Chapter 5

More Data Types and Operators

Key Skills & Concepts

- Understand and create arrays
- Create multidimensional arrays
- Create irregular arrays
- Know the alternative array declaration syntax
- Assign array references
- Use the **length** array member

- Use the for-each style **for** loop
- Work with strings
- Apply command-line arguments
- Use type inference with local variables
- Use the bitwise operators
- Apply the **?** operator

This chapter returns to the subject of Java's data types and operators. It discusses arrays, the **String** type, local variable type inference, the bitwise operators, and the **?** ternary operator. It also covers Java's for-each style **for** loop. Along the way, command-line arguments are described.

Arrays

An *array* is a collection of variables of the same type, referred to by a common name. In Java, arrays can have one or more dimensions, although the one-dimensional array is the most common. Arrays are used for a variety of purposes because they offer a convenient means of grouping together related variables. For example, you might use an array to hold a record of the daily high temperature for a month, a list of stock price averages, or a list of your collection of programming books.

The principal advantage of an array is that it organizes data in such a way that it can be easily manipulated. For example, if you have an array containing the incomes for a selected group of households, it is easy to compute the average income by cycling through the array. Also, arrays organize data in such a way that it can be easily sorted.

Although arrays in Java can be used just like arrays in other programming languages, they have one special attribute: they are implemented as objects. This fact is one reason that a discussion of arrays was deferred until objects had been introduced. By implementing arrays as objects, several important advantages are gained, not the least of which is that unused arrays can be garbage collected.

One-Dimensional Arrays

A one-dimensional array is a list of related variables. Such lists are common in programming. For example, you might use a one-dimensional array to store the account numbers of the active users on a network. Another array might be used to store the current batting averages for a baseball team.

To declare a one-dimensional array, you can use this general form:

```
type[ ] array-name = new type[size];
  // body of method
}
```

Here, *type* declares the element type of the array. (The element type is also commonly referred to as the base type.) The element type determines the data type of each element contained in the array. The number of elements that the array will hold is determined by *size*. Since arrays are implemented as objects, the creation of an array is a two-step process. First, you declare an array reference variable. Second, you allocate memory for the array, assigning a reference to that memory to the array variable. Thus, arrays in Java are dynamically allocated using the **new** operator.

Here is an example. The following creates an **int** array of 10 elements and links it to an array reference variable named **sample**:

```
int[] sample = new int[10];
```

This declaration works just like an object declaration. The **sample** variable holds a reference to the memory allocated by **new**. This memory is large enough to hold 10 elements of type **int**. As with objects, it is possible to break the preceding declaration in two. For example:

```
int[] sample;
sample = new int[10];
```

In this case, when **sample** is first created, it refers to no physical object. It is only after the second statement executes that **sample** is linked with an array.

An individual element within an array is accessed by use of an index. An *index* describes the position of an element within an array. In Java, all arrays have zero as the index of their first element. Because **sample** has 10 elements, it has index values of 0 through 9. To index an array, specify the number of the element you want, surrounded by square brackets. Thus, the first element in **sample** is **sample[0]**, and the last element is **sample[9]**. For example, the following program loads **sample** with the numbers 0 through 9:

```
// Demonstrate a one-dimensional array.
class ArrayDemo {
  public static void main(String[] args) {
    int[] sample = new int[10];
    int i;

    for(i = 0; i < 10; i = i+1)
      sample[i] = i;

    for(i = 0; i < 10; i = i+1)
```

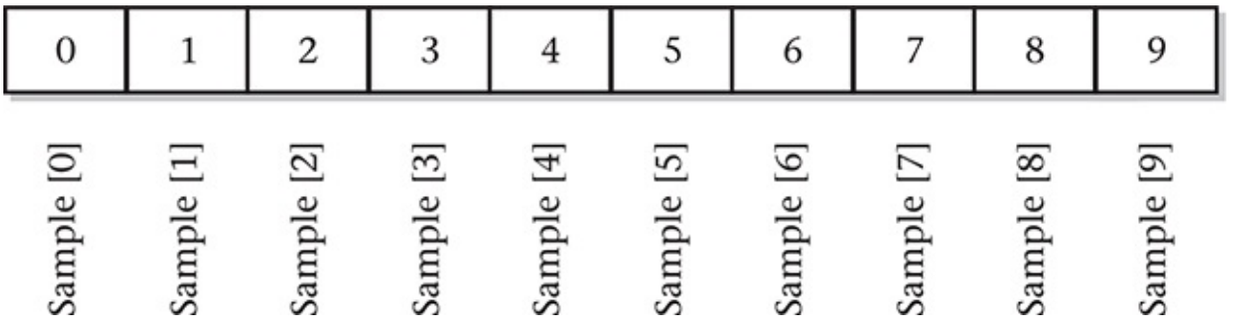
Arrays are indexed from zero.

```
        System.out.println("This is sample[" + i + "]: " +
                           sample[i]);
    }
}
```

The output from the program is shown here:

```
This is sample[0]: 0
This is sample[1]: 1
This is sample[2]: 2
This is sample[3]: 3
This is sample[4]: 4
This is sample[5]: 5
This is sample[6]: 6
This is sample[7]: 7
This is sample[8]: 8
This is sample[9]: 9
```

Conceptually, the **sample** array looks like this:



Arrays are common in programming because they let you deal easily with large numbers of related variables. For example, the following program finds the minimum and maximum values stored in the **nums** array by cycling through the array using a **for** loop:

```
// Find the minimum and maximum values in an array.
class MinMax {
  public static void main(String[] args) {
    int[] nums = new int[10];
    int min, max;

    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
    nums[9] = 49;

    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    System.out.println("min and max: " + min + " " + max);
  }
}
```

The output from the program is shown here:

```
min and max: -978 100123
```

In the preceding program, the **nums** array was given values by hand, using 10 separate assignment statements. Although perfectly correct, there is an easier way to accomplish this. Arrays can be initialized when they are created. The general form for initializing a one-dimensional array is shown here:

type[ ] *array-name* = { *val1, val2, val3, ... , valN* };

Here, the initial values are specified by *val1* through *valN*. They are assigned in sequence, left to right, in index order. Java automatically allocates an array large enough to hold the initializers that you specify. There is no need to explicitly use the **new** operator. For example, here is a better way to write the **MinMax** program:

```
// Use array initializers.
class MinMax2 {
  public static void main(String[] args) {
    int[] nums = { 99, -10, 100123, 18, -978,
                   5623, 463, -9, 287, 49 };
    int min, max;

    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    System.out.println("Min and max: " + min + " " + max);
  }
}
```

Array boundaries are strictly enforced in Java; it is a run-time error to overrun or underrun the end of an array. If you want to confirm this for yourself, try the following program that purposely overruns an array:

```
// Demonstrate an array overrun.
class ArrayErr {
  public static void main(String[] args) {
    int[] sample = new int[10];
    int i;

    // generate an array overrun
    for(i = 0; i < 100; i = i+1)
      sample[i] = i;
  }
}
```

As soon as **i** reaches 10, an **ArrayIndexOutOfBoundsException** is generated and the program is terminated.

Try This 5-1 Sorting an Array

`Bubble.java`

Because a one-dimensional array organizes data into an indexable linear list, it is the perfect data structure for sorting. In this project you will learn a simple way to sort an array. As you may know, there are a number of different sorting algorithms. There are the quick sort, the shaker sort, and the shell sort, to name just three. However, the best known, simplest, and easiest to understand is called the Bubble sort. Although the Bubble sort is not very efficient

—in fact, its performance is unacceptable for sorting large arrays—it may be used effectively for sorting small arrays.

1. Create a file called **Bubble.java**.
2. The Bubble sort gets its name from the way it performs the sorting operation. It uses the repeated comparison and, if necessary, exchange of adjacent elements in the array. In this process, small values move toward one end and large ones toward the other end. The process is conceptually similar to bubbles finding their own level in a tank of water. The Bubble sort operates by making several passes through the array, exchanging out-of-place elements when necessary. The number of passes required to ensure that the array is sorted is equal to one less than the number of elements in the array.

   Here is the code that forms the core of the Bubble sort. The array being sorted is called **nums**.

```
// This is the Bubble sort.
for(a=1; a < size; a++)
  for(b=size-1; b >= a; b--) {
    if(nums[b-1] > nums[b]) { // if out of order
      // exchange elements
      t = nums[b-1];
      nums[b-1] = nums[b];
      nums[b] = t;
  }
}
```

   Notice that sort relies on two **for** loops. The inner loop checks adjacent elements in the array, looking for out-of-order elements. When an out-of-order element pair is found, the two elements are exchanged. With each pass, the smallest of the remaining elements moves into its proper location. The outer loop causes this process to repeat until the entire array has been sorted.

