CHAPTER 7

ARRAYS: SETS OF SIMILAR DATA ITEMS

Computers process information and usually they need to process masses of information. In previous chapters we have studied programs that contain a few variables where information is stored. These kinds of programs are not sufficient to handle practical information processing tasks. Therefore, programming languages provide data structures that can store larger quantities of information.

Arrays are data structures in which it is possible to store many similar data items. Arrays can thus hold large amounts of information. In this chapter we shall learn how traditional arrays are declared, created, and used in Java. A traditional array is such that it can store data items of certain type (e.g. `int`, `long`, `byte`, or `double`). A traditional array has a certain length which specifies how many data items it can store.

7.1 Creating arrays and referring to array elements

Arrays are collections of similar data items, such as integers. We usually need an array when we want to handle many similar data items in a single program. Everything that can be stored in a variable can also be stored in an array. The difference between a variable and an array is that, while a variable stores exactly one data item, an array can hold hundreds of similar data items.

The declaration of an array is rather complex when we compare it to the declaration of a variable. A variable of type **int** can be declared by writing

```
int  some_name ;
```

but to declare and create an array that can hold 5 values of type **int**, we must write

```
int[]  some_name  =  new  int[ 5 ] ;
```

Both declarations above begin with the keyword **int** to indicate that we are declaring something that can hold integer values. In the array declaration a pair of empty brackets [] after the keyword **int** tells the Java compiler that now we are declaring an array, and not a variable. The expression to the right of the assignment operator = actually creates the array and reserves necessary memory space for it. When we write

```
new  int[ 5 ]
```

we specify that new memory space must be reserved to store 5 data items of type **int**. An array like the one above is said to have 5 array elements or positions, and the length of the array is 5. The storage capacity of this array is the same as the capacity of five separate variables of type **int**.

Arrays are used in the same way as variables. First we declare an array and then we store information in it. Storing information in a variable is simple because a variable holds only one piece of information at a time. An assignment statement can store a value in a variable, for example, in the following way

```
some_integer_variable  =  7  ;
```

Storing a value in an array is more complex because we have to somehow specify exactly where in the array we want the value to be stored. To store a number in an array, we need to use an index value which dictates in which element in the array the number will be stored. Let us suppose that we have the following array of integers created

```
int[] array_of_integers  =  new  int[ 50 ] ;
```

This array can be used to store up to 50 integer values. At the beginning, when the array is created, it does not contain meaningful data. We can refer to some array element by writing an index value in brackets after the array name. This way numbers can be stored in an array. For example, the following assignment statements write numbers 7 and 77 to the first and second element in the above array

```
array_of_integers[ 0 ]  =   7 ;
array_of_integers[ 1 ]  =  77 ;
```

Index values start counting from zero, not from one. When we want to refer to the first element of an array, we use index 0. The index of the second element is 1, the index of the third element is 2, and so on. The maximum index that the above array of integers can have is 49, one less than the defined array length. You should note that the number inside brackets has a different meaning in array creation and in a reference to an array element. In an array creation the number means the array length. In a reference to an array element, the number specifies which array element is going to be affected.

The above array of integers has a storage capacity that is equal to 50 variables of type **int**. Because a variable of type **int** requires 4 bytes (32 bits) of memory space, the above array requires 200 bytes (50 times 4 bytes) of memory. We can think that the array

is a 200-byte memory area that is divided into 50 slots, and each 4-byte slot can hold an **int** value. By writing **array_of_integers[0]** we can refer to the first slot or position in the array. **array_of_integers[49]** refers to the last position in the array. When an array is allocated memory space from a computer's main memory, we do not need to know the exact memory address of the array. We can, however, think that the first array position has the smallest numerical memory address, and the last position has the largest numerical memory address.

Program **ArrayDemo.java** shows different ways to assign values to the elements of an array whose type is **int[]**. The array in program **ArrayDemo.java** has 50 array elements which are filled with various integer values. You can see that a loop is the most efficient way to fill an array. Usually, **for** loops are used when entire arrays are processed. Figure 7-1 shows what the array of program **ArrayDemo.java** looks like in a computer's main memory after the program has been executed.

An index variable, such as **integer_index** in **ArrayDemo.java**, is the most efficient and common way to access the elements of an array. With an index variable we specify which position of an array we are currently accessing. An index variable is usually initialized before the processing of an array begins. During the processing of an array in a loop, the index variable is either incremented or decremented inside the loop. To understand how index variables work, let us suppose that we have made the following declarations:

```
int[] array_of_integers  =  new  int[ 50 ] ;
int   integer_index  =  4 ;
```

The index variable **integer_index** having been initialized with the value 4,

- **array_of_integers[integer_index]** refers to the 5th array element in the array of integers,
- **array_of_integers[integer_index - 1]** refers to the 4th element,
- **array_of_integers[integer_index + 2]** refers to the 7th element, and
- **array_of_integers[integer_index * 2]** refers to the 9th element.

