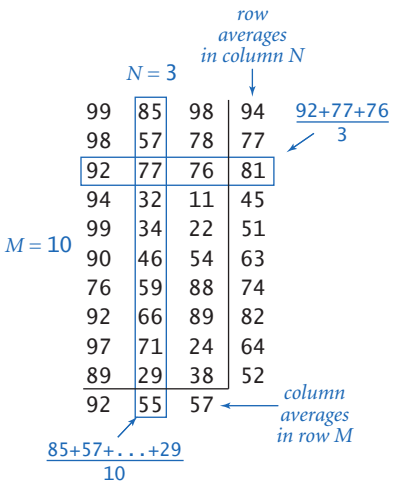
A 10-by-3 array

Setting values at compile time. The Java method for initializing an array of values at compile time follows immediately from the representation. A two-dimensional array is an array of rows, each row initialized as a one-dimensional array. To initialize a two-dimensional array, we enclose in braces a list of terms to initialize the rows, separated by commas. Each term in the list is itself a list: the values for the array elements in the row, enclosed in braces and separated by commas.

```
int[][] a =
{
   { 99, 85, 98,  0 },
   { 98, 57, 78,  0 },
   { 92, 77, 76,  0 },
   { 94, 32, 11,  0 },
   { 99, 34, 22,  0 },
   { 90, 46, 54,  0 },
   { 76, 59, 88,  0 },
   { 92, 66, 89,  0 },
   { 97, 71, 24,  0 },
   { 89, 29, 38,  0 },
   {  0,  0,  0,  0 }
};
```

Compile-time initialization of a two-dimensional array

Spreadsheets. One familiar use of arrays is a *spreadsheet* for maintaining a table of numbers. For example, a teacher with M students and N test grades for each student might maintain an $(M+1)$-by-$(N+1)$ array, reserving the last column for each student's average grade and the last row for the average test grades. Even though we typically do such computations within specialized applications, it is worthwhile to study the underlying code as an introduction to array processing. To compute the average grade for each student (average values for each row), sum the entries for each row and divide by N. The row-by-row order in which this code processes the matrix



Compute row averages

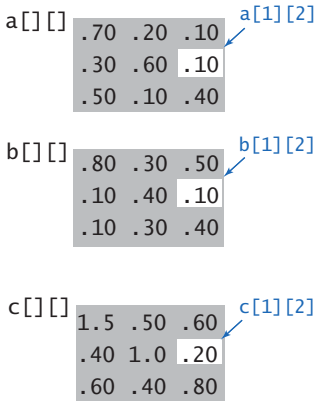
```
for (int i = 0; i < M; i++)
{  // Compute average for row i
   double sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j];
   a[i][N] = (int) Math.round(sum/N);
}
```

Compute column averages

```
for (int j = 0; j < N; j++)
{  // Compute average for column j
   double sum = 0.0;
   for (int i = 0; i < M; i++)
      sum += a[i][j];
   a[M][j] = (int) Math.round(sum/M);
}
```

Typical spreadsheet calculations

entries is known as *row-major* order. Similarly, to compute the average test grade (average values for each column), sum the entries for each column and divide by *M*. The column-by-column order in which this code processes the matrix entries is known as *column-major* order.



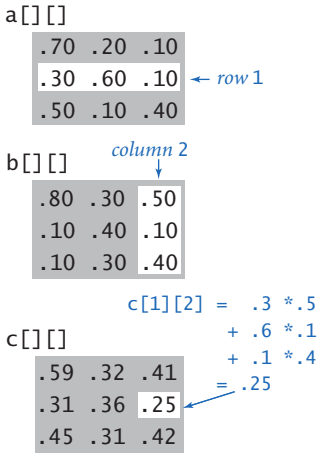
Matrix addition

Matrix operations. Typical applications in science and engineering involve representing matrices as two-dimensional arrays and then implementing various mathematical operations with matrix operands. Again, even though such processing is often done within specialized applications, it is worthwhile for you to understand the underlying computation. For example, we can *add* two *N*-by-*N* matrices as follows:

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      c[i][j] = a[i][j] + b[i][j];
```

Similarly, we can *multiply* two matrices. You may have learned matrix multiplication, but if you do not recall or are not familiar with it, the Java code below for square matrices is essentially the same as the mathematical definition. Each entry `c[i][j]` in the product of `a[]` and `b[]` is computed by taking the dot product of row `i` of `a[]` with column `j` of `b[]`.

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      // Compute dot product of row i and column j.
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k]*b[k][j];
   }
}
```



Matrix multiplication

The definition extends to matrices that are not necessarily square (see Exercise 1.4.17).

Special cases of matrix multiplication. Two special cases of matrix multiplication are important. These special cases occur when one of the dimensions of one of the matrices is 1, so it may be viewed as a vector. We have *matrix-vector multiplication*, where we multiply an M-by-N matrix by a *column vector* (an N-by-1 matrix) to get an M-by-1 column vector result (each entry in the result is the dot product of the corresponding row in the matrix with the operand vector). The second case is *vector-matrix multiplication*, where we multiply a *row vector* (a 1-by-M matrix) by an M-by-N matrix to get a 1-by-N row vector result (each entry in the result is the dot product of the operand vector with the corresponding column in the matrix). These operations provide a succinct way to express numerous matrix calculations. For example, the row-average computation for such a spreadsheet with M rows and N columns is equivalent to a matrix-vector multiplication where the column vector has M entries all equal to $1/M$. Similarly, the column-average computation in such a spreadsheet is equivalent to a vector-matrix multiplication where the row vector has N entries all equal to $1/N$. We return to vector-matrix multiplication in the context of an important application at the end of this chapter.

Matrix-vector multiplication `a[][]*x[] = b[]`

```
for (int i = 0; i < M; i++)
{  // Dot product of row i and x[].
   for (int j = 0; j < N; j++)
      b[i] += a[i][j]*x[j];
}
```

```
a[][]                        b[]
[99  85  98]                 [94]
[98  57  78]                 [77]
[92  77  76]     x[]         [81]
[94  32  11]                 [45]
[99  34  22]     [.33]       [51]  ← row averages
[90  46  54]     [.33]       [63]
[76  59  88]     [.33]       [74]
[92  66  89]                 [82]
[97  71  24]                 [64]
[89  29  38]                 [52]
```

Vector-matrix multiplication `y[]*a[][] = c[]`

```
for (int j = 0; j < N; j++)
{  // Dot product of y[] and column j.
   for (int i = 0; i < M; i++)
      c[j] += y[i]*a[i][j];
}
```

```
y[]  [ .1 .1 .1 .1 .1 .1 .1 .1 .1 .1 ]

a[][] [99  85  98]
      [98  57  78]
      [92  77  76]
      [94  32  11]
      [99  34  22]
      [90  46  54]
      [76  59  88]
      [92  66  89]
      [97  71  24]
      [89  29  38]

c[]   [92  55  57]  ← column averages
```

Matrix-vector and vector-matrix multiplication

Ragged arrays. There is actually no requirement that all rows in a two-dimensional array have the same length—an array with rows of nonuniform length is known as a *ragged array* (see Exercise 1.4.32 for an example application). The possibility of ragged arrays creates the need for more care in crafting array-processing code. For example, this code prints the contents of a ragged array:

```
for (int i = 0; i < a.length; i++)
{
   for (int j = 0; j < a[i].length; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

This code tests your understanding of Java arrays, so you should take the time to study it. In this book, we normally use square or rectangular arrays, whose dimension is given by a variable `M` or `N`. Code that uses `a[i].length` in this way is a clear signal to you that an array is ragged.

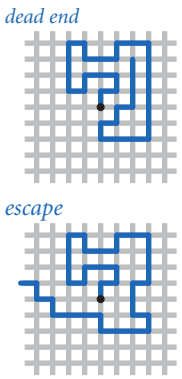
Multidimensional arrays. The same notation extends to allow us to write code using arrays that have any number of dimensions. For instance, we can declare and initialize a three-dimensional array with the code

