



D06

PROGRAMMING with JAVA

Ch8 – Arrays and Array Lists

Chapter Goals

- To become familiar with using **arrays** and **array lists**
- To learn about **wrapper classes**, **auto-boxing** and the **generalized for** loop
- To study common array algorithms
- To learn how to use two-dimensional arrays
- To understand when to choose array lists and arrays in your programs
- To implement partially filled arrays

Arrays

- In many programs, you need to manipulate collections of related values. It would be impractical to use a sequence of variables such as `data1`, `data2`, `data3`, ..., and so on. The **array** construct provides a better way of storing a collection of values
- An **array** is a sequence of values of the same type. The number of elements is called the **length** of the array



To construct an array use the **new** operator:

```
new double[10]
```

- You can store a reference to the array in a variable. The type of an array variable is the element type, followed by squared brackets **[]**:

```
double[] data = new double[10];
```

```
BankAccount[] accounts = new BankAccount[10];
```



When an array is created, all values are initialized depending on the array type:

- Numbers: **0**
- Boolean: **false**
- Object References: **null**

Continued...

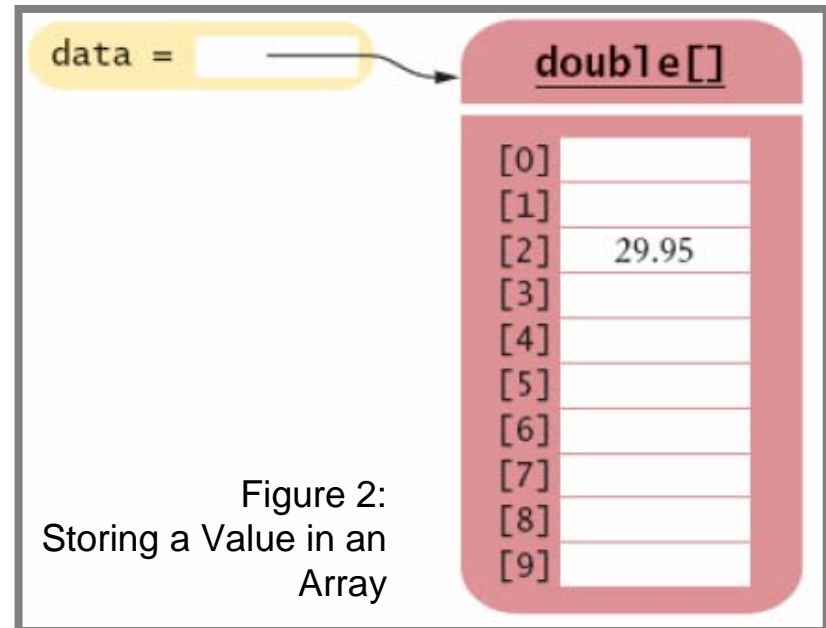
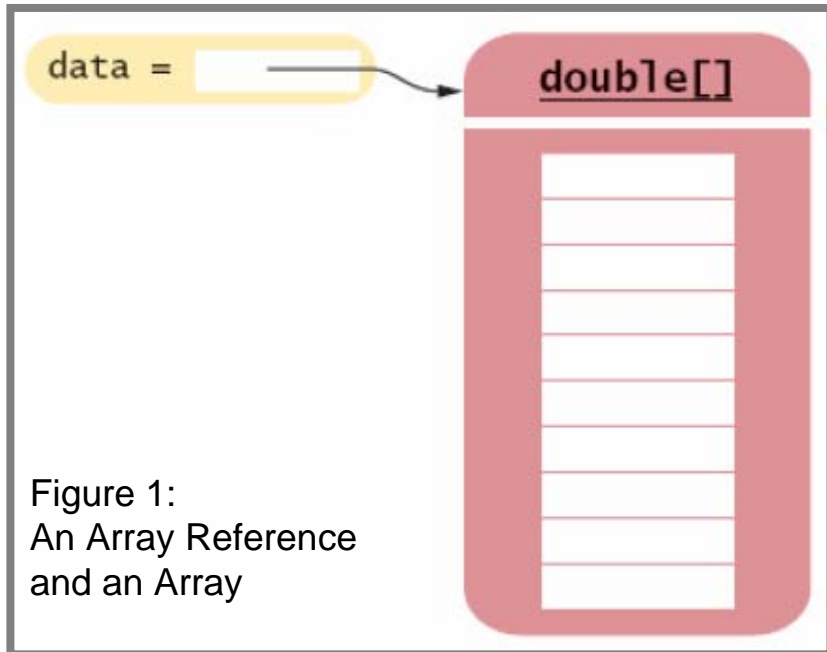
Arrays

- Each element in the array is specified by an integer **index** that is placed inside square brackets `[]`: `data[2]`
- You can store a value at that location with an assignment statement:

```
data[2] = 29.95;
```



The index values **start at 0**



Continued...

Arrays



The **last element** in the array has an index **one less** than the array length. If you try to access an element that does not exist, then an exception is thrown:

```
double[] data = new double[10];  
data[10] = 29.95; // ERROR
```

- To avoid **bound errors**, you will want to know how many elements are in an array. The **length** field returns the number of elements:
 - `data.length` is the length of the `data` array
 - you cannot assign a new value to the `length` instance variable, i.e., `length` is a `final public` instance variable
- The following code ensures that you only access the array when the index variable `i` is within the legal bounds:

```
if (0 <= i && i < data.length) data[i] = value;
```



Arrays length is fixed. If you start out with an array of 10 elements and later decide that you need to add additional elements, you need to make a new array and copy all values of the existing array into the new one

Syntax 8.1: Array Construction

```
new typeName[length]
```

Example:

```
new double[10]
```

Purpose:

To construct an array with a given number of elements

Syntax 8.2: Array Element Access

arrayReference[*index*]

Example:

`data[2]`

Purpose:

To access an element in an array

Bounds Errors



The most common **array error** is attempting to access a nonexistent position:

```
double[] data = new double[10];  
data[10] = 29.95; // ERROR
```

When the program runs, an **out-of-bounds** index generates an **exception** and terminates the program

- This is a great improvement over languages such as C and C++ (with those languages there is no error message)

Uninitialized Arrays



A common error is to allocate an array reference, but not an actual array:

```
double[] data; // array reference  
data[0] = 29.95; // ERROR
```

Array variables work exactly like **object variables** –they are only references to the actual array. To construct the actual array, you must use the **new** operator:

```
double[] data = new double[10];
```

Array Initialization

- You can **initialize an array** by allocating it and then filling each entry:

```
int[] primes = new int[5];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
primes[4] = 11;
```

- However, if you already know all the elements that you want to place in the array, there is an **easier way**: list all elements that you want to include in the array, enclosed in braces and separated by commas:

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

The Java compiler counts how many elements you want to place in the array, allocates an array of the correct size, and fills it with the elements that you specify

Self Check

1. What elements does the data array contain after the following statements?

```
double[] data = new double[10];  
for (int i = 0; i < data.length; i++) data[i] = i * i;
```

2. What do the following program segments print? Or, if there is an error, describe the error and specify whether it is detected at compile-time or at run-time

```
1. double[] a = new double[10];  
   System.out.println(a[0]);  
2. double[] b = new double[10];  
   System.out.println(b[10]);  
3. double[] c;  
   System.out.println(c[0]);
```

Answers

1. 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, but not 100

2.