The above examples demonstrate that it is possible to write an arithmetic expression inside the brackets in an array reference.When we use an index variable, it is easy to move to the following or previous element in an array by incrementing or decrementing the index variable by one. An array reference that contains an arithmetic expression does not alter the value of the index variable. In the array reference **array_of_integers[integer_index + 1]** the value of **integer_index** is not modified. The program calculates the value of **integer_index + 1** (i.e. the program evaluates expression **integer_index + 1**), but that value is discarded once the array reference is processed.

Exercises with program ArrayDemo.java

Exercise 7-1. Written twice in the program is "+ 2". How would the behavior of the program change if the plus sign and number 2 were taken away from the program?

Exercise 7-2. Modify the first **for** loop so that values 8, 11, 14, 17, 20, 23, 26, 29, etc. will be written to those array positions which currently hold values 7, 9, 11, 13, 15, 17, 19, 21, etc. This is not a large modification.

Exercise 7.3. What would happen if the beginning of the second **for** loop was written like

```
for ( integer_index  =  1 ;
      integer_index  <  29 ;
      integer_index  =  integer_index + 2 )
```

Exercise 7-4. Modify the second **for** loop, or replace it with a **while** loop, so that the loop prints out the integers from the array in reverse order. Studying program **Reverse.java** may help in this task.

The first three elements in the array are assigned values 333, 33, and 3.

The fourth element in the array gets a value that is the value of the third element plus 2. As the index value of an array element is always one less than the "serial number" of the element, the fourth element is accessed with index 3, and the third element with index 2.

```
//  ArrayDemo.java

class ArrayDemo
{
   public static void main( String[] not_in_use )
   {
      int[]  array_of_integers  =  new  int[ 50 ] ;

      int  integer_index ;

      array_of_integers[ 0 ]  =  333 ;
      array_of_integers[ 1 ]  =   33 ;
      array_of_integers[ 2 ]  =    3 ;
      array_of_integers[ 3 ]  =  array_of_integers[ 2 ]  +  2 ;

      for ( integer_index  =  4 ;
            integer_index  <  50 ;
            integer_index  ++ )
      {
         array_of_integers[ integer_index ]  =
                array_of_integers[ integer_index - 1 ]  +  2 ;
      }

      System.out.print( "\n The contents of \"array_of_integers\" is:\n" ) ;

      for ( integer_index  =  0 ;
            integer_index  <  50 ;
            integer_index  ++ )
      {
         if ( ( integer_index  %  10 )  ==  0 )
         {
            System.out.print( "\n" ) ;
         }

         System.out.printf( "%5d", array_of_integers[ integer_index ] ) ;
      }
   }
}
```

ArrayDemo.java - 1.+ A program that demonstrates the use of an array.

```
D:\javafiles2>java ArrayDemo

 The contents of "array_of_integers" is:

  333   33    3    5    7    9   11   13   15   17
   19   21   23   25   27   29   31   33   35   37
   39   41   43   45   47   49   51   53   55   57
   59   61   63   65   67   69   71   73   75   77
   79   81   83   85   87   89   91   93   95   97
```

ArrayDemo.java - X. 10 array elements are printed on each row.

This **for** loop writes numbers to array positions with index values from 4 to 49. As variable **integer_index** is assigned value 4 at the beginning, the 5th element in the array will be written first.

The value of variable **integer_index** is incremented by one always after the internal statements of the **for** loop have been executed. The variable will reach a value of 50 but that value is not used because the loop terminates.

```
for ( integer_index  =  4 ;
      integer_index  <  50 ;
      integer_index  ++ )
{
   array_of_integers[ integer_index ]  =
         array_of_integers[ integer_index - 1 ]  +  2 ;
}
```

Here the index is an arithmetic expression. This means that the value of this expression is used as an index to access the array. The computer calculates how much is "**integer_index** minus one" before accessing the array. The value of **integer_index** is not modified in this operation.

This statement means that the array element currently referred to with **integer_index** gets a value that is the value of the preceding array element plus 2. By writing **integer_index - 1** it is possible to refer to the preceding element in the array. Because the value of **array_of_integers[3]** is 5, the values of subsequent elements will be 7, 9, 11, 13, 15, etc.

ArrayDemo.java - 1 - 1: The for loop that writes array elements with indexes from 4 to 49.

Usually, when we want to go through an entire array in a **for** loop, the first index is zero, and the upper limit is the array length, which is 50 in this case. The value 50 is never used in array access because the operator <, "less than", terminates the loop with that value.