```
double[][][] a = new double[N][N][N];
```

and then refer to an entry with code like `a[i][j][k]`, and so forth.

TWO-DIMENSIONAL ARRAYS PROVIDE A NATURAL REPRESENTATION for matrices, which are omnipresent in science, mathematics, and engineering. They also provide a natural way to organize large amounts of data, a key factor in spreadsheets and many other computing applications. Through Cartesian coordinates, two- and three-dimensional arrays also provide the basis for a models of the physical world. We consider their use in all three arenas throughout this book.

Example: self-avoiding random walks Suppose that you leave your dog in the middle of a large city whose streets form a familiar grid pattern. We assume that there are N north-south streets and N east-west streets all regularly spaced and fully intersecting in a pattern known as a *lattice*. Trying to escape the city, the dog makes a random choice of which way to go at each intersection, but knows by scent to avoid visiting any place previously visited. But it is possible for the dog to get stuck in a dead end where there is no choice but to revisit some intersection. What is the chance that this will happen? This amusing problem is a simple example of a famous model known as the *self-avoiding random walk*, which has important scientific applications in the study of polymers and in statistical mechanics, among many others. For example, you can see



Self-avoiding walks

that this process models a chain of material growing a bit at a time, until no growth is possible. To better understand such processes, scientists seek to understand the properties of self-avoiding walks.

The dog's escape probability is certainly dependent on the size of the city. In a tiny 5-by-5 city, it is easy to convince yourself that the dog is certain to escape. But what are the chances of escape when the city is large? We are also interested in other parameters. For example, how long is the dog's path, on the average? How often does the dog come within one block of a previous position other than the one just left, on the average? How often does the dog come within one block of escaping? These sorts of properties are important in the various applications just mentioned.

`SelfAvoidingWalk` (PROGRAM 1.4.4) is a simulation of this situation that uses a two-dimensional `boolean` array, where each entry represents an intersection. The value `true` indicates that the dog has visited the intersection; `false` indicates that the dog has not visited the intersection. The path starts in the center and takes random steps to places not yet visited until getting stuck or escaping at a boundary. For simplicity, the code is written so that if a random choice is made to go to a spot that has already been visited, it takes no action, trusting that some subsequent random choice will find a new place (which is assured because the code explicitly tests for a dead end and leaves the loop in that case).