3. Here is the entire **Bubble** program:

```
/*
    Try This 5-1

    Demonstrate the Bubble sort.
*/

class Bubble {
  public static void main(String[] args) {
    int[] nums = { 99, -10, 100123, 18, -978,
                   5623, 463, -9, 287, 49 };
    int a, b, t;
    int size;

    size = 10; // number of elements to sort

    // display original array
    System.out.print("Original array is:");
    for(int i=0; i < size; i++)
      System.out.print(" " + nums[i]);
    System.out.println();

    // This is the Bubble sort.
    for(a=1; a < size; a++)
      for(b=size-1; b >= a; b--) {
        if(nums[b-1] > nums[b]) { // if out of order
          // exchange elements
          t = nums[b-1];
          nums[b-1] = nums[b];
          nums[b] = t;
        }
      }

    // display sorted array
    System.out.print("Sorted array is:");
    for(int i=0; i < size; i++)
      System.out.print(" " + nums[i]);
    System.out.println();
  }
}
```

The output from the program is shown here:

```
Original array is: 99 -10 100123 18 -978 5623 463 -9 287 49
Sorted array is: -978 -10 -9 18 49 99 287 463 5623 100123
```

4. Although the Bubble sort is good for small arrays, it is not efficient when used on larger ones. A much better general-purpose sorting algorithm is the Quicksort. The Quicksort, however, relies on features of Java that you have not yet learned about.

Multidimensional Arrays

Although the one-dimensional array is often the most commonly used array in programming, multidimensional arrays (arrays of two or more dimensions) are certainly not rare. In Java, a multidimensional array is an array of arrays.

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array **table** of size 10, 20 you would write

```
int[][] table = new int[10][20];
```

Pay careful attention to the declaration. Unlike some other computer languages, which use commas to separate the array dimensions, Java places each dimension in its own set of brackets. Similarly, to access point 3, 5 of array **table**, you would use **table[3][5]**.

In the next example, a two-dimensional array is loaded with the numbers 1 through 12.

```
// Demonstrate a two-dimensional array.
class TwoD {
  public static void main(String[] args) {
    int t, i;
    int[][] table = new int[3][4];

    for(t=0; t < 3; ++t) {
      for(i=0; i < 4; ++i) {
        table[t][i] = (t*4)+i+1;
        System.out.print(table[t][i] + " ");
      }
      System.out.println();
    }
  }
}
```

In this example, **table[0][0]** will have the value 1, **table[0][1]** the value 2, **table[0][2]** the value 3, and so on. The value of **table[2][3]** will be 12. Conceptually, the array will look like that shown in Figure 5-1.

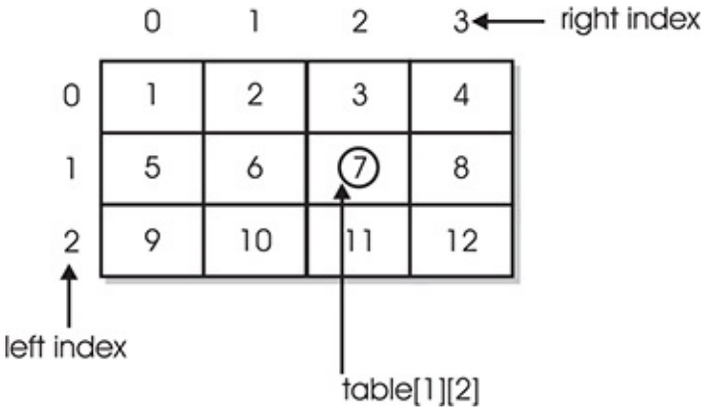


Figure 5-1 Conceptual view of the **table** array by the **TwoD** program

Irregular Arrays

When you allocate memory for a multidimensional array, you need to specify only the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, the following code allocates memory for the first dimension of **table** when it is declared. It allocates the second dimension manually.

```
int[][] table = new int[3][];
table[0] = new int[4];
table[1] = new int[4];
table[2] = new int[4];
```

Although there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions separately, you do not need to allocate the same number of elements for each index. Since multidimensional arrays are implemented as arrays of arrays, the length of each array is under your control. For example, assume you are writing a program that stores the number of passengers that ride an airport shuttle. If the shuttle runs 10 times a day during the week and twice a day on Saturday and Sunday, you could use the **riders** array shown in the following program to store the information. Notice that the length of the second dimension for the first five indices is 10 and the length of the second dimension for the last two indices is 2.

```
// Manually allocate differing size second dimensions.
class Ragged {
  public static void main(String[] args) {
    int[][] riders = new int[7][];
    riders[0] = new int[10];
    riders[1] = new int[10];
    riders[2] = new int[10];
    riders[3] = new int[10];
    riders[4] = new int[10];
    riders[5] = new int[2];
    riders[6] = new int[2];

    int i, j;

    // fabricate some fake data
    for(i=0; i < 5; i++)
      for(j=0; j < 10; j++)
        riders[i][j] = i + j + 10;
    for(i=5; i < 7; i++)
      for(j=0; j < 2; j++)
        riders[i][j] = i + j + 10;

    System.out.println("Riders per trip during the week:");
    for(i=0; i < 5; i++) {
      for(j=0; j < 10; j++)
        System.out.print(riders[i][j] + " ");
      System.out.println();
    }

    System.out.println();

    System.out.println("Riders per trip on the weekend:");
    for(i=5; i < 7; i++) {
      for(j=0; j < 2; j++)
        System.out.print(riders[i][j] + " ");
      System.out.println();
    }
  }
}
```

Here, the second dimensions are 10 elements long.

But here, they are 2 elements long.

The use of irregular (or ragged) multidimensional arrays does not, obviously, apply to all cases. However, irregular arrays can be quite effective in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), an irregular array might be a perfect solution.

Arrays of Three or More Dimensions

Java allows arrays with more than two dimensions. Here is the general form of a multidimensional array declaration:

type[ ] [ ]...[ ] *name* = new *type*[*size1*][*size2*]...[*sizeN*];

For example, the following declaration creates a 4 × 10 × 3 three-dimensional integer array.

```
int[][][] multidim = new int[4][10][3];
```

Initializing Multidimensional Arrays

A multidimensional array can be initialized by enclosing each dimension's initializer list within its own set of curly braces. For example, the general form of array initialization for a two-dimensional array is shown here:

type-specifier[ ] [ ] *array_name* = {
  { *val, val, val, ..., val* },
  { *val, val, val, ..., val* },
.
.
.
  { *val, val, val, ..., val* }
};

Here, *val* indicates an initialization value. Each inner block designates a row. Within each row, the first value will be stored in the first position of the subarray, the second value in the second position, and so on. Notice that commas separate the initializer blocks and that a semicolon follows the closing **}**.

For example, the following program initializes an array called **sqrs** with the numbers 1 through 10 and their squares:

```
// Initialize a two-dimensional array.
class Squares {
  public static void main(String[] args) {
    int[][] sqrs = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };
    int i, j;

    for(i=0; i < 10; i++) {
      for(j=0; j < 2; j++)
        System.out.print(sqrs[i][j] + " ");
      System.out.println();
    }
  }
}
```

Notice how each row has its own set of initializers.

Here is the output from the program:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Alternative Array Declaration Syntax

There is a second form that can be used to declare an array:

```
type var-name[ ];
```

Here, the square brackets follow the name of the array variable, not the type specifier. For

example, the following two declarations are equivalent:

```
int counter[] = new int[3];
int[] counter = new int[3];
```

The following declarations are also equivalent:

```
char table[][] = new char[3][4];
char[][] table = new char[3][4];
```

This alternative declaration form offers convenience when converting code from C/C++ to Java. (In C/C++, arrays are declared in a fashion similar to Java's alternative form.) It also lets you declare both array and non-array variables in a single declaration statement. Today, the alternative form of array declaration is less commonly used, but it is still important that you are familiar with it because both forms of array declarations are legal in Java.

Assigning Array References

As with other objects, when you assign one array reference variable to another, you are simply changing what object that variable refers to. You are not causing a copy of the array to be made, nor are you causing the contents of one array to be copied to the other. For example, consider this program:

```
// Assigning array reference variables.
class AssignARef {
  public static void main(String[] args) {
    int i;

    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < 10; i++)
      nums1[i] = i;

    for(i=0; i < 10; i++)
      nums2[i] = -i;

    System.out.print("Here is nums1: ");
    for(i=0; i < 10; i++)
      System.out.print(nums1[i] + " ");
    System.out.println();

    System.out.print("Here is nums2: ");
    for(i=0; i < 10; i++)
      System.out.print(nums2[i] + " ");
    System.out.println();

    nums2 = nums1; // now nums2 refers to nums1 <-- Assign an array reference.

    System.out.print("Here is nums2 after assignment: ");
    for(i=0; i < 10; i++)
      System.out.print(nums2[i] + " ");
    System.out.println();

    // now operate on nums1 array through nums2
    nums2[3] = 99;

    System.out.print("Here is nums1 after change through nums2: ");
    for(i=0; i < 10; i++)
      System.out.print(nums1[i] + " ");
    System.out.println();
  }
}
```

The output from the program is shown here:

```
Here is nums1: 0 1 2 3 4 5 6 7 8 9
Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9
```

As the output shows, after the assignment of **nums1** to **nums2**, both array reference variables refer to the same object.