This **if** construct ensures that a newline character is printed after ten values from the array have been printed. Here the remainder operator % returns the remainder for the case where the **integer_index** is divided by 10. A newline will be printed if the remainder is zero. This results in a newline being printed with **integer_index** values 0, 10, 20, 30, and 40. The use of operator % does not affect the value of **integer_index**.

```
for ( integer_index  =  0 ;
      integer_index  <  50 ;
      integer_index  ++ )
{
   if ( ( integer_index  %  10 )  ==  0 )
   {
      System.out.print( "\n" ) ;
   }

   System.out.printf( "%5d", array_of_integers[ integer_index ] ) ;
}
```

Method **printf()** prints the values of the array elements. This format specifier stipulates that the values are printed right-justified to a printing field that is 5 character positions wide. This ensures that all numbers end in the same column on the screen, regardless of how many digits the numbers may have.

ArrayDemo.java - 1 - 2. The for loop that prints the entire array to the screen.

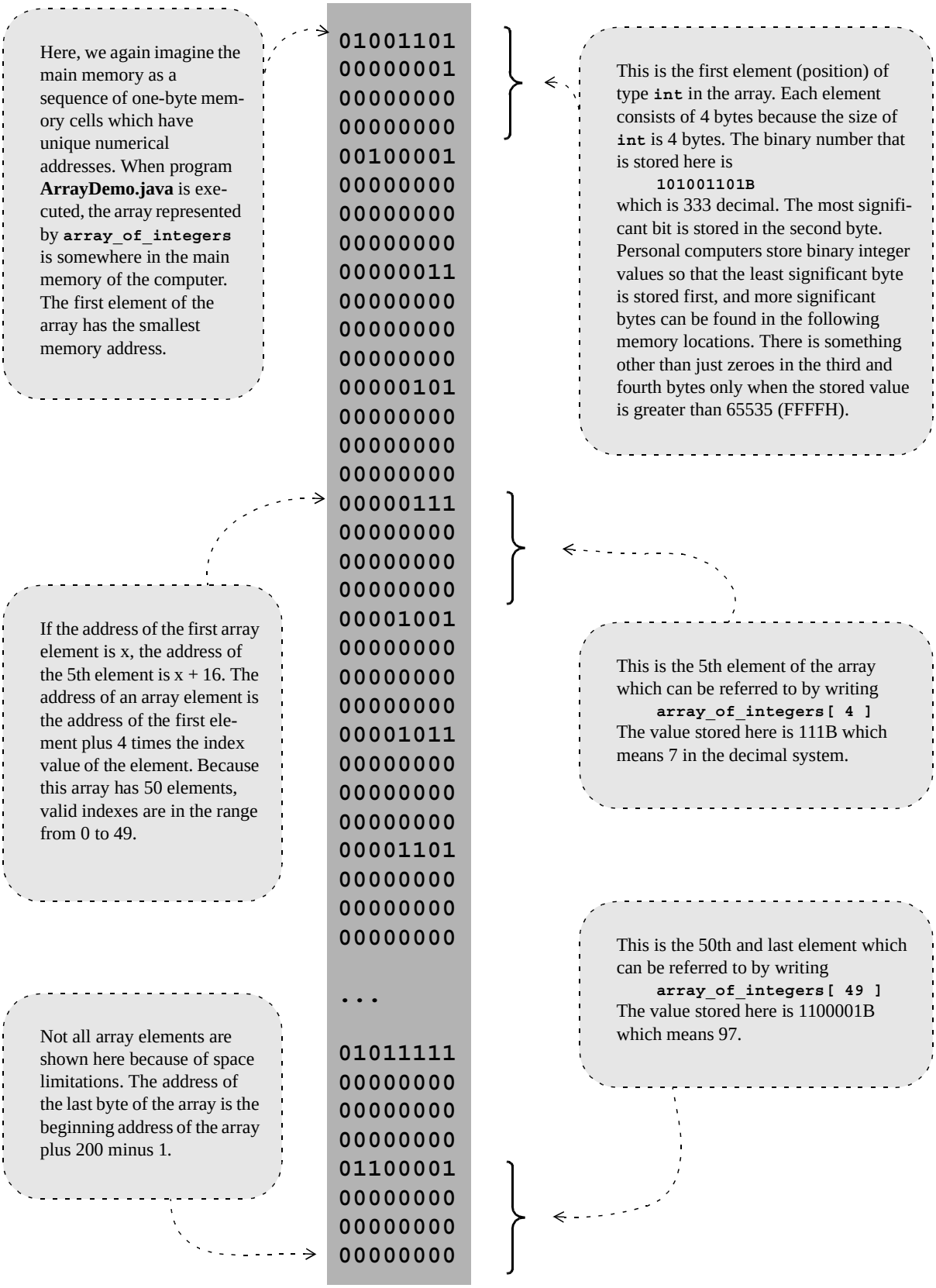


Figure 7-1. The array of integers in the main memory after the execution of ArrayDemo.java.