Note that the code depends on Java initializing all of the array entries to `false` for each experiment. It also exhibits an important programming technique where we code the loop exit test in the `while` statement as a *guard* against an illegal statement in the body of the loop. In this case, the `while` loop continuation test serves as a guard against an out-of-bounds array access within the loop. This corresponds to checking whether the dog has escaped. Within the loop, a successful dead-end test results in a `break` out of the loop.

As you can see from the sample runs, the unfortunate truth is that your dog is nearly certain to get trapped in a dead end in a large city. If you are interested in learning more about self-avoiding walks, you can find several suggestions in the exercises. For example, the dog is virtually certain to escape in the three-dimensional version of the problem. While this is an intuitive result that is confirmed by our tests, the development of a mathematical model that explains the behavior of self-avoiding walks is a famous open problem: despite extensive research, no one knows a succinct mathematical expression for the escape probability, the average length of the path, or any other important parameter.

Program 1.4.4 Self-avoiding random walks

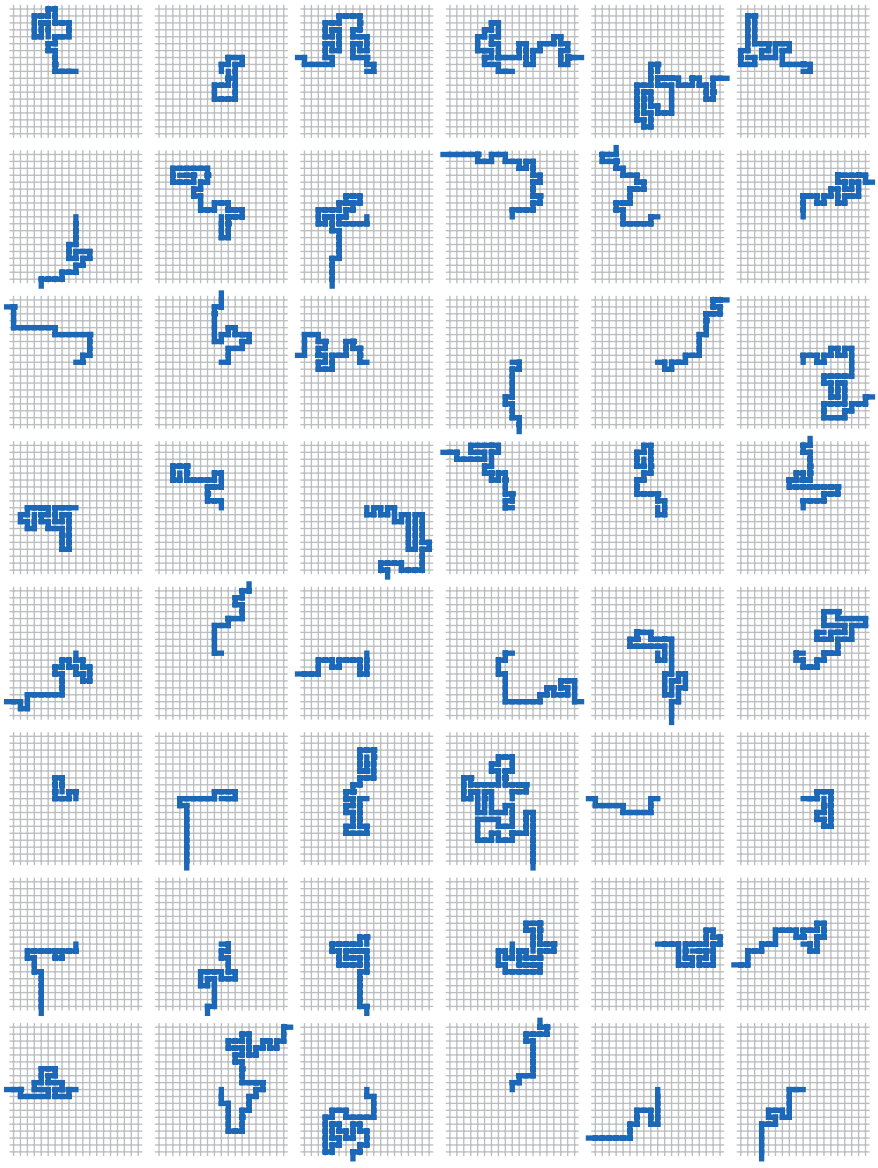
```
public class SelfAvoidingWalk
{
   public static void main(String[] args)
   {  // Do T random self-avoiding walks
      //   in an N-by-N lattice
      int N = Integer.parseInt(args[0]);
      int T = Integer.parseInt(args[1]);
      int deadEnds = 0;
      for (int t = 0; t < T; t++)
      {
         boolean[][] a = new boolean[N][N];
         int x = N/2, y = N/2;
         while (x > 0 && x < N-1 && y > 0 && y < N-1)
         {  // Check for dead end and make a random move.
            a[x][y] = true;
            if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1])
            {  deadEnds++; break;  }
            double r = Math.random();
            if      (r < 0.25) { if (!a[x+1][y]) x++; }
            else if (r < 0.50) { if (!a[x-1][y]) x--; }
            else if (r < 0.75) { if (!a[x][y+1]) y++; }
            else if (r < 1.00) { if (!a[x][y-1]) y--; }
         }
      }
      System.out.println(100*deadEnds/T + "% dead ends");
   }
}
```

N	*lattice size*
T	*number of trials*
deadEnds	*trials resulting in a dead end*
a[][]	*intersections visited*
x, y	*current position*
r	*random number in (0, 1)*

This program takes command-line arguments N *and* T *and computes* T *self-avoiding walks in an* N*-by-*N *lattice. For each walk, it creates a boolean array, starts the walk in the center, and continues until either a dead end or a boundary is reached. The result of the computation is the percentage of dead ends. As usual, increasing the number of experiments increases the precision of the results.*