Using the length Member

Recall that in Java, arrays are implemented as objects. One benefit of this approach is that each array has associated with it a **length** instance variable that contains the number of elements that the array can hold. (In other words, **length** contains the size of the array.) Here is a program that demonstrates this property:

```
// Use the length array member.
class LengthDemo {
  public static void main(String[] args) {
    int[] list = new int[10];
    int[] nums = { 1, 2, 3 };
    int[][] table = { // a variable-length table
      {1, 2, 3},
      {4, 5},
      {6, 7, 8, 9}
    };

    System.out.println("length of list is " + list.length);
    System.out.println("length of nums is " + nums.length);
    System.out.println("length of table is " + table.length);
    System.out.println("length of table[0] is " + table[0].length);
    System.out.println("length of table[1] is " + table[1].length);
    System.out.println("length of table[2] is " + table[2].length);
    System.out.println();

    // use length to initialize list
    for(int i=0; i < list.length; i++)
      list[i] = i * i;

    System.out.print("Here is list: ");
    // now use length to display list
    for(int i=0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }
}
```

Use **length** to control a **for** loop.

This program displays the following output:

```
length of list is 10
length of nums is 3
length of table is 3
length of table[0] is 3
length of table[1] is 2
length of table[2] is 4

Here is list: 0 1 4 9 16 25 36 49 64 81
```

Pay special attention to the way **length** is used with the two-dimensional array **table**. As explained, a two-dimensional array is an array of arrays. Thus, when the expression

```
table.length
```

is used, it obtains the number of arrays stored in table, which is 3 in this case. To obtain the length of any individual array in table, you will use an expression such as this,

```
table[0].length
```

which, in this case, obtains the length of the first array.

One other thing to notice in **LengthDemo** is the way that **list.length** is used by the **for** loops to govern the number of iterations that take place. Since each array carries with it its own length, you can use this information rather than manually keeping track of an array's size. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It contains the number of elements that the array is capable of holding.

The inclusion of the **length** member simplifies many algorithms by making certain types of array operations easier—and safer—to perform. For example, the following program uses **length** to copy one array to another while preventing an array overrun and its attendant run-time exception.

```
// Use length variable to help copy an array.
class ACopy {
  public static void main(String[] args) {
```

```
    int i;
    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < nums1.length; i++)
      nums1[i] = i;

    // copy nums1 to nums2
    if(nums2.length >= nums1.length) <-- Use length to compare array sizes.
      for(i = 0; i < nums1.length; i++)
        nums2[i] = nums1[i];

    for(i=0; i < nums2.length; i++)
      System.out.print(nums2[i] + " ");
  }
}
```

Here, **length** helps perform two important functions. First, it is used to confirm that the target array is large enough to hold the contents of the source array. Second, it provides the termination condition of the **for** loop that performs the copy. Of course, in this simple example, the sizes of the arrays are easily known, but this same approach can be applied to a wide range of more challenging situations.

Try This 5-2 A Queue Class

`QDemo.java`

As you may know, a data structure is a means of organizing data. The simplest data structure is the array, which is a linear list that supports random access to its elements. Arrays are often used as the underpinning for more sophisticated data structures, such as stacks and queues. A *stack* is a list in which elements can be accessed in first-in, last-out (FILO) order only. A *queue* is a list in which elements can be accessed in first-in, first-out (FIFO) order only. Thus, a stack is like a stack of plates on a table—the first down is the last to be used. A queue is like a line at a bank—the first in line is the first served.

What makes data structures such as stacks and queues interesting is that they combine storage for information with the methods that access that information. Thus, stacks and queues are *data engines* in which storage and retrieval are provided by the data structure itself, not manually by your program. Such a combination is, obviously, an excellent choice for a class, and in this project you will create a simple queue class.

)In general, queues support two basic operations: put and get. Each put operation places a new element on the end of the queue. Each get operation retrieves the next element from the front of the queue. Queue operations are *consumptive*: once an element has been retrieved, it cannot be retrieved again. The queue can also become full, if there is no space available to store an item, and it can become empty, if all of the elements have been removed.

One last point: There are two basic types of queues—circular and noncircular. A *circular queue* reuses locations in the underlying array when elements are removed. A *noncircular queue* does not reuse locations and eventually becomes exhausted. For the sake of simplicity, this example creates a noncircular queue, but with a little thought and effort, you can easily transform it into a circular queue.

1. Create a file called **QDemo.java**.

2. Although there are other ways to support a queue, the method we will use is based upon an array. That is, an array will provide the storage for the items put into the queue. This array will be accessed through two indices. The *put* index determines where the next element of data will be stored. The *get* index indicates at what location the next element of data will be obtained. Keep in mind that the get operation is consumptive, and it is not possible to retrieve the same element twice. Although the queue that we will be creating stores characters, the same logic can be used to store any type of object. Begin creating the **Queue** class with these lines:

```
class Queue {
  char[] q; // this array holds the queue
  int putloc, getloc; // the put and get indices
```

3. The constructor for the **Queue** class creates a queue of a given size. Here is the **Queue** constructor:

```
Queue(int size) {
  q = new char[size]; // allocate memory for queue
  putloc = getloc = 0;
}
```

Notice that the put and get indices are initially set to zero.

4. The **put( )** method, which stores elements, is shown next:

```
// put a character into the queue
void put(char ch) {
  if(putloc==q.length) {
    System.out.println(" - Queue is full.");
    return;
  }

  q[putloc++] = ch;
}
```

The method begins by checking for a queue-full condition. If **putloc** is equal to one past the last location in the **q** array, there is no more room in which to store elements.

Otherwise, the new element is stored at that location and **putloc** is incremented. Thus, **putloc** is always the index where the next element will be stored.

5. To retrieve elements, use the **get( )** method, shown next:

```
// get a character from the queue
char get() {
  if(getloc == putloc) {
    System.out.println(" - Queue is empty.");
    return (char) 0;
  }

  return q[getloc++];
}
```

Notice first the check for queue-empty. If **getloc** and **putloc** both index the same element, the queue is assumed to be empty. This is why **getloc** and **putloc** were both initialized to zero by the **Queue** constructor. Then, the next element is returned. In the process, **getloc** is incremented. Thus, **getloc** always indicates the location of the next element to be retrieved.

6. Here is the entire **QDemo.java** program:

```
/*
    Try This 5-2

    A queue class for characters.
*/

class Queue {
  char[] q; // this array holds the queue
  int putloc, getloc; // the put and get indices

  Queue(int size) {
    q = new char[size]; // allocate memory for queue
    putloc = getloc = 0;
  }

  // put a character into the queue
  void put(char ch) {
    if(putloc==q.length) {
      System.out.println(" - Queue is full.");
      return;
    }

    q[putloc++] = ch;
  }

  // get a character from the queue
  char get() {
    if(getloc == putloc) {
      System.out.println(" - Queue is empty.");
      return (char) 0;
    }
```

```
    return q[getloc++];
  }
}

// Demonstrate the Queue class.
class QDemo {
  public static void main(String[] args) {
    Queue bigQ = new Queue(100);
    Queue smallQ = new Queue(4);
    char ch;
    int i;

    System.out.println("Using bigQ to store the alphabet.");
    // put some numbers into bigQ
    for(i=0; i < 26; i++)
      bigQ.put((char) ('A' + i));

    // retrieve and display elements from bigQ
    System.out.print("Contents of bigQ: ");
    for(i=0; i < 26; i++) {
      ch = bigQ.get();
      if(ch != (char) 0) System.out.print(ch);
    }

    System.out.println("\n");


    System.out.println("Using smallQ to generate errors.");
    // Now, use smallQ to generate some errors
    for(i=0; i < 5; i++) {
      System.out.print("Attempting to store " +
                       (char) ('Z' - i));

      smallQ.put((char) ('Z' - i));

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

    // more errors on smallQ
    System.out.print("Contents of smallQ: ");
    for(i=0; i < 5; i++) {
      ch = smallQ.get();

      if(ch != (char) 0) System.out.print(ch);
    }
  }
}
```

7. The output produced by the program is shown here:

```
Using bigQ to store the alphabet.
Contents of bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Using smallQ to generate errors.

Attempting to store Z
Attempting to store Y
Attempting to store X
Attempting to store W
Attempting to store V - Queue is full.


Contents of smallQ: ZYXW - Queue is empty.
```

8. On your own, try modifying **Queue** so that it stores other types of objects. For example, have it store **int**s or **double**s.