Program **Reverse.java** is another example of using an array of integers. This program reads integers from the keyboard and stores the read numbers in an array. The reading loop terminates when the user types in a zero. The reading loop is followed by another loop that outputs the integers from the array so that the integers are printed in reverse order. This is achieved by decrementing the index variable towards zero. Arrays are used in programs when many similar data items need to be stored and manipulated. Program **Reverse.java** has to store all the input integers because it needs to remember them all in order to print them in reverse order.

When we create an array, we must specify a length for the array. For example, when we write

```
int[] array_of_integers  =  new  int[ 50 ] ;
```

we create an array whose length is 50, i.e., this array has 50 different positions for storing **int** values. The length of an array cannot be modified after the array is created.

The length of an array dictates which are the legal index values when we refer to the individual array elements. For an array whose length is 50, the legal index values are in the range from 0 to 49. The Java compiler can compile a program in which an illegal index value is used, but the program cannot be executed. For example, a program containing the statements

```
int[] array_of_integers  =  new  int[ 50 ] ;
array_of_integers[ 52 ]  =   88 ;
array_of_integers[ 99 ]  =  888 ;
```

can be compiled, but it cannot be successfully executed because the index values 52 and 99 are larger than the largest legal index value 49.

Program **Reverse.java** does not have any protection against the use of too large index values. If you had the patience to type in more than 100 integers into **Reverse.java**, the program would terminate with the text

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 100
```

appearing on the screen. **Reverse.java** was written without any protection against the use of too large index values because it is easier at first to study simple programs. However, when you write programs that have purposes other than educational, you must take care that array indexes are not growing too large.

Program **MeanvalueArray.java** is an example of a safer program that ensures that the index variable **number_index** does not exceed the length of the used array. The **while** loop that writes data to the array is equipped with the boolean expression

```
number_index  <  array_of_numbers.length
```

which terminates the loop if **number_index** becomes too large. By writing **.length** in the above boolean expression we can find out what length was specified for the array when it was created.

length is a data field that is associated with every array. This data field stores the length of an array and it can be read by using the dot operator . in the following way

```
array_name.length
```

The **length** field provides an integer value that tells what is the length of the array in question. This integer value can be used in many ways. For example, the program lines

```
int[] some_array  =  new  int[ 33 ] ;
System.out.print( "Array length is " + some_array.length ) ;
```

would produce the following line to the screen

```
Array length is 33
```

Here we declare and create an array that has space to store 100 **int** values. This array reserves 400 bytes of memory. **integer_index** is a variable that is used to index the array. With initial value 0, **integer_index** refers to the first position in the array.

The value of variable **integer_from_keyboard** is copied to an array position specified by the value of variable **integer_index**. As the value of **integer_index** is incremented by one after this statement, the next integer will be written to the next position in the array.

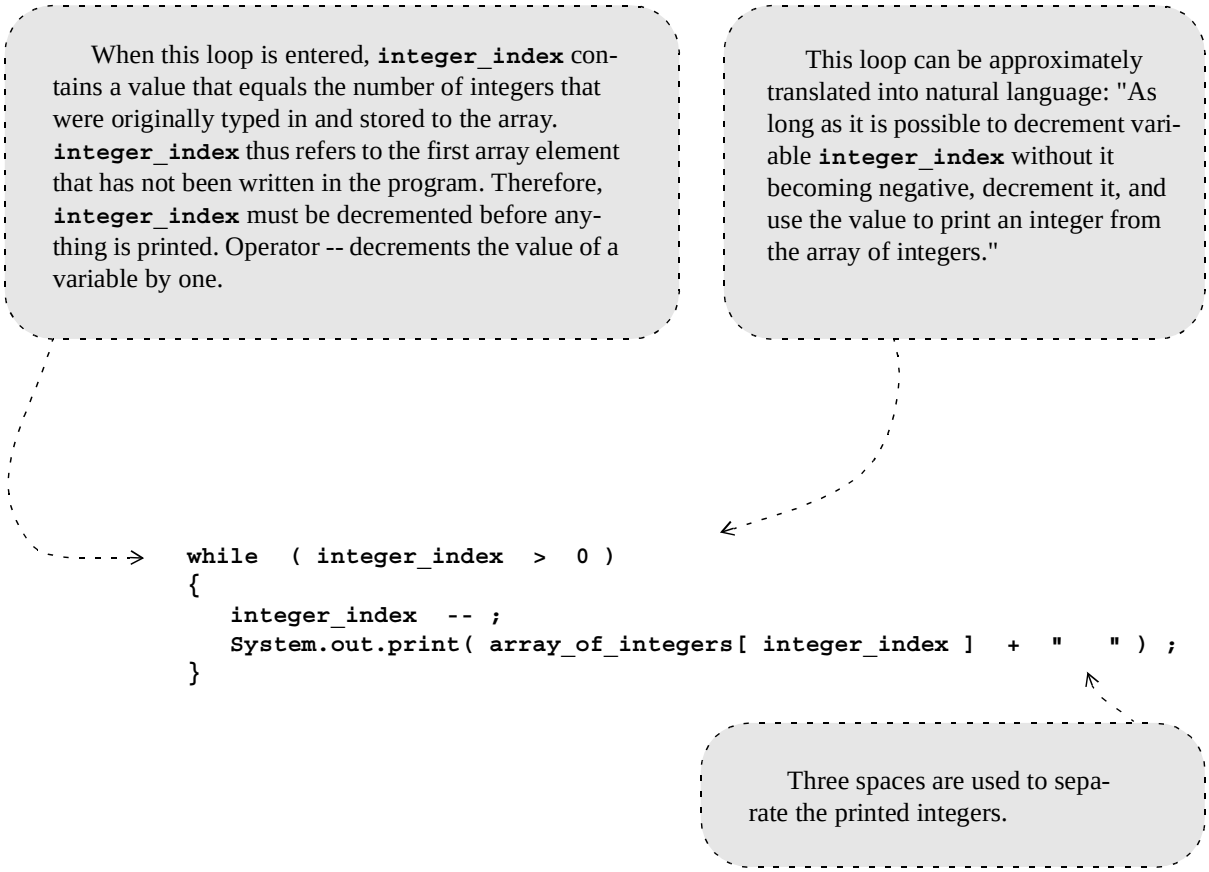
```
//  Reverse.java  (c) Kari Laitinen