1. 0

2. a run-time error: array index out of bounds

3. a compile-time error: c is not initialized

Array Lists



Arrays are a rather primitive construct. In this section we introduce the **ArrayList** class that lets you collect objects, just like an array does. Array lists offer two significant conveniences:

- Array lists can be **grow** and **shrink** as needed
- The `ArrayList` class supplies methods for many common tasks, such as **inserting** and **removing** elements



Use an **ArrayList<T>** whenever you want to collect **objects** of type **T**. Keep in mind that you cannot use primitive types as type parameters – there is no `ArrayList<int>` or `ArrayList<double>`

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();  
accounts.add(new BankAccount(1001));  
accounts.add(new BankAccount(1015));  
accounts.add(new BankAccount(1022));
```

- When you construct an `ArrayList` object, it has size **0**. You use the **add()** method to add an object to the end of the array list

Continued...

Array Lists

- The size of an array list increases after each call to **add()**. The **size()** method yields the current size of the array list
- To get objects out of the array list, use the **get()** method, not the **[]** operator. As with arrays, index values start at **0** and the last valid index is given by **i - 1**, where **i** is the size of the array list:

```
BankAccount anAccount = accounts.get(2);  
    // gets the third element of the array list
```

```
int i = accounts.size();  
anAccount = accounts.get(i); // Error  
    // legal index values are 0. . .i-1
```

- To set an array list element to a new value, use the **set()** method:

```
BankAccount anAccount = new BankAccount(1729);  
accounts.set(2, anAccount);
```

- The **set()** method can only overwrite existing values. It is different from the **add()** method, which adds a new object to the end of the array list

Continued...

Array Lists

- You can also **insert** an object in the middle of an array list. The call `accounts.add(i, a)` adds the object `a` at position `i` and moves all elements by one position, from the current element at position `i` to the last element in the array list (after each call to the add method, the size of the array list increases by 1)
- Conversely, the call `accounts.remove(i)` removes the element at position `i`, moves all elements after the removed element down by one position, and reduces the size of the array list by 1

 To use array lists, you have to import the generic class `java.util.ArrayList`

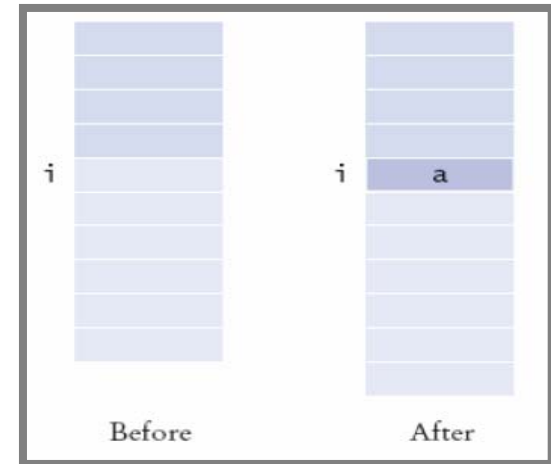


Figure 3: Adding an Element in the Middle of an Array List

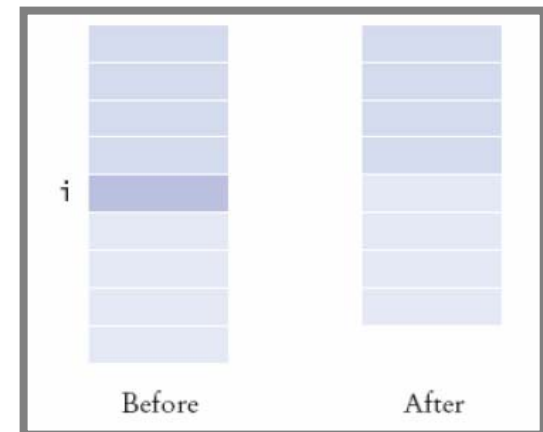


Figure 4: Removing an Element in the Middle of an Array List

File: ArrayListTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     This program tests the ArrayList class.
05: */
06: public class ArrayListTester
07: {
08:     public static void main(String[] args)
09:     {
10:         ArrayList<BankAccount> accounts
11:             = new ArrayList<BankAccount>();
12:         accounts.add(new BankAccount(1001));
13:         accounts.add(new BankAccount(1015));
14:         accounts.add(new BankAccount(1729));
15:         accounts.add(1, new BankAccount(1008));
16:         accounts.remove(0);
```

Continued...

File: ArrayListTester.java

```
17:
18:     System.out.println("size=" + accounts.size());
19:     BankAccount first = accounts.get(0);
20:     System.out.println("first account number="
21:         + first.getAccountNumber());
22:     BankAccount last = accounts.get(accounts.size() - 1);
23:     System.out.println("last account number="
24:         + last.getAccountNumber());
25:     }
26: }
```

File: BankAccount.java

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance
09:         @param anAccountNumber the account number for this account
10:     */
11:     public BankAccount(int anAccountNumber)
12:     {
13:         accountNumber = anAccountNumber;
14:         balance = 0;
15:     }
16:
```

Continued...

File: BankAccount.java

```
17:     /**
18:         Constructs a bank account with a given balance
19:         @param anAccountNumber the account number for this account
20:         @param initialBalance the initial balance
21:     */
22:     public BankAccount(int anAccountNumber, double initialBalance)
23:     {
24:         accountNumber = anAccountNumber;
25:         balance = initialBalance;
26:     }
27:
28:     /**
29:         Gets the account number of this bank account.
30:         @return the account number
31:     */
32:     public int getAccountNumber()
33:     {
34:         return accountNumber;
35:     }
```

Continued...

File: BankAccount.java

```
36:
37:     /**
38:         Deposits money into the bank account.
39:         @param amount the amount to deposit
40:     */
41:     public void deposit(double amount)
42:     {
43:         double newBalance = balance + amount;
44:         balance = newBalance;
45:     }
46:
47:     /**
48:         Withdraws money from the bank account.
49:         @param amount the amount to withdraw
50:     */
51:     public void withdraw(double amount)
52:     {
53:         double newBalance = balance - amount;
54:         balance = newBalance;
```

Continued...

File: BankAccount.java

```
55:     }
56:
57:     /**
58:         Gets the current balance of the bank account.
59:         @return the current balance
60:     */
61:     public double getBalance()
62:     {
63:         return balance;
64:     }
65:
66:     private int accountNumber;
67:     private double balance;
68: }
```

Output

```
size=3
first account number=1008
last account number=1729
```

Length and Size



The Java **syntax** for determining the number of elements in an array, an array list, and a string is not at all consistent:

- array **a** → **a.length** (final instance field)
- array list **a** → **a.size()** (method)
- string **a** → **a.length()** (method)

Self Check

3. How do you construct an array of 10 strings? An array list of strings?
4. What is the content of `names` after the following statements?