```
% java SelfAvoidingWalk 5 100
0% dead ends
% java SelfAvoidingWalk 20 100
36% dead ends
% java SelfAvoidingWalk 40 100
80% dead ends
% java SelfAvoidingWalk 80 100
98% dead ends
% java SelfAvoidingWalk 160 100
100% dead ends
```

```
% java SelfAvoidingWalk 5 1000
0% dead ends
% java SelfAvoidingWalk 20 1000
32% dead ends
% java SelfAvoidingWalk 40 1000
70% dead ends
% java SelfAvoidingWalk 80 1000
95% dead ends
% java SelfAvoidingWalk 160 1000
100% dead ends
```



Self-avoiding random walks in a 21-by-21 grid

Summary Arrays are the fourth basic element (after assignments, conditionals, and loops) found in virtually every programming language, completing our coverage of basic Java constructs. As you have seen with the sample programs that we have presented, you can write programs that can solve all sorts of problems using just these constructs.

Arrays are prominent in many of the programs that we consider, and the basic operations that we have discussed here will serve you well in addressing many programming tasks. When you are not using arrays explicitly (and you are sure to be doing so frequently), you will be using them implicitly, because all computers have a memory that is conceptually equivalent to an indexed array.

The fundamental ingredient that arrays add to our programs is a potentially huge increase in the size of a program's *state*. The state of a program can be defined as the information you need to know to understand what a program is doing. In a program without arrays, if you know the values of the variables and which statement is the next to be executed, you can normally determine what the program will do next. When we trace a program, we are essentially tracking its state. When a program uses arrays, however, there can be too huge a number of values (each of which might be changed in each statement) for us to effectively track them all. This difference makes writing programs with arrays more of a challenge than writing programs without them.

Arrays directly represent vectors and matrices, so they are of direct use in computations associated with many basic problems in science and engineering. Arrays also provide a succinct notation for manipulating a potentially huge amount of data in a uniform way, so they play a critical role in any application that involves processing large amounts of data, as you will see throughout this book.

Q&A

Q. Some Java programmers use `int a[]` instead of `int[] a` to declare arrays. What's the difference?

A. In Java, both are legal and equivalent. The former is how arrays are declared in C. The latter is the preferred style in Java since the type of the variable `int[]` more clearly indicates that it is an *array* of integers.

Q. Why do array indices start at `0` instead of `1`?

A. This convention originated with machine-language programming, where the address of an array element would be computed by adding the index to the address of the beginning of an array. Starting indices at 1 would entail either a waste of space at the beginning of the array or a waste of time to subtract the 1.

Q. What happens if I use a negative number to index an array?

A. The same thing as when you use an index that is too big. Whenever a program attempts to index an array with an index that is not between zero and the array length minus one, Java will issue an `ArrayIndexOutOfBoundsException` and terminate the program.

Q. What happens when I compare two arrays with `(a == b)`?

A. The expression evaluates to `true` only if `a[]` and `b[]` refer to the same array, not if they have the same sequence of elements. Unfortunately, this is rarely what you want.

Q. If `a[]` is an array, why does `System.out.println(a)` print out a hexadecimal integer, like `@f62373` , instead of the elements of the array?

A. Good question. It is printing out the memory address of the array, which, unfortunately, is rarely what you want.

Q. What other pitfalls should I watch out for when using arrays?

A. It is very important to remember that Java *always* initializes arrays when you create them, so that *creating an array takes time proportional to the size of the array*.

Exercises

1.4.1 Write a program that declares and initializes an array `a[]` of size `1000` and accesses `a[1000]`. Does your program compile? What happens when you run it?

1.4.2 Describe and explain what happens when you try to compile a program with the following statement:

```
int N = 1000;
int[] a = new int[N*N*N*N];
```

1.4.3 Given two vectors of length `N` that are represented with one-dimensional arrays, write a code fragment that computes the *Euclidean distance* between them (the square root of the sums of the squares of the differences between corresponding entries).

1.4.4 Write a code fragment that reverses the order of a one-dimensional array `a[]` of `String` values. Do not create another array to hold the result. *Hint*: Use the code in the text for exchanging two elements.

1.4.5 What is wrong with the following code fragment?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

Solution. It does not allocate memory for `a[]` with `new`. This code results in a `variable a might not have been initialized` compile-time error.

1.4.6 Write a code fragment that prints the contents of a two-dimensional boolean array, using * to represent `true` and a space to represent `false`. Include row and column numbers.

1.4.7 What does the following code fragment print?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(a[i]);
```

1.4.8 What values does the following code put in the array `a[]`?

```
int N = 10;
int[] a = new int[N];
a[0] = 1;
a[1] = 1;
for (int i = 2; i < N; i++)
   a[i] = a[i-1] + a[i-2];
```

1.4.9 What does the following code fragment print?

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a == b);
```

1.4.10 Write a program `Deal` that takes an command-line argument `N` and prints `N` poker hands (five cards each) from a shuffled deck, separated by blank lines.

1.4.11 Write code fragments to create a two-dimensional array `b[][]` that is a copy of an existing two-dimensional array `a[][]`, under each of the following assumptions:

a. `a[][]` is square

b. `a[][]` is rectangular

c. `a[][]` may be ragged

Your solution to *b* should work for *a*, and your solution to *c* should work for both *b* and *a*, but your code should get progressively more complicated.

1.4.12 Write a code fragment to print the *transposition* (rows and columns changed) of a square two-dimensional array. For the example spreadsheet array in the text, you code would print the following:

```
99  98  92  94  99  90  76  92  97  89
85  57  77  32  34  46  59  66  71  29
98  78  76  11  22  54  88  89  24  38
```

1.4.13 Write a code fragment to transpose a square two-dimensional array *in place* without creating a second array.

1.4.14 Write a program that takes an integer *N* from the command line and creates an *N*-by-*N* boolean array `a[][]` such that `a[i][j]` is `true` if `i` and `j` are relatively prime (have no common factors), and `false` otherwise. Use your solution to EXERCISE 1.4.6 to print the array. *Hint*: Use sieving.

1.4.15 Write a program that computes the product of two square matrices of boolean values, using the *or* operation instead of + and the *and* operation instead of *.

1.4.16 Modify the spreadsheet code fragment in the text to compute a *weighted* average of the rows, where the weights of each test score are in a one-dimensional array `weights[]`. For example, to assign the last of the three tests in our example to be twice the weight of the others, you would use

```
double[] weights = { .25, .25, .50 };
```

Note that the weights should sum to 1.

1.4.17 Write a code fragment to multiply two rectangular matrices that are not necessarily square. *Note*: For the dot product to be well-defined, the number of columns in the first matrix must be equal to the number of rows in the second matrix. Print an error message if the dimensions do not satisfy this condition.

1.4.18 Modify `SelfAvoidingWalk` (PROGRAM 1.4.4) to calculate and print the average length of the paths as well as the dead-end probability. Keep separate the average lengths of escape paths and dead-end paths.

1.4.19 Modify `SelfAvoidingWalk` to calculate and print the average area of the smallest axis-oriented rectangle that encloses the path. Keep separate statistics for escape paths and dead-end paths.

Creative Exercises

1.4.20 *Dice simulation.* The following code computes the exact probability distribution for the sum of two dice:

```
double[] dist = new double[13];
for (int i = 1; i <= 6; i++)
   for (int j = 1; j <= 6; j++)
      dist[i+j] += 1.0;

for (int k = 1; k <= 12; k++)
   dist[k] /= 36.0;
```

The value `dist[k]` is the probability that the dice sum to `k`. Run experiments to validate this calculation simulating N dice throws, keeping track of the frequencies of occurrence of each value when you compute the sum of two random integers between 1 and 6. How large does N have to be before your empirical results match the exact results to three decimal places?

1.4.21 *Longest plateau.* Given an array of integers, find the length and location of the longest contiguous sequence of equal values where the values of the elements just before and just after this sequence are smaller.

1.4.22 *Empirical shuffle check.* Run computational experiments to check that our shuffling code works as advertised. Write a program `ShuffleTest` that takes command-line arguments M and N, does N shuffles of an array of size M that is initialized with `a[i] = i` before each shuffle, and prints an M-by-M table such that row `i` gives the number of times `i` wound up in position `j` for all `j`. All entries in the array should be close to N/M.

1.4.23 *Bad shuffling.* Suppose that you choose a random integer between `0` and `N-1` in our shuffling code instead of one between `i` and `N-1`. Show that the resulting order is *not* equally likely to be one of the $N!$ possibilities. Run the test of the previous exercise for this version.

1.4.24 *Music shuffling.* You set your music player to shuffle mode. It plays each of the N songs before repeating any. Write a program to estimate the likelihood that you will not hear any sequential pair of songs (that is, song 3 does not follow song 2, song 10 does not follow song 9, and so on).

1.4.24 *Minima in permutations.* Write a program that takes an integer `N` from the command line, generates a random permutation, prints the permutation, and prints the number of left-to-right minima in the permutation (the number of times an element is the smallest seen so far). Then write a program that takes integers `M` and `N` from the command line, generates `M` random permutations of size `N`, and prints the average number of left-to-right minima in the permutations generated. *Extra credit*: Formulate a hypothesis about the number of left-to-right minima in a permutation of size `N`, as a function of `N`.

1.4.25 *Inverse permutation.* Write a program that reads in a permutation of the integers `0` to `N-1` from `N` command-line arguments and prints the inverse permutation. (If the permutation is in an array `a[]`, its inverse is the array `b[]` such that `a[b[i]] = b[a[i]] = i`.) Be sure to check that the input is a valid permutation.

1.4.26 *Hadamard matrix.* The N-by-N Hadamard matrix $H(N)$ is a boolean matrix with the remarkable property that any two rows differ in exactly $N/2$ entries. (This property makes it useful for designing error-correcting codes.) $H(1)$ is a 1-by-1 matrix with the single entry `true`, and for $N>1$, $H(2N)$ is obtained by aligning four copies of $H(N)$ in a large square, and then inverting all of the entries in the lower right N-by-N copy, as shown in the following examples (with `T` representing `true` and `F` representing `false`, as usual).

$H(1)$	$H(2)$	$H(4)$
T	T T	T T T T
	T F	T F T F
		T T F F
		T F F T

Write a program that takes one command-line argument N and prints $H(N)$. Assume that N is a power of 2.

1.4.27 *Rumors.* Alice is throwing a party with N other guests, including Bob. Bob starts a rumor about Alice by telling it to one of the other guests. A person hearing this rumor for the first time will immediately tell it to one other guest, chosen at random from all the people at the party except Alice and the person from whom

they heard it. If a person (including Bob) hears the rumor for a second time, he or she will not propagate it further. Write a program to estimate the probability that everyone at the party (except Alice) will hear the rumor before it stops propagating. Also calculate an estimate of the expected number of people to hear the rumor.

1.4.28 *Find a duplicate.* Given an array of `N` elements with each element between `1` and `N`, write an algorithm to determine whether there are any duplicates. You do not need to preserve the contents of the given array, but do not use an extra array.

1.4.29 *Counting primes.* Compare `PrimeSieve` with the method that we used to demonstrate the `break` statement, at the end of SECTION 1.3. This is a classic example of a time-space tradeoff: `PrimeSieve` is fast, but requires a `boolean` array of size N; the other approach uses only two integer variables, but is substantially slower. Estimate the magnitude of this difference by finding the value of N for which this second approach can complete the computation in about the same time as `java PrimeSeive 1000000`.

1.4.30 *Minesweeper.* Write a program that takes 3 command-line arguments M, N, and p and produces an M-by-N boolean array where each entry is occupied with probability p. In the minesweeper game, occupied cells represent bombs and empty cells represent safe cells. Print out the array using an asterisk for bombs and a period for safe cells. Then, replace each safe square with the number of neighboring bombs (above, below, left, right, or diagonal) and print out the solution.