import java.util.* ;

class Reverse
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      int[] array_of_integers      =  new  int[ 100 ] ;
      int   integer_index          =  0 ;
      int   integer_from_keyboard  =  0 ;

      System.out.print("\n This program reads integers from the keyboard."
                +  "\n After receiving a zero, it prints the numbers"
                +  "\n in reverse order. Please, start entering numbers."
                +  "\n The program will stop when you enter a zero.\n\n") ;
      do
      {
         System.out.print( " "  +  integer_index  +  "  Enter an integer: ") ;
         integer_from_keyboard  =  keyboard.nextInt() ;

         array_of_integers[ integer_index ]  =  integer_from_keyboard ;
         integer_index  ++ ;
      }
       while  ( integer_from_keyboard  !=  0 ) ;

      System.out.print( "\n Reverse order:  " ) ;

      while  ( integer_index  >  0 )
      {
         integer_index  -- ;
         System.out.print( array_of_integers[ integer_index ]  +  "   " ) ;
      }
   }
}
```

This **while** loop prints the contents of the array to the screen. Because **integer_index** is decremented inside the loop, the numbers are printed in reverse order.

This **do-while** loop terminates when the user of the program types in a zero. Also the zero is written to the array. When this loop terminates, **integer_index** has a value that is exactly the same as the number of integers written to the array. The zero is included in that count.

Remember that **do-while** loops execute at least once. This program works also when the user enters nothing but the zero.

Reverse.java - 1.+ A program that inputs integers and prints them in reverse order.



Reverse.java - 1 - 1. The while loop that prints the array in reverse order.

```
D:\javafiles2>java Reverse

 This program reads integers from the keyboard.
 After receiving a zero, it prints the numbers
 in reverse order. Please, start entering numbers.
 The program will stop when you enter a zero.

 0  Enter an integer: 22
 1  Enter an integer: 33
 2  Enter an integer: 444
 3  Enter an integer: 555
 4  Enter an integer: 6666
 5  Enter an integer: 7777
 6  Enter an integer: 88
 7  Enter an integer: 99
 8  Enter an integer: 0

 Reverse order:  0   99   88   7777   6666   555   444   33   22
```

Reverse.java - X. Here the program is executed with 9 integers.

An array is created here so that its length is 100. The array is thus capable of holding 100 different values of type **double**. Later, the **length** field is used to refer to the previously specified length of the array.