```
ArrayList<String> names = new ArrayList<String>();  
names.add("A");  
names.add(0, "B");  
names.add("C");  
names.remove(1);
```

Answers

3.

```
new String[10];  
new ArrayList<String>( );
```

4. names contains the strings "B" and "C" at positions 0 and 1

Wrappers

- Because numbers are not objects in Java, you cannot directly insert them into array lists (for example, you cannot form an `ArrayList<double>`)



To store sequences of numbers in an array list, you must turn them into objects by using **wrapper classes**. There are wrapper classes for all eight primitive types: `Byte`, `Boolean`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` (note that the wrapper class names start with uppercase letters)

- Each wrapper class object contains a value of the corresponding primitive type. Wrapper objects can be used anywhere that objects are required instead of primitive type values

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

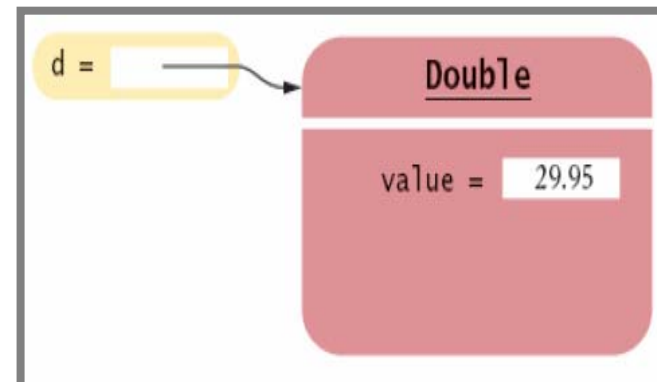


Figure 5: An Object of a Wrapper Class

Auto-boxing



Starting with Java 5.0, conversion between primitive types and the corresponding wrapper classes is automatic. This process is called **auto-boxing** (or **auto-wrapping**). Conversely, wrapper objects are automatically “unboxed” to primitive types

- For example, if you assign a number to a `Double` object, the number is automatically “put into a box”, namely a wrapper object

```
Double d = 29.95; // auto-boxing; same as Double d = new Double(29.95);  
double x = d; // auto-unboxing; same as double x = d.doubleValue();
```

- If you use Java version 5.0 or higher, array lists of numbers are straightforward. Simply remember to use the wrapper type when you declare the array list, and then rely on auto-boxing:

```
ArrayList<Double> data = new ArrayList<Double>();  
data.add(29.95);  
double x = data.get(0);
```



Storing wrapped numbers is quite inefficient. The use of wrappers is acceptable for short array lists, but you should use arrays for long sequences of numbers or characters

Self Check

5. What is the difference between the types `double` and `Double`?
6. Suppose `data` is an `ArrayList<Double>` of size > 0 . How do you increment the element with index 0?

Answers

5. `double` is one of the eight primitive types.
`Double` is a class type.
6. `data.set(0, data.get(0) + 1);`

The Enhanced `for` Loop



Often, you need to iterate through a sequence of elements –such as the elements of an array or array list. The **enhanced `for` loop** makes this process particularly easy to program:

```
double[] data = . . . ;
double sum = 0;
for (double e : data) // You should read this loop as "for each e in data"
{
    sum = sum + e;
}
```

- You don't have to use the **"for each"** construct to loop through all elements in an array:

```
double[] data = . . . ;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
    double e = data[i];
    sum = sum + e;
}
```

- In the "for each" loop, the element variable **e** is assigned values `data[0]`, `data[1]`, and so on. In the ordinary for loop, the index variable **i** is assigned values 0, 1, and so on

Continued...

The Enhanced `for` Loop

- You can also use the **enhanced `for` loop** to visit all elements of an array list

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
    sum = sum + a.getBalance();
}
```

- This loop is equivalent to the following ordinary **`for`** loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    sum = sum + a.getBalance();
}
```

- The **“`for each`”** loop traverses the elements of a collection from the beginning to the end. Sometimes you don't want to start at the beginning, or you may need to traverse the collection backwards. In those situations, use an ordinary `for` loop

Syntax 8.3: The "for each" Loop

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

Example:

```
for (double e : data)  
    sum = sum + e;
```

Purpose:

To execute a loop for each element in the collection. In each iteration, the variable is assigned the next element of the collection. Then the statement is executed.

Self Check

7. Write a "for each" loop that prints all elements in the array data
8. Why is the "for each" loop not an appropriate shortcut for the following ordinary `for` loop?

```
for (int i = 0; i < data.length; i++) data[i] = i * i;
```

Answers

7.

```
for (double x : data) System.out.println(x);
```
8. The loop writes a value into `data[i]`. The "for each" loop does not have the index variable `i`.

Simple Array Algorithms: Counting Matches

- Go through the entire collection and **increment a counter** each time you find a match:

```
public class Bank
{
    public int count(double atLeast)
    {
        int matches = 0;
        for (BankAccount a : accounts)
        {
            if (a.getBalance() >= atLeast) matches++;
            // Found a match
        }
        return matches;
    }
    . . .
    private ArrayList<BankAccount> accounts;
}
```

Simple Array Algorithms: Finding a Value

- Check all elements until you have found a match (note that the loop might fail to find an answer). This search process is called a **linear search** through the array list:

```
public class Bank
{
    public BankAccount find(int accountNumber)
    {
        for (BankAccount a : accounts)
        {
            if (a.getAccountNumber() == accountNumber) // Found a match
                return a;
        }
        return null; // No match in the entire array list
    }
    . . .
}
```

Simple Array Algorithms: Finding the Maximum or Minimum

- Initialize a **candidate** with the starting element
- **Compare** the candidate with each of the remaining elements
- **Update** it if you find a larger (if max) or smaller (if min) value