```
* * . . .        * * 1 0 0
. . . . .        3 3 2 0 0
. * . . .        1 * 1 0 0
```

Try to write your code so that you have as few special cases as possible to deal with, by using an $(M+2)$-by-$(N+2)$ boolean array.

1.4.31 *Self-avoiding walk length.* Suppose that there is no limit on the size of the grid. Run experiments to estimate the average walk length.

1.4.32 *Three-dimensional self-avoiding walks.* Run experiments to verify that the dead-end probability is 0 for a three-dimensional self-avoiding walk and to compute the average walk length for various values of N.

1.4.33 *Random walkers.* Suppose that N random walkers, starting in the center of an N-by-N grid, move one step at a time, choosing to go left, right, up, or down with equal probability at each step. Write a program to help formulate and test a hypothesis about the number of steps taken before all cells are touched.

1.4.34 *Bridge hands.* In the game of bridge, four players are dealt hands of 13 cards each. An important statistic is the distribution of the number of cards in each suit in a hand. Which is the most likely, 5-3-3-2, 4-4-3-2, or 4-3-3-3?

1.4.35 *Birthday problem.* Suppose that people enter an empty room until a pair of people share a birthday. On average, how many people will have to enter before there is a match? Run experiments to estimate the value of this quantity. Assume birthdays to be uniform random integers between 0 and 364.

1.4.36 *Coupon collector.* Run experiments to validate the classical mathematical result that the expected number of coupons needed to collect N values is about NH_N. For example, if you are observing the cards carefully at the blackjack table (and the dealer has enough decks randomly shuffled together), you will wait until about 235 cards are dealt, on average, before seeing every card value.

1.4.37 *Binomial coefficients.* Write a program that builds and prints a two-dimensional ragged array `a` such that `a[N][k]` contains the probability that you get exactly `k` heads when you toss a coin `N` times. Take a command-line argument to specify the maximum value of `N`. These numbers are known as the *binomial distribution*: if you multiply each entry in row i by 2^N, you get the *binomial coefficients* (the coefficients of x^k in $(x+1)^N$) arranged in *Pascal's triangle*. To compute them, start with `a[N][0] = 0` for all `N` and `a[1][1] = 1`, then compute values in successive rows, left to right, with `a[N][k] = (a[N-1][k] + a[N-1][k-1])/2`.

Pascal's triangle	*binomial distribution*
1	1
1 1	1/2 1/2
1 2 1	1/4 1/2 1/4
1 3 3 1	1/8 3/8 3/8 1/8
1 4 6 4 1	1/16 1/4 3/8 1/4 1/16