```
//  MeanvalueArray.java

import java.util.* ;

class MeanvalueArray
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;
      System.out.print( "\n This program calculates the mean value of"
                     +  "\n the numbers you enter from the keyboard."
                     +  "\n The program stops when you enter a letter."
                     +  "\n\n   Enter a number: " ) ;

      double[]  array_of_numbers  =  new double[ 100 ] ;
      int       number_index   =  0  ;
      boolean  keyboard_input_is_numerical  =  true ;

      while ( keyboard_input_is_numerical ==  true  &&
              number_index  <  array_of_numbers.length  )
      {
         try
         {
            double  number_from_keyboard  =  keyboard.nextDouble() ;

            array_of_numbers[ number_index ]  =  number_from_keyboard ;
            number_index  ++ ;
            System.out.print(  "   Enter a number: " ) ;
         }
         catch ( Exception  not_numerical_input_exception )
         {
            keyboard_input_is_numerical  =  false ;
         }
      }

      int  number_of_numbers_in_array  =  number_index ;
      double  sum_of_numbers  =  0 ;

      for (  number_index  =  0 ;
             number_index  <  number_of_numbers_in_array ;
             number_index  ++ )
      {
         sum_of_numbers  =  sum_of_numbers  +
                            array_of_numbers[ number_index ] ;
      }

      double  mean_value  =  0 ;

      if ( number_of_numbers_in_array  >  0 )
      {
         mean_value  =  sum_of_numbers /
                        (double) number_of_numbers_in_array ;
      }

      System.out.print( "\n The mean value is: " + mean_value + " \n" ) ;
   }
}
```

MeanvalueArray.java - 1.+ An improved version of program MeanvalueException.java.

As in program **MeanvalueException.java**, numerical values are read from the keyboard inside a **try-catch** construct. When method **nextDouble()** detects that non-numerical data were read from the keyboard, the **catch** block is executed.

The boolean expression of this **while** loop consists of two subexpressions. The latter part of the boolean expression takes care that too large index values are not used. The length of the array was defined to be 100 at the beginning of the program. If **number_index** reaches this value, the loop terminates. The logical-AND operator && combines the two parts of the boolean expression.

```
while ( keyboard_input_is_numerical ==  true  &&
        number_index  <  array_of_numbers.length  )
{
   try
   {
      double  number_from_keyboard  =  keyboard.nextDouble() ;

      array_of_numbers[ number_index ]  =  number_from_keyboard ;
      number_index  ++ ;
      System.out.print(  "   Enter a number: " ) ;
   }
   catch ( Exception  not_numerical_input_exception )
   {
      keyboard_input_is_numerical  =  false ;
   }
}
```

This statement is executed when the user of the program types in something else than valid numbers. When the boolean variable is given the value **false**, the loop terminates.

This statement copies the number typed in from the keyboard into the array position determined by the value of **number_index**.

MeanvalueArray.java - 1 -1. The while loop that inputs numbers from the keyboard.

```
D:\javafiles2>java MeanvalueArray

 This program calculates the mean value of
 the numbers you enter from the keyboard.
 The program stops when you enter a letter.

   Enter a number: 1040.609
   Enter a number: 2030.456
   Enter a number: 2345
   Enter a number: 2346.789
   Enter a number: 3344.99
   Enter a number: z

 The mean value is: 2221.5688
```

MeanvalueArray.java - X. Calculating the mean value of five input numbers.

Program **MeanvalueArray.java** is an improved version of programs **Meanvalue.java** and **MeanvalueException.java** which we studied in Chapter 6. These programs are able to calculate mean values of integers, but **MeanvalueArray.java** is also able to handle floating-point numbers and even the zero. The array in **MeanvalueArray.java** is of type `double`, the double-precision floating-point type. Although the array is a floating-point array, it is indexed with an `int` variable.

Every variable type in Java can also be the type of an array (see Figure 7-2). All types of arrays are indexed in the same way, using indexes of type `int`. The arrays