```
BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    if (a.getBalance() > largestYet.getBalance())
        largestYet = a;
}
return largestYet;
```

- This approach works only if there is at least one element in the array list. Otherwise, we can return **null**

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
. . .
```

File Bank.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     This bank contains a collection of bank accounts.
05: */
06: public class Bank
07: {
08:     /**
09:         Constructs a bank with no bank accounts.
10:     */
11:     public Bank()
12:     {
13:         accounts = new ArrayList<BankAccount>();
14:     }
15:
16:     /**
17:         Adds an account to this bank.
18:         @param a the account to add
19:     */
```

Continued...

File Bank.java

```
20:     public void addAccount(BankAccount a)
21:     {
22:         accounts.add(a);
23:     }
24:
25:     /**
26:      * Gets the sum of the balances of all accounts in this bank.
27:      * @return the sum of the balances
28:      */
29:     public double getTotalBalance()
30:     {
31:         double total = 0;
32:         for (BankAccount a : accounts)
33:         {
34:             total = total + a.getBalance();
35:         }
36:         return total;
37:     }
38:
```

Continued...

File Bank.java

```
39:     /**
40:         Counts the number of bank accounts whose balance is at
41:         least a given value.
42:         @param atLeast the balance required to count an account
43:         @return the number of accounts having least the given
// balance
44:     */
45:     public int count(double atLeast)
46:     {
47:         int matches = 0;
48:         for (BankAccount a : accounts)
49:         {
50:             if (a.getBalance() >= atLeast) matches++; // Found
// a match
51:         }
52:         return matches;
53:     }
54:
```

Continued...

File Bank.java

```
55:     /**
56:         Finds a bank account with a given number.
57:         @param accountNumber the number to find
58:         @return the account with the given number, or null
59:         if there is no such account
60:     */
61:     public BankAccount find(int accountNumber)
62:     {
63:         for (BankAccount a : accounts)
64:         {
65:             if (a.getAccountNumber() == accountNumber)
66:                 return a;
67:         }
68:         return null; // No match in the entire array list
69:     }
70:
```

Continued...

File Bank.java

```
71:     /**
72:         Gets the bank account with the largest balance.
73:         @return the account with the largest balance, or
74:         null if the bank has no accounts
75:     */
76:     public BankAccount getMaximum()
77:     {
78:         if (accounts.size() == 0) return null;
79:         BankAccount largestYet = accounts.get(0);
80:         for (int i = 1; i < accounts.size(); i++)
81:         {
82:             BankAccount a = accounts.get(i);
83:             if (a.getBalance() > largestYet.getBalance())
84:                 largestYet = a;
85:         }
86:         return largestYet;
87:     }
88:
89:     private ArrayList<BankAccount> accounts;
90: }
```

File BankTester.java

```
01: /**
02:     This program tests the Bank class.
03: */
04: public class BankTester
05: {
06:     public static void main(String[] args)
07:     {
08:         Bank firstBankOfJava = new Bank();
09:         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
10:         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
11:         firstBankOfJava.addAccount(new BankAccount(1729, 15000));
12:
13:         double threshold = 15000;
14:         int c = firstBankOfJava.count(threshold);
15:         System.out.println(c + " accounts with balance >= "
+ threshold);
```

Continued...

File BankTester.java

```
16:
17:     int accountNumber = 1015;
18:     BankAccount a = firstBankOfJava.find(accountNumber);
19:     if (a == null)
20:         System.out.println("No account with number "
+ accountNumber);
21:     else
22:         System.out.println("Account with number "
+ accountNumber
23:             + " has balance " + a.getBalance());
24:
25:     BankAccount max = firstBankOfJava.getMaximum();
26:     System.out.println("Account with number "
27:         + max.getAccountNumber()
28:         + " has the largest balance.");
29: }
30: }
```

Continued...

File BankTester.java

Output

```
2 accounts with balance >= 15000.0  
Account with number 1015 has balance 10000.0  
Account with number 1001 has the largest balance.
```

Self Check

9. What does the `find` method do if there are two bank accounts with a matching account number?
10. Would it be possible to use a "for each" loop in the `getMaximum` method?

Answers

9. It returns the first match that it finds
10. Yes, but the first comparison would always fail

Two-Dimensional Arrays

- Arrays and array lists can store linear sequences. Occasionally, you want to store collections that have a two-dimensional layout. Such an arrangement, consisting of rows and columns of values, is called a **two-dimensional array** or matrix
- When constructing a two-dimensional array, you specify how many rows and columns you need:

```
final int ROWS = 3;  
final int COLUMNS = 3;  
String[][] board = new String[ROWS][COLUMNS];
```

- You access elements with an index pair `a[i][j]`

```
board[i][j] = "x";
```

Continued...

Two-Dimensional Arrays

- When filling or searching a two-dimensional array, it is common to use two **nested loops**:

```
for (int i = 0; i < ROWS; i++)  
    for (int j = 0; j < COLUMNS; j++)  
        board[i][j] = " ";
```

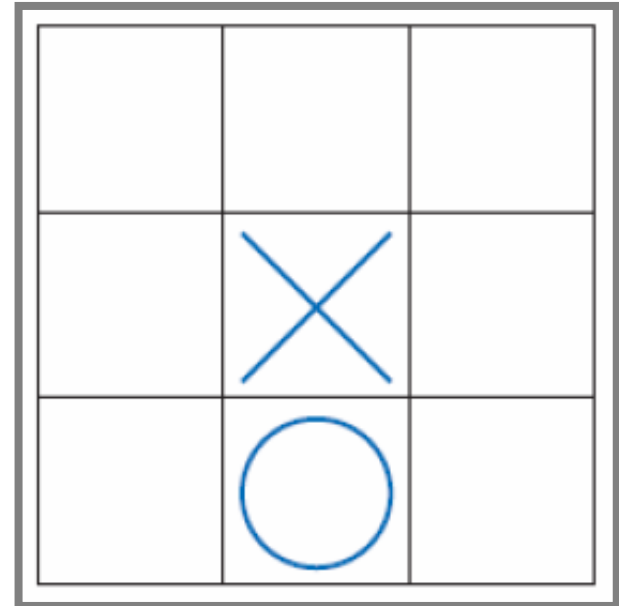


Figure 6:
A Tic-Tac-Toe Board

File TicTacToe.java

```
01: /**
02:     A 3 x 3 tic-tac-toe board.
03: */
04: public class TicTacToe
05: {
06:     /**
07:         Constructs an empty board.
08:     */
09:     public TicTacToe()
10:     {
11:         board = new String[ROWS][COLUMNS];
12:         // Fill with spaces
13:         for (int i = 0; i < ROWS; i++)
14:             for (int j = 0; j < COLUMNS; j++)
15:                 board[i][j] = " ";
16:     }
17:
```

Continued...

File TicTacToe.java