The For-Each Style for Loop

When working with arrays, it is common to encounter situations in which each element in an array must be examined, from start to finish. For example, to compute the sum of the values held in an array, each element in the array must be examined. The same situation occurs when computing an average, searching for a value, copying an array, and so on. Because such "start to finish" operations are so common, Java defines a second form of the **for** loop that streamlines this operation.

The second form of the **for** implements a "for-each" style loop. A for-each loop cycles through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. For reasons that will become clear, for-each style loops have become quite popular among both computer language designers and programmers. Interestingly, Java did not originally offer a for-each style loop. However, several years ago (beginning with JDK 5), the **for** loop was enhanced to provide this option. The for-each style of **for** is also referred to as the *enhanced **for** loop*. Both terms are used in this book.

The general form of the for-each style **for** is shown here.

for(*type itr-var* : *collection*) *statement-or-block*

Here, *type* specifies the type, and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this book is the array. With each iteration

of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained. Thus, when iterating over an array of size *N*, the enhanced **for** obtains the elements in the array in index order, from 0 to *N*–1.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the element type of the array.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

Ask the Expert

Q: **Aside from arrays, what other types of collections can the for-each style** for **loop cycle through?**

A: One of the most important uses of the for-each style **for** is to cycle through the contents of a collection defined by the Collections Framework. The Collections Framework is a set of classes that implement various data structures, such as lists, vectors, sets, and maps. A discussion of the Collections Framework is beyond the scope of this book, but detailed coverage of the Collections Framework can be found in *Java: The Complete Reference, Twelfth Edition* (McGraw Hill, 2022)

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable. Furthermore, the starting and ending value for the loop control variable, and its increment, must be explicitly specified.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for:**

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1, on the second iteration, **x** contains 2, and so on. Not only is the syntax streamlined, it also prevents boundary errors.

Here is an entire program that demonstrates the for-each version of the **for** just described:

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // Use for-each style for to display and sum the values.
    for(int x : nums) {  <-- A for-each style for loop
      System.out.println("Value is: " + x);
      sum += x;
    }

    System.out.println("Summation: " + sum);
  }
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

As this output shows, the for-each style **for** automatically cycles through an array in sequence from the lowest index to the highest.

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement. For example, this loop sums only the first five elements of **nums**:

```
// Sum only the first 5 elements.
for(int x : nums) {
  System.out.println("Value is: " + x);
  sum += x;
  if(x == 5) break; // stop the loop when 5 is obtained
}
```

There is one important point to understand about the for-each style **for** loop. Its iteration variable is "read-only" as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can't change the contents of the array by assigning the iteration variable a new value. For example, consider this program:

```
// The for-each loop is essentially read-only.
class NoChange {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for(int x : nums) {
      System.out.print(x + " ");
      x = x * 10; // no effect on nums
    }

    System.out.println();

    for(int x : nums)
      System.out.print(x + " ");

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

This does not change **nums**.

The first **for** loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array **nums**, as the second **for** loop illustrates. The output, shown here, proves this point:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Iterating Over Multidimensional Arrays

The enhanced **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays.* (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array because each iteration obtains the *next array,* not an individual

element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. To understand the implications of this, consider the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row order, from first to last.

```
// Use for-each style for on a two-dimensional array.
class ForEach2 {
  public static void main(String[] args) {
    int sum = 0;
    int[][] nums = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // Use for-each for loop to display and sum the values.
    for(int[] x : nums) { <-- Notice how x is declared.
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

In the program, pay special attention to this line:

```
for(int[] x : nums) {
```

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for

Since the for-each style **for** can only cycle through an array sequentially, from start to finish, you might think that its use is limited. However, this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching.
For example, the following program uses a **for** loop to search an unsorted array for a value. It stops if the value is found.

```
// Search an array using for-each style for.
class Search {
  public static void main(String[] args) {
    int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // Use for-each style for to search nums for val.
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Value found!");
  }
}
```

The for-each style **for** is an excellent choice in this application because searching an unsorted array involves examining each element in sequence. (Of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

Now that the for-each style **for** has been introduced, it will be used where appropriate throughout the remainder of this book.

Strings

From a day-to-day programming standpoint, one of the most important of Java's data types is **String**. **String** defines and supports character strings. In some other programming languages, a string is an array of characters. This is not the case with Java. In Java, strings are objects.

Actually, you have been using the **String** class since Chapter 1, but you did not know it. When you create a string literal, you are actually creating a **String** object. For example, in the statement

```
System.out.println("In Java, strings are objects.");
```

the string "In Java, strings are objects." is automatically made into a **String** object by Java. Thus, the use of the **String** class has been "below the surface" in the preceding programs. In the following sections, you will learn to handle it explicitly. Be aware, however, that the **String** class is quite large, and we will only scratch its surface here. It is a class that you will want to explore on its own.

Constructing Strings

You can construct a **String** just like you construct any other type of object: by using **new** and calling the **String** constructor. For example:

```
String str = new String("Hello");
```

This creates a **String** object called **str** that contains the character string "Hello". You can also construct a **String** from another **String**. For example:

```
String str = new String("Hello");
String str2 = new String(str);
```

After this sequence executes, **str2** will also contain the character string "Hello".

Another easy way to create a **String** is shown here:

```
String str = "Java strings are powerful.";
```

In this case, **str** is initialized to the character sequence "Java strings are powerful."

Once you have created a **String** object, you can use it anywhere that a quoted string is allowed. For example, you can use a **String** object as an argument to **println( )**, as shown in this example:

```
// Introduce String.
class StringDemo {
  public static void main(String[] args) {
    // declare strings in various ways
    String str1 = new String("Java strings are objects.");
    String str2 = "They are constructed various ways.";
    String str3 = new String(str2);

    System.out.println(str1);
    System.out.println(str2);
    System.out.println(str3);
  }
}
```

The output from the program is shown here:

```
Java strings are objects.
They are constructed various ways.
They are constructed various ways.
```

Operating on Strings

The **String** class contains several methods that operate on strings. Here are the general forms

for a few:

boolean equals(*str*)	Returns true if the invoking string contains the same character sequence as *str*.
int length( )	Obtains the length of a string.
char charAt(*index*)	Obtains the character at the index specified by *index*.
int compareTo(*str*)	Returns less than zero if the invoking string is less than *str*, greater than zero if the invoking string is greater than *str*, and zero if the strings are equal.
int indexOf(*str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the first match or –1 on failure.
int lastIndexOf(*str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the last match or –1 on failure.

Here is a program that demonstrates these methods:

```
// Some String operations.
class StrOps {
  public static void main(String[] args) {
    String str1 =
      "When it comes to Web programming, Java is #1.";
    String str2 = new String(str1);
    String str3 = "Java strings are powerful.";
    int result, idx;
    char ch;

    System.out.println("Length of str1: " +
                       str1.length());

    // display str1, one char at a time.
    for(int i=0; i < str1.length(); i++)
      System.out.print(str1.charAt(i));
    System.out.println();

    if(str1.equals(str2))
      System.out.println("str1 equals str2");
    else
      System.out.println("str1 does not equal str2");

    if(str1.equals(str3))
      System.out.println("str1 equals str3");
    else
      System.out.println("str1 does not equal str3");
```

```
    result = str1.compareTo(str3);
    if(result == 0)
      System.out.println("str1 and str3 are equal");
    else if(result < 0)
      System.out.println("str1 is less than str3");
    else
      System.out.println("str1 is greater than str3");

    // assign a new string to str2
    str2 = "One Two Three One";

    idx = str2.indexOf("One");
    System.out.println("Index of first occurrence of One: " + idx);
    idx = str2.lastIndexOf("One");
    System.out.println("Index of last occurrence of One: " + idx);
  }
}
```

This program generates the following output:

```
Length of str1: 45
When it comes to Web programming, Java is #1.
str1 equals str2
str1 does not equal str3
str1 is greater than str3
Index of first occurrence of One: 0
Index of last occurrence of One: 14
```

You can *concatenate* (join together) two strings using the + operator. For example, this statement

```
String str1 = "One";
String str2 = "Two";
String str3 = "Three";
String str4 = str1 + str2 + str3;
```

initializes **str4** with the string "OneTwoThree".

Ask the Expert

Q: Why does **String** define the **equals( )** method? Can’t I just use ==?

A: The **equals( )** method compares the character sequences of two **String** objects for

equality. Applying the == to two **String** references simply determines whether the two references refer to the same object.