```
int[]    array_of_integers    =  new  int[ 50 ] ;
float[]  array_of_floats      =  new  float[ 50 ] ;
double[] array_of_numbers     =  new  double[ 50 ] ;
char[]   array_of_characters  =  new  char[ 50 ] ;
```

are all acceptable to the Java compiler. The length of all these arrays is 50, but they would need different amounts of memory. The array of type `float[]` would need 200 bytes of memory because a variable of type `float` needs 4 bytes. The array of type `char[]` would need only 100 bytes because a variable of type `char` is a two-byte variable.

When we use arrays in our programs, we usually need index variables to access the array. In this book the index variables are mostly named so that the name of the index variable indicates what is stored in the array being indexed. For example, if an array stores integers, the name of the index variable is `integer_index`; arrays that store other kinds of numbers are indexed with a variable named `number_index`; an array of type `char[]` is indexed with `character_index`; and so on.

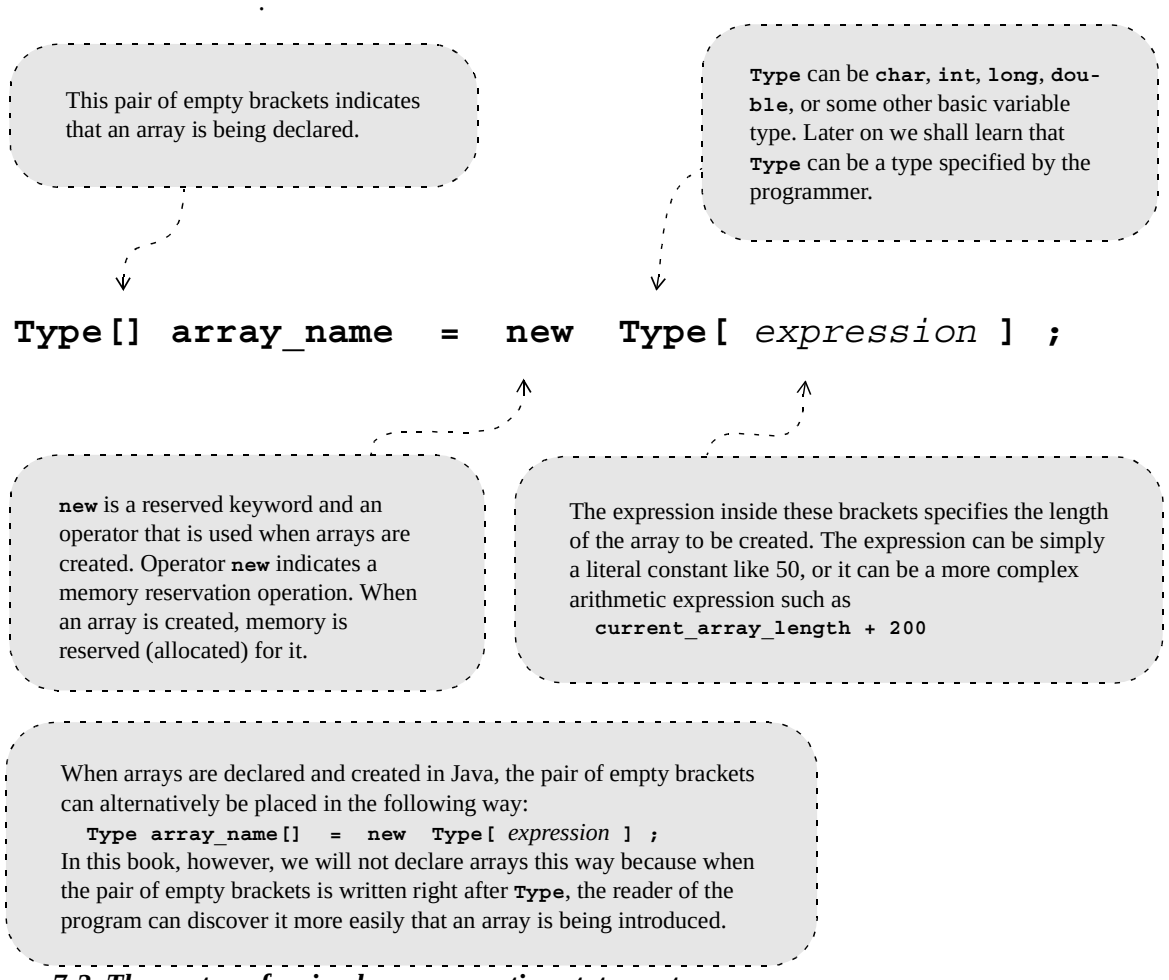


Figure 7-2. The syntax of a simple array creation statement.

7.2 Array declaration vs. array creation

In the example programs of the preceding section, arrays were created with statements such as

```
int[] array_of_integers  =  new  int[ 50 ] ;
```

This statement actually includes two separate operations: the declaration of an array and the creation of an array. The above statement can be replaced with the two statements

```
int[] array_of_integers ;

array_of_integers  =  new  int[ 50 ] ;
```

of which the first statement declares an array and the latter statement creates an array. The array declaration statement

```
int[] array_of_integers ;
```

specifies an array reference that can refer to or point to an array that will be created later. The pair of empty brackets informs the compiler that here we are declaring an array, and not a variable. Keyword **`int`** in the declaration stipulates that the array can contain only values of type **`int`**.

The array creation statement