```
18:    /**
19:        Sets a field in the board. The field must be unoccupied.
20:        @param i the row index
21:        @param j the column index
22:        @param player the player ("x" or "o")
23:    */
24:    public void set(int i, int j, String player)
25:    {
26:        if (board[i][j].equals(" "))
27:            board[i][j] = player;
28:    }
29:
30:    /**
31:        Creates a string representation of the board, such as
32:        |x  o|
33:        |  x|
34:        |  o|
35:        @return the string representation
36:    */
```

Continued...

File TicTacToe.java

```
37:     public String toString()
38:     {
39:         String r = "";
40:         for (int i = 0; i < ROWS; i++)
41:         {
42:             r = r + "|";
43:             for (int j = 0; j < COLUMNS; j++)
44:                 r = r + board[i][j];
45:             r = r + "|\n";
46:         }
47:         return r;
48:     }
49:
50:     private String[][] board;
51:     private static final int ROWS = 3;
52:     private static final int COLUMNS = 3;
53: }
```

File TicTacToeTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program tests the TicTacToe class by prompting the
05:     user to set positions on the board and printing out the
06:     result.
07: */
08: public class TicTacToeTester
09: {
10:     public static void main(String[] args)
11:     {
12:         Scanner in = new Scanner(System.in);
13:         String player = "x";
14:         TicTacToe game = new TicTacToe();
15:         boolean done = false;
16:         while (!done)
17:         {
```

Continued...

File TicTacToeTester.java

```
18:         System.out.print(game.toString());
19:         System.out.print(
20:             "Row for " + player + " (-1 to exit): ");
21:         int row = in.nextInt();
22:         if (row < 0) done = true;
23:         else
24:         {
25:             System.out.print("Column for " + player + ": ");
26:             int column = in.nextInt();
27:             game.set(row, column, player);
28:             if (player.equals("x"))
29:                 player = "o";
30:             else
31:                 player = "x";
32:         }
33:     }
34: }
35: }
```

Continued...

Output

```
| |  
| |  
| |  
Row for x (-1 to exit): 1  
Column for x: 2  
  
| |  
| x|  
|  
Row for o (-1 to exit): 0  
Column for o: 0  
  
|o |  
| x|  
| |  
Row for x (-1 to exit): -1
```

Two-Dimensional Arrays with Variable Row Lengths

- When you declare a two-dimensional array with the command `int[][] a = new int[5][5];` then you get a 5-by-5 matrix that can store 25 elements. In this matrix, all rows have the same length
- In Java, it is possible to declare arrays in which the **row length varies**. To allocate such an array, first you must allocate space to hold five rows, indicating that you will manually set each row by leaving the second array index empty. Then, allocate each row separately:

```
int[][] b = new int[5][];  
for (int i = 0; i < b.length; i++)  
    b[i] = new int[i + 1];
```

Multidimensional Arrays

- You can declare arrays with **more than two** dimensions:

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

- Each array element is specified by three index values:

```
rubiksCube[i][j][k];
```

- However, these arrays are quite rare, particularly in object-oriented programs

Self Check

11. How do you declare and initialize a 4-by-4 array of integers?
12. How do you count the number of spaces in the tic-tac-toe board?

Answers

11.

```
int[][] array = new int[4][4];
```

12.

```
int count = 0;
for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLUMNS; j++)
        if (board[i][j] == ' ') count++;
```

Copying Arrays:

Copying Array References

⚠ Array variables work just like **object variables** –they hold a reference to the actual array. If you copy the reference, you get another reference to the same array:

```
double[] data = new double[10];  
// fill array . . .  
double[] prices = data;
```

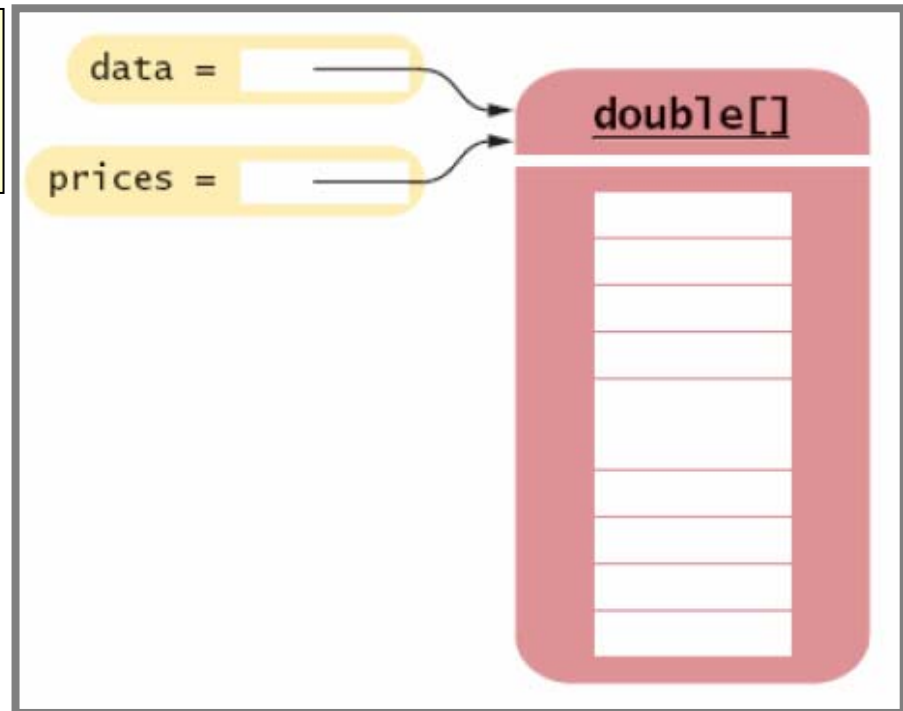


Figure 7:
Two References to the Same Array

Copying Arrays: Cloning Arrays

- If you want to make a true copy of an array, call the `clone()` method (note that you need to **cast** the return value of the `clone()` method from the type `Object` to type `double[]`):

```
double[] prices = (double[]) data.clone();
```