Arrays of Strings

Like any other data type, strings can be assembled into arrays. For example:

```
// Demonstrate String arrays.
class StringArrays {
  public static void main(String[] args) {
    String[] strs = { "This", "is", "a", "test." };

    System.out.println("Original array: ");
    for(String s : strs)
      System.out.print(s + " ");
    System.out.println("\n");

    // change a string
    strs[1] = "was";
    strs[3] = "test, too!";

    System.out.println("Modified array: ");
    for(String s : strs)
      System.out.print(s + " ");
  }
}
```

Here is the output from this program:

```
Original array:
This is a test.

Modified array:
This was a test, too!
```

Strings Are Immutable

The contents of a **String** object are immutable. That is, once created, the character sequence that makes up the string cannot be altered. This restriction allows Java to implement strings more efficiently. Even though this probably sounds like a serious drawback, it isn't. When you need a string that is a variation on one that already exists, simply create a new string that contains the desired changes. Since unused **String** objects are automatically garbage collected, you don't even need to worry about what happens to the discarded strings. It must be made clear, however, that **String** reference variables may, of course, change the object to

which they refer. It is just that the contents of a specific **String** object cannot be changed after it is created.

To fully understand why immutable strings are not a hindrance, we will use another of **String**'s methods: **substring( )**. The **substring( )** method returns a new string that contains a specified portion of the invoking string. Because a new **String** object is manufactured that contains the substring, the original string is unaltered, and the rule of immutability remains intact. The form of **substring( )** that we will be using is shown here:

```
String substring(int startIndex, int endIndex)
```

Ask the Expert

Q: **You say that once created,** String **objects are immutable. I understand that, from a practical point of view, this is not a serious restriction, but what if I want to create a string that can be changed?**

A: You're in luck. Java offers a class called **StringBuffer**, which creates string objects that can be changed. For example, in addition to the **charAt( )** method, which obtains the character at a specific location, **StringBuffer** defines **setCharAt( )**, which sets a character within the string. Java also supplies **StringBuilder**, which is related to **StringBuffer**, and also supports strings that can be changed. However, for most purposes you will want to use **String**, not **StringBuffer** or **StringBuilder**.

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. Here is a program that demonstrates **substring( )** and the principle of immutable strings:

```
// Use substring().
class SubStr {
  public static void main(String[] args) {
    String orgstr = "Java makes the Web move.";

    // construct a substring
    String substr = orgstr.substring(5, 18);

    System.out.println("orgstr: " + orgstr);
    System.out.println("substr: " + substr);
  }
}
```

This creates a new string that contains the desired substring.

Here is the output from the program:

```
orgstr: Java makes the Web move.
substr: makes the Web
```

As you can see, the original string **orgstr** is unchanged, and **substr** contains the substring.

Using a String to Control a switch Statement

As explained in Chapter 3, in the past a **switch** had to be controlled by an integer type, such as **int** or **char**. This precluded the use of a **switch** in situations in which one of several actions is selected based on the contents of a string. Instead, an **if-else-if** ladder was the typical solution. Although an **if-else-if** ladder is semantically correct, a **switch** statement would be the more natural idiom for such a selection. Fortunately, this situation has been remedied. With a modern version of Java, you can use a **String** to control a switch. This results in more readable, streamlined code in many situations.

Here is an example that demonstrates controlling a **switch** with a **String**:

```
// Use a string to control a switch statement.

class StringSwitch {
  public static void main(String[] args) {

    String command = "cancel";

    switch(command) {
      case "connect":
        System.out.println("Connecting");
        break;
      case "cancel":
        System.out.println("Canceling");
        break;
      case "disconnect":
        System.out.println("Disconnecting");
        break;
      default:
        System.out.println("Command Error!");
        break;
    }
  }
}
```

As you would expect, the output from the program is

```
Canceling
```

The string contained in **command** (which is "cancel" in this program) is tested against the

case constants. When a match is found (as it is in the second **case**), the code sequence associated with that sequence is executed.

Being able to use strings in a **switch** statement can be very convenient and can improve the readability of some code. For example, using a string-based **switch** is an improvement over using the equivalent sequence of **if**/**else** statements. However, switching on strings can be less efficient than switching on integers. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form. In other words, don't use strings in a **switch** unnecessarily.

Ask the Expert

Q: I have heard about another type of string literal called a *text block*. Can you tell me about it?

A: Yes, text blocks were added to Java by JDK 15. A text block is a new kind of string literal that is comprised of a sequence of characters that can occupy more than one line. A text block reduces the tedium programmers often face when creating multiline string literals because newline characters can be used in a text block without the need for the **\n** escape sequence. Furthermore, tab and double quote characters can also be entered directly, without using an escape sequence, and the indentation of a multiline string can be preserved. Thus, text blocks provide an elegant alternative to what can be a rather annoying process.

A text block is supported by a delimiter that consists of three double-quote characters: """. A block of text is created by enclosing a string within a set of these delimiters. Specifically, a text block begins immediately following the newline after the opening """. Thus, the opening delimiter must end with a newline. The text block begins on the next line. A text block ends at the first character of the closing """. It is important to emphasize that even though a text block uses the """ delimiter, it is still of type **String**. Thus, a text block can be used wherever any other string can.

Here is a simple example of a text block. It assigns a multiline text block to **str**.

```
String str = """
Text blocks make multiple lines easy because they eliminate
    the need to use \n escape sequences to indicate a newline.
As a result, text blocks make the programmer's life better!
""";
```

This example creates a string in which each line is separated from the next by a newline. It is not necessary to use the **\n** escape sequence to obtain the newline. Thus, the text block automatically preserves the newlines in the text. Notice that the second line is indented. When **str** is output using this statement:

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

the following is displayed:

```
Text blocks make multiple lines easy because they eliminate
    the need to use \n escape sequences to indicate a newline.
As a result, text blocks make the programmer's life better!
```

As the output shows, the newlines and the indentation of the second line are preserved. These are key benefits of text blocks.

Text blocks have additional attributes, such as the ability to removed unwanted leading whitespace. It is a feature that you will want to look at more closely as you advance in your study of Java. Simply put, text blocks make what was often a difficult coding task easy.

Using Command-Line Arguments

Now that you know about the **String** class, you can understand the **args** parameter to **main( )** that has been in every program shown so far. Many programs accept what are called *command-line arguments.* A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to **main( )**. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line information.
class CLDemo {
  public static void main(String[] args) {
    System.out.println("There are " + args.length +
                       " command-line arguments.");

    System.out.println("They are: ");
    for(int i=0; i<args.length; i++)
      System.out.println("arg[" + i + "]: " + args[i]);
  }
}
```

If **CLDemo** is executed like this,

```
java CLDemo one two three
```

you will see the following output:

```
There are 3 command-line arguments.
They are:
arg[0]: one
arg[1]: two
arg[2]: three
```

Notice that the first argument is stored at index 0, the second argument is stored at index 1, and so on.

To get a taste of the way command-line arguments can be used, consider the next program. It takes one command-line argument that specifies a person's name. It then searches through a two-dimensional array of strings for that name. If it finds a match, it displays that person's telephone number.

```
// A simple automated telephone directory.
class Phone {
  public static void main(String[] args) {
    String[][] numbers = {
      { "Tom", "555-3322" },
      { "Mary", "555-8976" },
      { "Jon", "555-1037" },
      { "Rachel", "555-1400" }
    };

    int i;

    if(args.length != 1) ←
      System.out.println("Usage: java Phone <name>");
    else {
      for(i=0; i<numbers.length; i++) {
        if(numbers[i][0].equals(args[0])) {
          System.out.println(numbers[i][0] + ": " +
                             numbers[i][1]);
          break;
        }
      }
      if(i == numbers.length)
        System.out.println("Name not found.");
    }
  }
}
```

To use the program, one command-line argument must be present.

Here is a sample run:

```
java Phone Mary
Mary: 555-8976
```

Using Type Inference with Local Variables

Not long ago, a feature called *local variable type inference* was added to the Java language. To begin, let's review two important aspects of variables. First, all variables in Java must be declared prior to their use. Second, a variable can be initialized with a value when it is declared. Furthermore, when a variable is initialized, the type of the initializer must be the same as (or convertible to) the declared type of the variable. Thus, in principle, it would not be necessary to specify an explicit type for an initialized variable because it could be inferred from the type of its initializer. Of course, in the past, Java did not support such inference and all variables required an explicitly declared type, whether they were initialized or not. Today, that situation has changed.

Beginning with JDK 10, it became possible to let the compiler infer the type of a local variable based on the type of its initializer, thus avoiding the need to explicitly specify the type. Local variable type inference offers a number of advantages. For example, it can streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer. It can simplify declarations in cases in which the type is quite lengthy, such as can be the case with some class names. It can also be helpful when a type is difficult to discern or cannot be denoted. (An example of a type that cannot be denoted is the type of an anonymous class, discussed in Chapter 17.) Furthermore, local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up-to-date with evolving trends in language design. To support local variable type inference, the context-sensitive keyword **var** was added to Java.

To use local variable type inference, the variable must be declared with **var** as the type name and it must include an initializer. Let's begin with a simple example. Consider the following statement that declares a local **double** variable called **avg** that is initialized with the value 10.0:

```
double avg = 10.0;
```

Using type inference, this declaration can also be written like this:

```
var avg = 10.0;
```

In both cases, **avg** will be of type **double**. In the first case, its type is explicitly specified. In the second, its type is inferred as **double** because the initializer 10.0 is of type **double**.

As mentioned, **var** is context-sensitive. When it is used as the type name in the context of a local variable declaration, it tells the compiler to use type inference to determine the type of the variable being declared based on the type of the initializer. Thus, in a local variable declaration, **var** is a placeholder for the actual inferred type. However, when used in most other places, **var** is simply a user-defined identifier with no special meaning. For example, the following declaration is still valid:

```
int var = 1;  // In this case, var is simply a user-defined identifier.
```

In this case, the type is explicitly specified as **int** and **var** is the name of the variable being declared. Even though it is context-sensitive, there are a few places in which the use of **var** is illegal. It cannot be used as the name of a class, for example.

The following program puts the preceding discussion into action:

```
// A simple demonstration of local variable type inference.
class VarDemo {
  public static void main(String[] args) {

    // Use type inference to determine the type of the
    // variable named avg. In this case, double is inferred.
    var avg = 10.0; <-- Use var to infer type of avg.
    System.out.println("Value of avg: " + avg);

    // In the following context, var is not a predefined identifier.
    // It is simply a user-defined variable name.
    int var = 1;
    System.out.println("Value of var: " + var);

    // Interestingly, in the following sequence, var is used
    // as both the type of the declaration and as a variable name
    // in the initializer.
    var k = -var;
    System.out.println("Value of k: " + k);
  }
}
```

Here is the output:

```
Value of avg: 10.0
Value of var: 1
Value of k: -1
```

The preceding example uses **var** to declare only simple variables, but you can also use **var** to declare an array. For example:

```
var myArray = new int[10]; // This is valid.
```

Notice that neither **var** nor **myArray** has brackets. Instead, the type of **myArray** is inferred to be **int[ ]**. Furthermore, you *cannot* use brackets on the left side of a **var** declaration. Thus, both of these declarations are invalid:

```
var[] myArray = new int[10]; // Wrong
var myArray[] = new int[10]; // Wrong
```

In the first line, an attempt is made to bracket **var**. In the second, an attempt is made to

bracket **myArray**. In both cases, the use of the brackets is wrong because the type is inferred from the type of the initializer.

It is important to emphasize that **var** can be used to declare a variable only when that variable is initialized. Therefore, the following statement is wrong:

```
var counter; // Wrong! Initializer required.
```

Also, remember that **var** can be used only to declare local variables. It cannot be used when declaring instance variables, parameters, or return types, for example.

Local Variable Type Inference with Reference Types

The preceding examples introduced the fundamentals of local variable type inference using primitive types. However, it is with reference types, such as class types, that the full benefits of type inference become apparent. Moreover, local variable type inference with reference types constitutes a primary use of this feature.

Let's again begin with a simple example. The following declarations use type inference to declare two **String** variables called **myStr** and **mySubStr**:

```
var myStr = "This is a string";
var mySubStr = myStr.substring(5, 10);
```

Recall that a quoted string is an object of type **String**. Because a quoted string is used as an initializer, the type of **myStr** is inferred to be **String**. The type of **mySubStr** is also inferred to be **String** because the type of reference returned by the **substring( )** method is **String**.

Of course, you can also use local variable type inference with user-defined classes, as the following program illustrates. It creates a class called **MyClass** and then uses local variable type inference to declare and initialize an object of that class.

```
// Local variable type inference with a user-defined class type.
class MyClass {
  private int i;
```

```
  MyClass(int k) { i = k;}

  int geti() { return i; }
  void seti(int k) { if(k >= 0) i = k; }
}

class VarDemo2 {
  public static void main(String[] args) {
    var mc = new MyClass(10); // Notice the use of var here.

    System.out.println("Value of i in mc is " + mc.geti());
    mc.seti(19);
    System.out.println("Value of i in mc is now " + mc.geti());
  }
}
```

The output of the program is shown here:

```
Value of i in mc is 10
Value of i in mc is now 19
```

In the program, pay special attention to this line:

```
var mc = new MyClass(10); // Notice the use of var here.
```

Here, the type of **mc** will be inferred as **MyClass** because that is the type of the initializer, which is a new **MyClass** object.

As mentioned, one of the primary benefits of local variable type inference is its ability to streamline code, and it is with reference types where such streamlining is most apparent. As you advance in your study of Java, you will find that many class types have rather long names. For example, in Chapter 10 you will learn about the **FileInputStream** class, which is used to open a file for input operations. Without the use of type inference, you would declare and initialize a **FileInputStream** using a traditional declaration like the one shown here:

```
FileInputStream fin = new FileInputStream("test.txt");
```

With the use of **var**, it can now be written like this:

```
var fin = new FileInputStream("test.txt");
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer. There is no need to explicitly repeat the type name. As a result, this declaration of **fin** is substantially shorter than writing it the traditional way. Thus, the use of **var** streamlines the declaration. In general, the streamlining attribute of local variable type inference helps lessen the tedium of entering long type names into your program. Of course, local variable type inference must be used carefully to avoid reducing the readability of your program and thus

obscuring its meaning. In essence, it is a feature that you should use wisely.

Using Local Variable Type Inference in a for Loop

Another place that local variable type inference can be used is in a **for** loop when declaring and initializing the loop control variable inside a traditional **for** loop, or when specifying the iteration variable in a for-each **for**. The following program shows an example of each case:

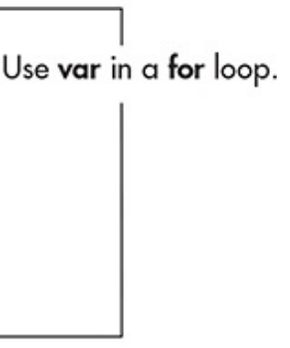
```
// Use type inference in a for loop.
class VarDemo3 {
  public static void main(String[] args) {

    // Use type inference with the loop control variable.
    System.out.print("Values of x: ");
    for(var x = 2.5; x < 100.0; x = x * 2)
      System.out.print(x + " ");

    System.out.println();

    // Use type inference with the iteration variable.
    int[] nums = { 1, 2, 3, 4, 5, 6};
    System.out.print("Values in nums array: ");
    for(var v : nums)
      System.out.print(v + " ");

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



The output is shown here:

```
Values of x: 2.5 5.0 10.0 20.0 40.0 80.0
Values in nums array: 1 2 3 4 5 6
```

In this example, loop control variable **x** in this line:

```
for(var x = 2.5; x < 100.0; x = x * 2)
```

is inferred to be type **double** because that is the type of its initializer. Iteration variable **v** in this line:

```
for(var v : nums)
```

is inferred to be of type **int** because that is the element type of the array **nums**.

Some var Restrictions

In addition to those mentioned in the preceding discussion, several other restrictions apply to the use of **var**. Only one variable can be declared at a time; a variable cannot use **null** as an initializer; and the variable being declared cannot be used by the initializer expression. Although you can declare an array type using **var**, you cannot use **var** with an array initializer. For example, this is valid:

```
var myArray = new int[10]; // This is valid.
```

but this is not:

```
var myArray = { 1, 2, 3 }; // Wrong
```

As mentioned earlier, **var** cannot be used as the name of a class. It also cannot be used as the name of other reference types, including an interface, enumeration, or annotation, which are described later in this book. Here are two other restrictions that relate to Java features also described later, but mentioned here in the interest of completeness. Local variable type inference cannot be used to declare the exception type caught by a **catch** statement. Also, neither lambda expressions nor method references can be used as initializers.

NOTE

At the time of this writing, a number of readers will be using Java environments that predate JDK 10. So that as many of the code examples as possible can be compiled and run with older JDKs, local variable type inference will not be used by most of the programs in the remainder of this edition of the book. Using the full syntax also makes it very clear at a glance what type of variable is being created, which is important for example code. Of course, going forward, you should consider the use of local variable type inference where appropriate in your own code.

The Bitwise Operators

In Chapter 2 you learned about Java's arithmetic, relational, and logical operators. Although these are often the most commonly used, Java provides additional operators that expand the set of problems to which Java can be applied: the bitwise operators. The bitwise operators can be used on values of type **long**, **int**, **short**, **char**, or **byte**. Bitwise operations cannot be used on **boolean**, **float**, or **double**, or class types. They are called the bitwise operators because they are used to test, set, or shift the individual bits that make up a value. Bitwise operations are important to a wide variety of systems-level programming tasks in which status information from a device must be interrogated or constructed. Table 5-1 lists the bitwise operators.

Operator	Result
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Unsigned shift right
<<	Shift left
~	One's complement (unary NOT)

Table 5-1 The Bitwise Operators

The Bitwise AND, OR, XOR, and NOT Operators

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

The bitwise operators AND, OR, XOR, and NOT are **&**, |, ^, and ~. They perform the same operations as their Boolean logical equivalents described in Chapter 2. The difference is that the bitwise operators work on a bit-by-bit basis. The following table shows the outcome of each operation using 1's and 0's:

In terms of one common usage, you can think of the bitwise AND as a way to turn bits off. That is, any bit that is 0 in either operand will cause the corresponding bit in the outcome to be set to 0. For example:

```
  1101 0011
& 1010 1010
  1000 0010
```

The following program demonstrates the **&** by turning any lowercase letter into uppercase by resetting the 6th bit to 0. As the Unicode/ASCII character set is defined, the lowercase letters are the same as the uppercase ones except that the lowercase ones are greater in value by exactly 32. Therefore, to transform a lowercase letter to uppercase, just turn off the 6th bit, as this program illustrates:

```
// Uppercase letters.
class UpCase {
  public static void main(String[] args) {
    char ch;

    for(int i=0; i < 10; i++) {
      ch = (char) ('a' + i);
      System.out.print(ch);

      // This statement turns off the 6th bit.
      ch = (char) ((int) ch & 65503); // ch is now uppercase

      System.out.print(ch + " ");
    }
  }
}
```

The output from this program is shown here:

```
aA bB cC dD eE fF gG hH iI jJ
```

The value 65,503 used in the AND statement is the decimal representation of 1111 1111 1101 1111. Thus, the AND operation leaves all bits in **ch** unchanged except for the 6th one, which is set to 0.

The AND operator is also useful when you want to determine whether a bit is on or off. For example, this statement determines whether bit 4 in **status** is set:

```
if((status & 8)!= 0) System.out.println("bit 4 is on");
```

The number 8 is used because it translates into a binary value that has only the 4th bit set. Therefore, the **if** statement can succeed only when bit 4 of **status** is also on. An interesting use of this concept is to show the bits of a **byte** value in binary format.

```
// Display the bits within a byte.
class ShowBits {
  public static void main(String[] args) {
    int t;
    byte val;

    val = 123;
    for(t=128; t > 0; t = t/2) {
      if((val & t) != 0) System.out.print("1 ");
      else System.out.print("0 ");
    }
  }
}
```

The output is shown here:

```
0 1 1 1 1 0 1 1
```

The **for** loop successively tests each bit in **val**, using the bitwise AND, to determine whether it is on or off. If the bit is on, the digit **1** is displayed; otherwise, **0** is displayed. In Try This 5-3, you will see how this basic concept can be expanded to create a class that will display the bits in any type of integer.

The bitwise OR, as the reverse of AND, can be used to turn bits on. Any bit that is set to 1 in either operand will cause the corresponding bit in the result to be set to 1. For example:

```
  1101 0011
| 1010 1010
  1111 1011
```

We can make use of the OR to change the uppercasing program into a lowercasing program, as shown here:

```
// Lowercase letters.
class LowCase {
  public static void main(String[] args) {
    char ch;

    for(int i=0; i < 10; i++) {
      ch = (char) ('A' + i);
      System.out.print(ch);

      // This statement turns on the 6th bit.
      ch = (char) ((int) ch | 32); // ch is now lowercase

      System.out.print(ch + " ");
    }
  }
}
```

The output from this program is shown here:

```
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

The program works by ORing each character with the value 32, which is 0000 0000 0010 0000 in binary. Thus, 32 is the value that produces a value in binary in which only the 6th bit is set. When this value is ORed with any other value, it produces a result in which the 6th bit is set and all other bits remain unchanged. As explained, for characters this means that each uppercase letter is transformed into its lowercase equivalent.

An exclusive OR, usually abbreviated XOR, will result in a set bit if, and only if, the bits being compared are different, as illustrated here:

```
  0 1 1 1 1 1 1 1
^ 1 0 1 1 1 0 0 1
  1 1 0 0 0 1 1 0
```

The XOR operator has an interesting property that makes it a simple way to encode a message. When some value X is XORed with another value Y, and then that result is XORed with Y again, X is produced. That is, given the sequence

```
R1 = X ^ Y; R2 = R1 ^ Y;
```

then R2 is the same value as X. Thus, the outcome of a sequence of two XORs can produce the original value.

You can use this principle to create a simple cipher program in which some integer is the key that is used to both encode and decode a message by XORing the characters in that message. To encode, the XOR operation is applied the first time, yielding the cipher text. To decode, the XOR is applied a second time, yielding the plain text. Of course, such a cipher

has no practical value, being trivially easy to break. It does, however, provide an interesting way to demonstrate the XOR. Here is a program that uses this approach to encode and decode a short message:

```
// Use XOR to encode and decode a message.
class Encode {
  public static void main(String[] args) {
    String msg = "This is a test";
    String encmsg = "";
    String decmsg = "";
    int key = 88;

    System.out.print("Original message: ");
    System.out.println(msg);

    // encode the message
    for(int i=0; i < msg.length(); i++)
      encmsg = encmsg + (char) (msg.charAt(i) ^ key);

    System.out.print("Encoded message: ");
    System.out.println(encmsg);

    // decode the message
    for(int i=0; i < msg.length(); i++)
      decmsg = decmsg + (char) (encmsg.charAt(i) ^ key);

    System.out.print("Decoded message: ");
    System.out.println(decmsg);
  }
}
```

This constructs the encoded string.

This constructs the decoded string.

Here is the output:

```
Original message: This is a test
Encoded message: 01+x1+x9x,=+,
Decoded message: This is a test
```

As you can see, the result of two XORs using the same key produces the decoded message.

The unary one's complement (NOT) operator reverses the state of all the bits of the operand. For example, if some integer called **A** has the bit pattern 1001 0110, then **~A** produces a result with the bit pattern 0110 1001.

The following program demonstrates the NOT operator by displaying a number and its complement in binary:

```
// Demonstrate the bitwise NOT.
class NotDemo {
  public static void main(String[] args) {
    byte b = -34;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) System.out.print("1 ");
      else System.out.print("0 ");
    }
    System.out.println();

    // reverse all bits
    b = (byte) ~b;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) System.out.print("1 ");
      else System.out.print("0 ");
    }
  }
}
```

Here is the output:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

The Shift Operators

In Java it is possible to shift the bits that make up a value to the left or to the right by a specified amount. Java defines the three bit-shift operators shown here:

<<	Left shift
>>	Right shift
>>>	Unsigned right shift

The general forms for these operators are shown here:

value << *num-bits*
value >> *num-bits*
value >>> *num-bits*

Here, *value* is the value being shifted by the number of bit positions specified by *num-bits.*

Each left shift causes all bits within the specified value to be shifted left one position and a 0 bit to be brought in on the right. Each right shift shifts all bits to the right one position and

preserves the sign bit. As you may know, negative numbers are usually represented by setting the high-order bit of an integer value to 1, and this is the approach used by Java. Thus, if the value being shifted is negative, each right shift brings in a 1 on the left. If the value is positive, each right shift brings in a 0 on the left.

In addition to the sign bit, there is something else to be aware of when right shifting. Java uses *two's complement* to represent negative values. In this approach negative values are stored by first reversing the bits in the value and then adding 1. Thus, the byte value for –1 in binary is 1111 1111. Right shifting this value will always produce –1!

If you don't want to preserve the sign bit when shifting right, you can use an unsigned right shift (>>>), which always brings in a 0 on the left. For this reason, the >>> is also called the *zero-fill* right shift. You will use the unsigned right shift when shifting bit patterns, such as status codes, that do not represent integers.

For all of the shifts, the bits shifted out are lost. Thus, a shift is not a rotate, and there is no way to retrieve a bit that has been shifted out.

Shown next is a program that graphically illustrates the effect of a left and right shift. Here, an integer is given an initial value of 1, which means that its low-order bit is set. Then, a series of eight shifts are performed on the integer. After each shift, the lower 8 bits of the value are shown. The process is then repeated, except that a 1 is put in the 8th bit position, and right shifts are performed.

```
// Demonstrate the shift << and >> operators.
class ShiftDemo {
  public static void main(String[] args) {
    int val = 1;

    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) System.out.print("1 ");
        else System.out.print("0 ");
      }
      System.out.println();
      val = val << 1; // left shift
    }
    System.out.println();

    val = 128;
    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) System.out.print("1 ");
        else System.out.print("0 ");
      }
      System.out.println();
      val = val >> 1; // right shift
    }
  }
}
```

The output from the program is shown here:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
```

```
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

You need to be careful when shifting **byte** and **short** values because Java will automatically promote these types to **int** when evaluating an expression. For example, if you right shift a **byte** value, it will first be promoted to **int** and then shifted. The result of the shift will also be of type **int**. Often this conversion is of no consequence. However, if you shift a negative **byte** or **short** value, it will be sign-extended when it is promoted to **int**. Thus, the high-order bits of the resulting integer value will be filled with ones. This is fine when performing a normal right shift. But when you perform a zero-fill right shift, there are 24 ones to be shifted before the byte value begins to see zeros.