```
array_of_integers  =  new  int[ 50 ] ;
```

actually creates the array by reserving necessary memory area for it. An array of type **`int[]`** whose length is 50 needs a memory area whose size is equal to the memory needed by 50 separate variables of type **`int`**.

When a Java program is being run on a computer, the program uses separate memory areas from the main memory in the following way:

- The program needs "program memory" for itself. The Java interpreter (the Java virtual machine) stores the executable Java program from a **.class** file to a memory area which we can call "program memory". (An executable Java program consists of bytecode instructions which are processed by the Java interpreter. The Java interpreter is itself a "real" computer program consisting of executable machine instructions.)
- The stack memory is a memory area from which memory space is allocated for small data items such as variables.
- The heap memory is a memory area from which the program can reserve memory space for large pieces of data. The large pieces of data include arrays and other large objects that we'll study later in this book.

When an array is declared and created in a Java program, both the stack memory and the heap memory are needed. These separate memory areas, that reside "somewhere" in the main memory, are used so that an array creation statement reserves memory space from the heap memory, and an array declaration statement reserves four bytes from the stack memory. The four bytes that are reserved from the stack memory are able to store a 32-bit memory address, the address of the memory space in the heap memory. Figure 7-3 describes the usage of these two memory areas.

When we speak about the separate memory areas in the main memory of a computer, we can only say that they reside "somewhere" in the main memory. We do not need to know exactly where these memory areas are. The operating system of the computer manages these memory areas, and takes care that program executing on the computer can access them. We, programmers, need to understand how these memory areas are used when we specify different kinds of data constructs in our programs.

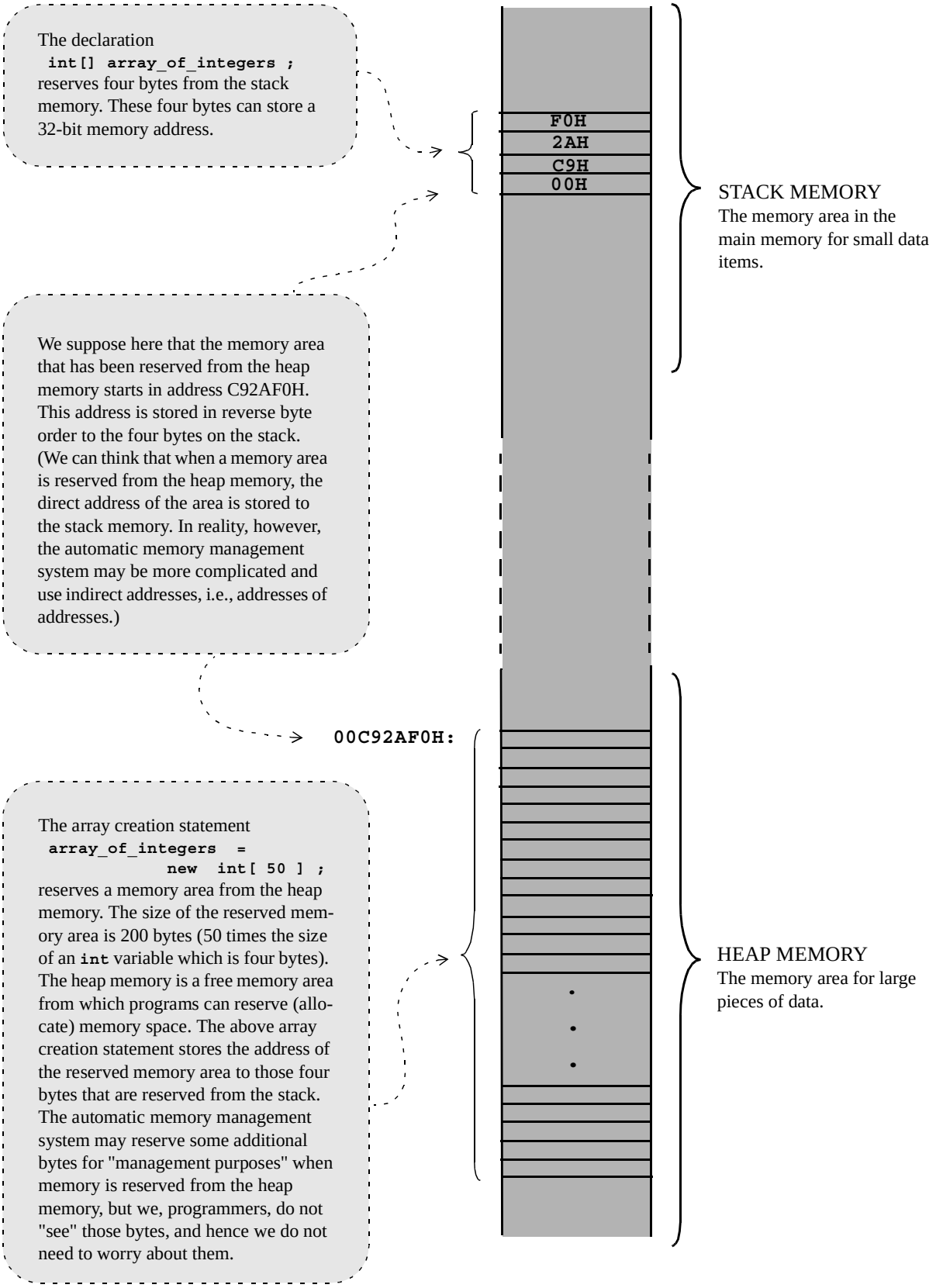


Figure 7-3. Memory used by an array.