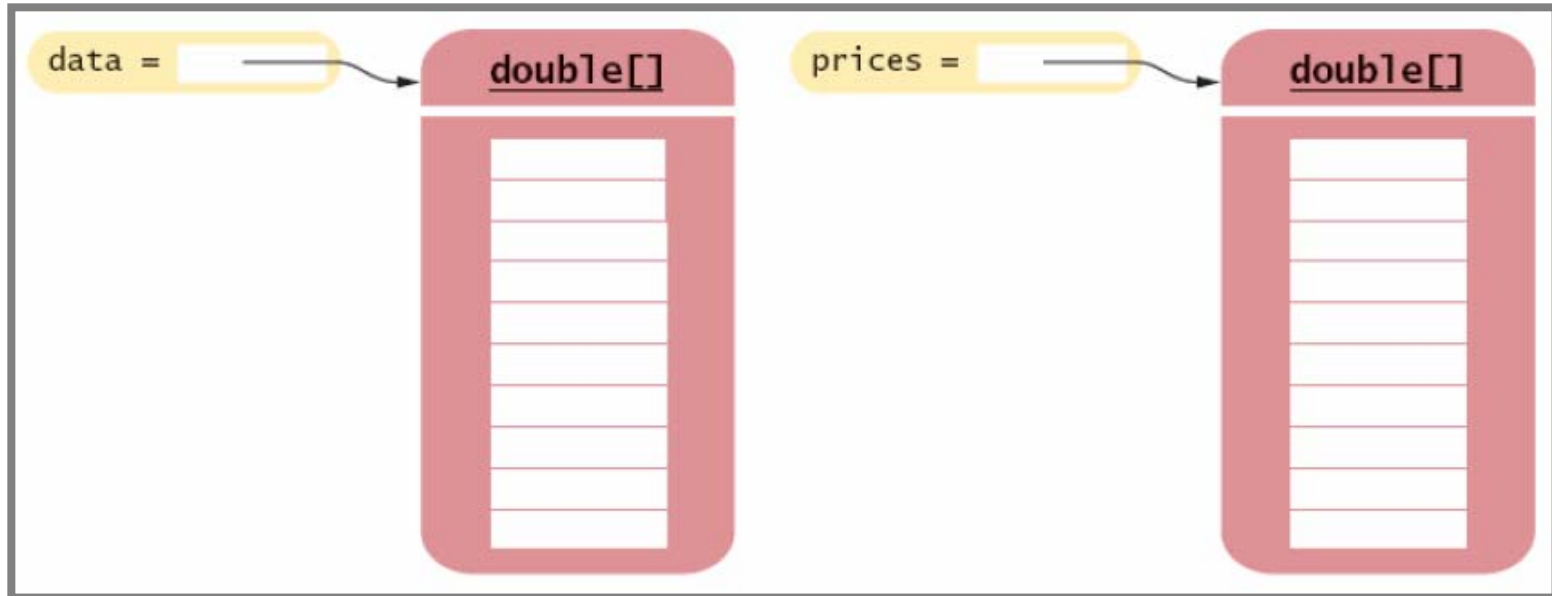


Figure 8: Cloning an Array

Copying Arrays: Copying Array Elements

- Occasionally, you need to copy elements from one array into another array. For that purpose, you can use the static **`System.arraycopy()`** method:

```
System.arraycopy(from, fromStart, to, toStart, count);
```

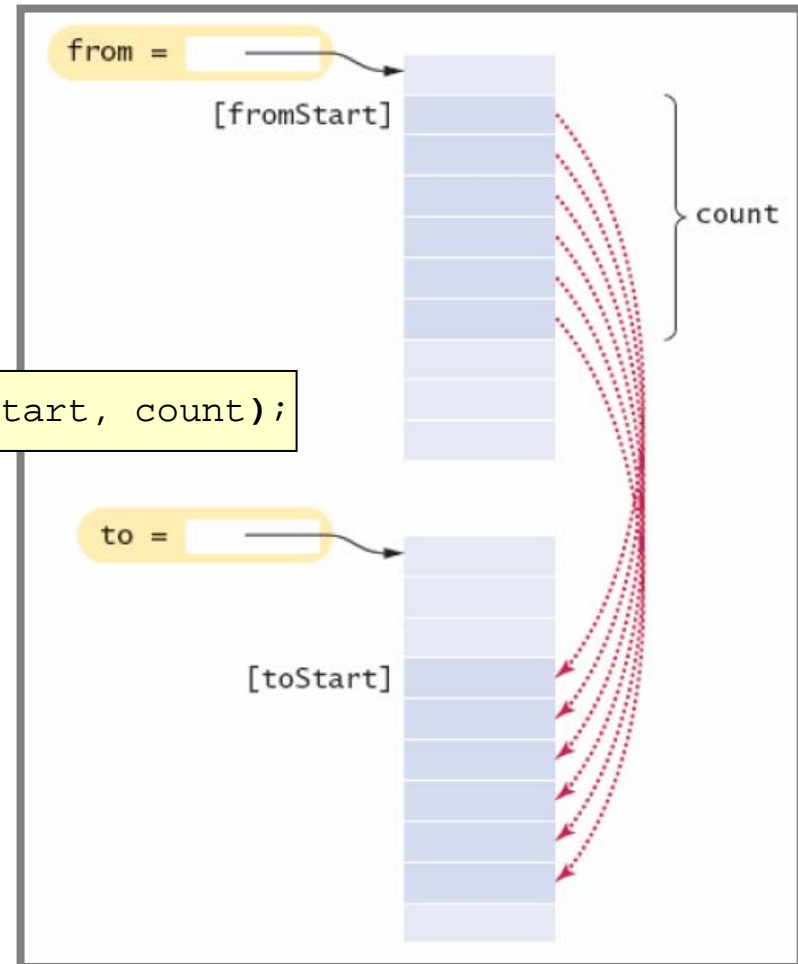


Figure 9:
The `System.arraycopy` Method

Adding an Element to an Array

- One use of the `System.arraycopy()` method is to add or remove elements in the middle of an array
- To add a new element at position `i` into `data`, first move all elements from `i` onward one position up, then insert the new value (note that the last element in the array is lost):

```
System.arraycopy(data, i, data, i + 1, data.length - i - 1);  
data[i] = x;
```