Bitwise Shorthand Assignments

All of the binary bitwise operators have a shorthand form that combines an assignment with the bitwise operation. For example, the following two statements both assign to **x** the outcome of an XOR of **x** with the value 127.

```
x = x ^ 127;
x ^= 127;
```

Ask the Expert

Q: **Since binary is based on powers of two, can the shift operators be used as a shortcut for multiplying or dividing an integer by two?**

A: Yes. The bitwise shift operators can be used to perform very fast multiplication or division by two. A shift left doubles a value. A shift right halves it.

Try This 5-3 A ShowBits Class

`ShowBitsDemo.java`

This project creates a class called **ShowBits** that enables you to display in binary the bit pattern for any integer value. Such a class can be quite useful in programming. For example, if you are debugging device-driver code, then being able to monitor the data stream in binary is often a benefit.

1. Create a file called **ShowBitsDemo.java**.
2. Begin the **ShowBits** class as shown here:

```
class ShowBits {
  int numbits;

  ShowBits(int n) {
    numbits = n;
  }
```

ShowBits creates objects that display a specified number of bits. For example, to create an object that will display the low-order 8 bits of some value, use

```
ShowBits byteval = new ShowBits(8)
```

The number of bits to display is stored in **numbits**.

3. To actually display the bit pattern, **ShowBits** provides the method **show( )**, which is shown here:

```
void show(long val) {
  long mask = 1;

  // left-shift a 1 into the proper position
  mask <<= numbits-1;

  int spacer = 0;
  for(; mask != 0; mask >>>= 1) {
    if((val & mask) != 0) System.out.print("1");
    else System.out.print("0");
    spacer++;
    if((spacer % 8) == 0) {
      System.out.print(" ");
      spacer = 0;
    }
  }
  System.out.println();
}
```

Notice that **show( )** specifies one **long** parameter. This does not mean that you always have to pass **show( )** a **long** value, however. Because of Java's automatic type promotions, any integer type can be passed to **show( )**. The number of bits displayed is determined by the value stored in **numbits**. After each group of 8 bits, **show( )** outputs a space. This makes it easier to read the binary values of long bit patterns.

4. The **ShowBitsDemo** program is shown here:

```
/*
    Try This 5-3
    A class that displays the binary representation of a value.
*/

class ShowBits {
  int numbits;

  ShowBits(int n) {
    numbits = n;
  }

  void show(long val) {
    long mask = 1;

    // left-shift a 1 into the proper position
    mask <<= numbits-1;

    int spacer = 0;
    for(; mask != 0; mask >>>= 1) {
      if((val & mask) != 0) System.out.print("1");
      else System.out.print("0");
      spacer++;
      if((spacer % 8) == 0) {
        System.out.print(" ");
        spacer = 0;
      }
    }
    System.out.println();
  }
}

// Demonstrate ShowBits.
class ShowBitsDemo {
  public static void main(String[] args) {
    ShowBits b = new ShowBits(8);
    ShowBits i = new ShowBits(32);
    ShowBits li = new ShowBits(64);
```

```
        System.out.println("123 in binary: ");
        b.show(123);

        System.out.println("\n87987 in binary: ");
        i.show(87987);

        System.out.println("\n237658768 in binary: ");
        li.show(237658768);


        // you can also show low-order bits of any integer
        System.out.println("\nLow order 8 bits of 87987 in binary: ");
        b.show(87987);
      }
    }
```

5. The output from **ShowBitsDemo** is shown here:

```
123 in binary:
01111011

87987 in binary:
00000000 00000001 01010111 10110011

237658768 in binary:
00000000 00000000 00000000 00000000 00001110 00101010 01100010
10010000

Low order 8 bits of 87987 in binary:
10110011
```

The ? Operator

One of Java's most fascinating operators is the **?**. The **?** operator is often used to replace **if-else** statements of this general form:

```
if (condition)
  myVar = expression1;
else
  myVar = expression2;
```

Here, the value assigned to *myVar* depends upon the outcome of the condition controlling the **if**.

The **?** is called a *ternary operator* because it requires three operands. It takes the general form

Exp1 ? *Exp2* : *Exp3*;

where *Exp1* is a **boolean** expression, and *Exp2* and *Exp3* are expressions of any type other than **void**. The type of *Exp2* and *Exp3* must be the same (or compatible), though. Notice the use and placement of the colon.

The value of a **?** expression is determined like this: *Exp1* is evaluated. If it is true, then *Exp2* is evaluated and becomes the value of the entire **?** expression. If *Exp1* is false, then *Exp3* is evaluated and its value becomes the value of the expression. Consider this example, which assigns **absval** the absolute value of **val**:

```
absval = val < 0 ? -val : val; // get absolute value of val
```

Here, **absval** will be assigned the value of **val** if **val** is zero or greater. If **val** is negative, then **absval** will be assigned the negative of that value (which yields a positive value). The same code written using the **if-else** structure would look like this:

```
if(val < 0) absval = -val;
else absval = val;
```

Here is another example of the **?** operator. This program divides two numbers, but will not allow a division by zero.

```
// Prevent a division by zero using the ?.
class NoZeroDiv {
  public static void main(String[] args) {
    int result;

    for(int i = -5; i < 6; i++) {
      result = i != 0 ? 100 / i : 0; <-- This prevents a divide-by-zero.
      if(i != 0)
        System.out.println("100 / " + i + " is " + result);
    }
  }
}
```

The output from the program is shown here:

```
100 / -5 is -20
100 / -4 is -25
100 / -3 is -33
100 / -2 is -50
100 / -1 is -100
100 / 1 is 100
100 / 2 is 50
100 / 3 is 33
100 / 4 is 25
100 / 5 is 20
```

Pay special attention to this line from the program:

```
result = i != 0 ? 100 / i : 0;
```

Here, **result** is assigned the outcome of the division of 100 by **i**. However, this division takes place only if **i** is not zero. When **i** is zero, a placeholder value of zero is assigned to **result**.

You don't actually have to assign the value produced by the **?** to some variable. For example, you could use the value as an argument in a call to a method. Or, if the expressions are all of type **boolean**, the **?** can be used as the conditional expression in a loop or **if** statement. For example, here is the preceding program rewritten a bit more efficiently. It produces the same output as before.

```
// Prevent a division by zero using the ?.
class NoZeroDiv2 {
  public static void main(String[] args) {

    for(int i = -5; i < 6; i++)
      if(i != 0 ? true : false)
        System.out.println("100 / " + i +
                           " is " + 100 / i);
  }
}
```

Notice the **if** statement. If **i** is zero, then the outcome of the **if** is false, the division by zero is prevented, and no result is displayed. Otherwise, the division takes place.

✓ *Chapter 5 Self Test*

1. Show two ways to declare a one-dimensional array of 12 **double**s.
2. Show how to initialize a one-dimensional array of integers to the values 1 through 5.
3. Write a program that uses an array to find the average of 10 **double** values. Use any 10 values you like.

4. Change the sort in Try This 5-1 so that it sorts an array of strings. Demonstrate that it works.

5. What is the difference between the **String** methods **indexOf( )** and **lastIndexOf( )**?

6. Since all strings are objects of type **String**, show how you can call the **length( )** and **charAt( )** methods on this string literal: "I like Java".

7. Expanding on the **Encode** cipher class, modify it so that it uses an eight-character string as the key.

8. Can the bitwise operators be applied to the **double** type?

9. Show how this sequence can be rewritten using the **?** operator.

```
if(x < 0) y = 10;
else y = 20;
```

10. In the following fragment, is the **&** a bitwise or logical operator? Why?

```
boolean a, b;
// ...
if(a & b) ...
```

11. Is it an error to overrun the end of an array? Is it an error to index an array with a negative value?

12. What is the unsigned right-shift operator?

13. Rewrite the **MinMax** class shown earlier in this chapter so that it uses a for-each style **for** loop.

14. Can the **for** loops that perform sorting in the **Bubble** class shown in Try This 5-1 be converted into for-each style loops? If not, why not?

15. Can a **String** control a **switch** statement?

16. What keyword is reserved for use with local variable type inference?

17. Show how to use local variable type inference to declare a **boolean** variable called **done** that has an initial value of **false**.

18. Can **var** be the name of a variable? Can **var** be the name of a class?

19. Is the following declaration valid? If not, why not.

```
var[] avgTemps = new double[7];
```

20. Is the following declaration valid? If not, why not?

```
var alpha = 10, beta = 20;
```

21. In the **show( )** method of the **ShowBits** class developed in Try This 5-3, the local

variable **mask** is declared as shown here:

```
long mask = 1;
```

Change this declaration so that it uses local variable type inference. When doing so, be sure that **mask** is of type **long** (as it is here), and not of type **int**.