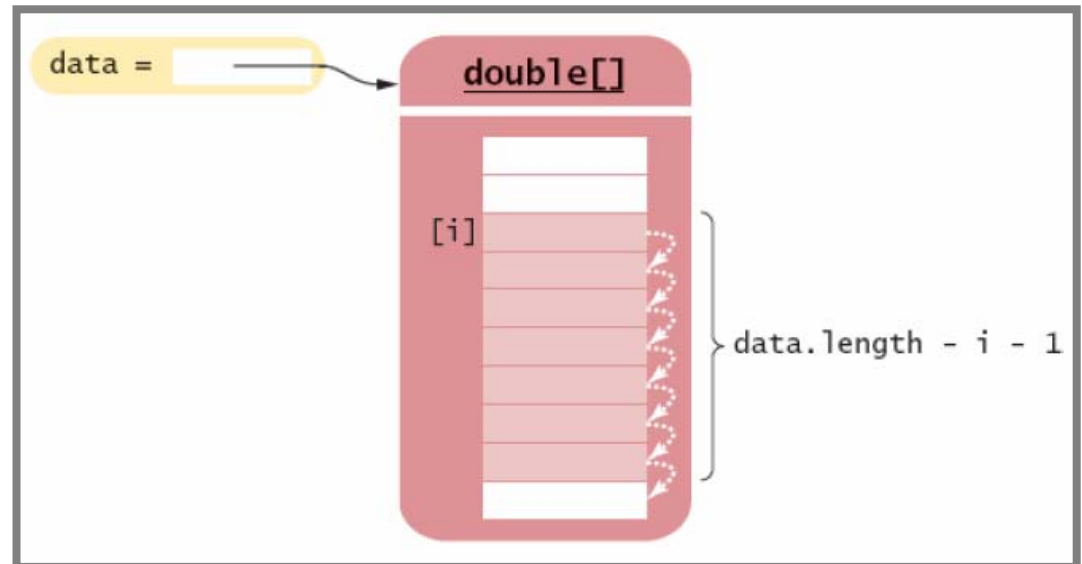


Figure 10:
Inserting a New Element Into an Array

Removing an Element from an Array

- To **remove** the element at position i , copy the elements above the position downward:

```
System.arraycopy(data, i + 1, data, i, data.length - i - 1);
```

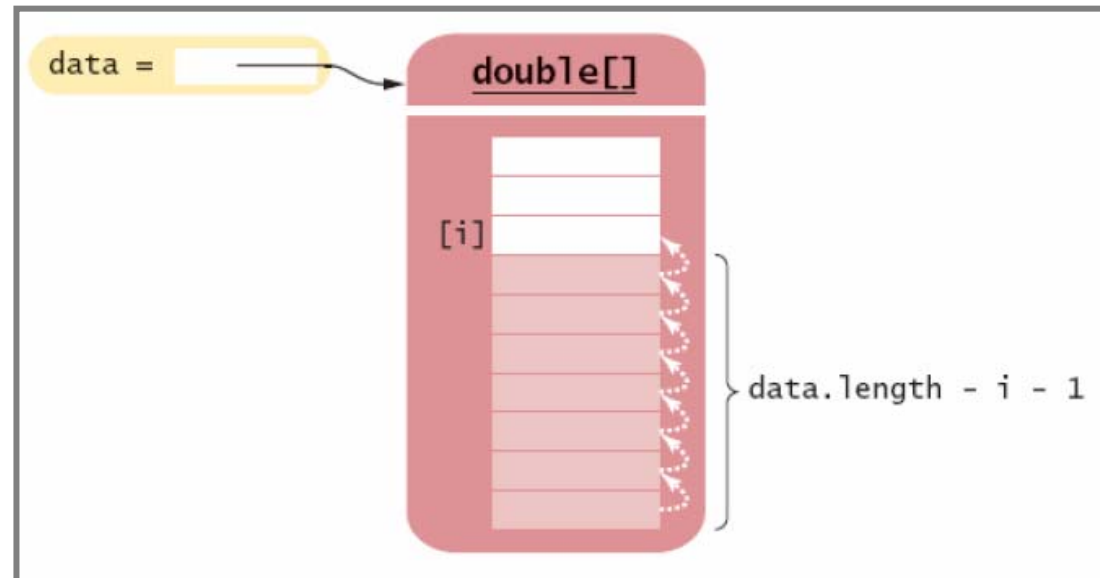


Figure 11
Removing an Element from an Array

Growing an Array

- If the array is full and you need more space, you can **grow** the array:

1. Create a new, larger array.

```
double[] newData = new double[2 * data.length];
```

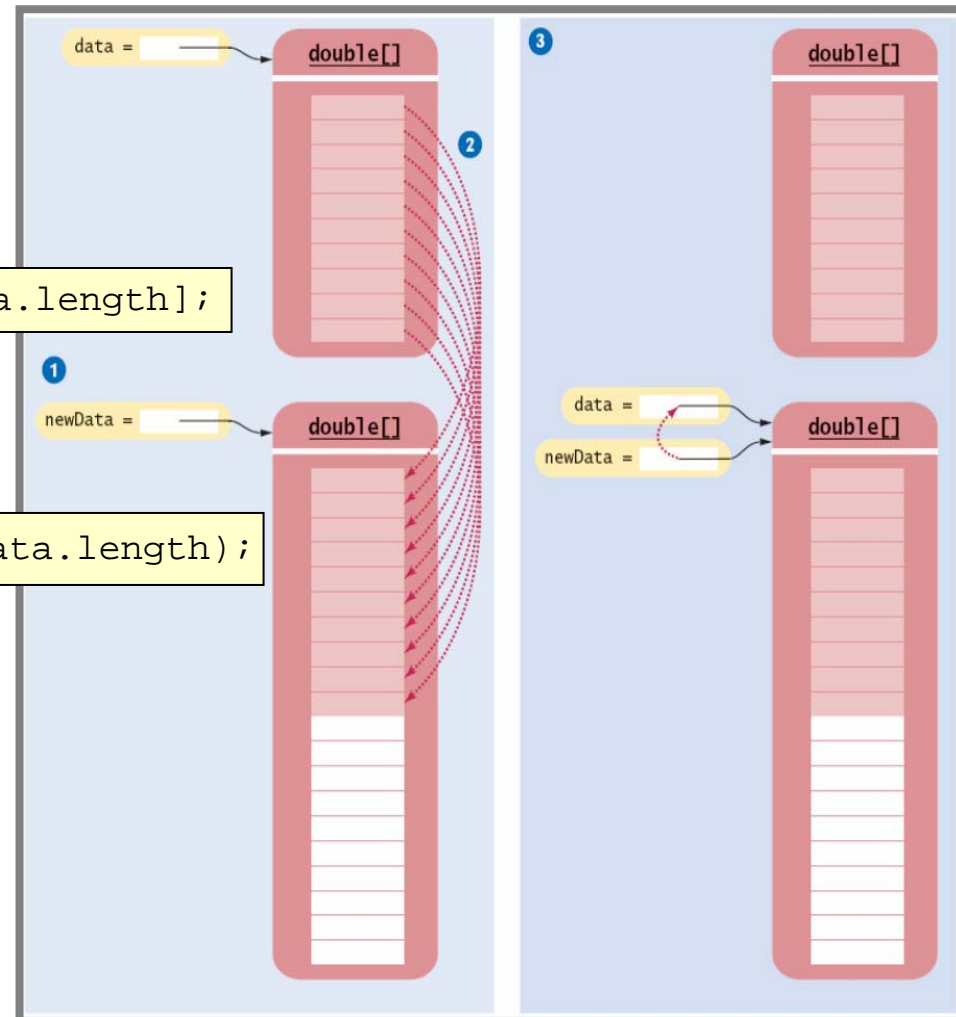
2. Copy all elements into the new array

```
System.arraycopy(data, 0, newData, 0, data.length);
```

3. Store the reference to the new array in the array variable

```
data = newData;
```

Figure 12: Growing an Array



Self Check

13. How do you add or remove elements in the middle of an array list?
14. Why do we double the length of the array when it has run out of space rather than increasing it by one element?

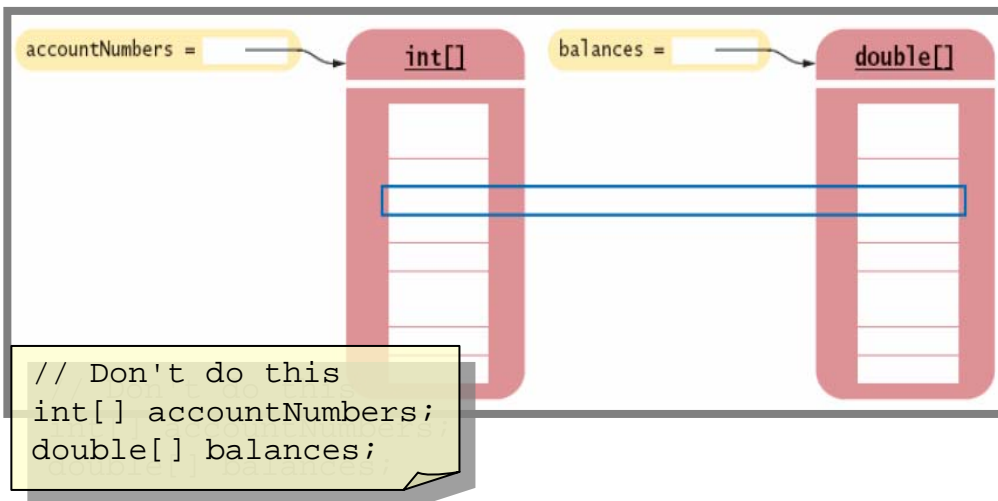
Answers

13. Use the `insert` and `remove` methods.
14. Allocating a new array and copying the elements is time-consuming. You wouldn't want to go through the process every time you add an element.

Make Parallel Arrays into Arrays of Objects

- Programmers who are familiar with arrays, but unfamiliar with object-oriented programming, sometimes distribute information across separate arrays. Arrays such as these are called **parallel arrays** (note that the *i*th slice contains data that need to be processed together):
- If you find yourself using two arrays that have the same length, ask yourself whether you couldn't replace them with a single array of a class type (look at a slice to find the concept that it represents, then make the concept into a class)

Figure 13: Avoid Parallel Arrays



```
BankAccount[] = accounts;
```

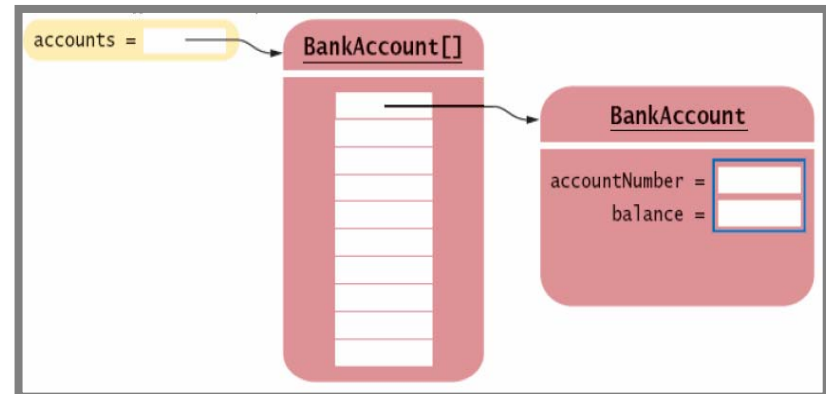


Figure 14: Reorganizing Parallel Arrays into Arrays of Objects

Partially Filled Arrays

- Suppose you write a program that reads a sequence of numbers into an array. You need to set the size of the array before you know how many elements you need. Once the array size is set, it cannot be changed. To solve this problem:
 - Make an array that is guaranteed to be larger than the largest possible number of entries, and partially fill it; keep a companion variable that tells you how many elements in the array are actually used:

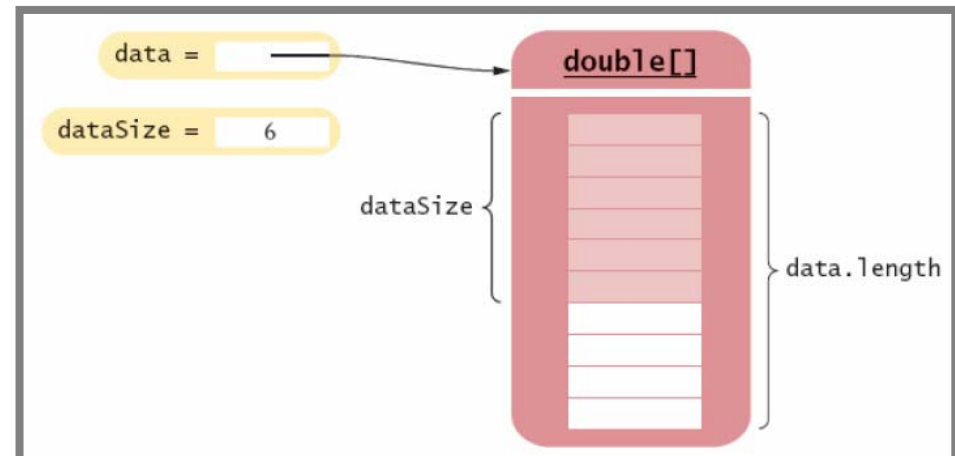
```
final int DATA_LENGTH = 100;  
double[] data = new double[DATA_LENGTH];  
int dataSize = 0;
```

```
data[dataSize] = x;  
dataSize++;
```

- When you run out of space, make a new array and copy the elements into it

Arrays lists use this technique behind the scenes

Figure 15:
A Partially Filled Array



Methods with a Variable Number of Parameters

- Starting with Java version 5.0, it is possible to declare methods that receive a **variable number of parameters**:

```
data.add(1, 3, 7);  
data.add(4);  
data.add(); //OK but useless
```

- The former `add()` method must be declared as:

```
public void add(double ... xs);
```

(the `...` symbol indicates that the method can receive any number of `double` values)

- The method implementation traverses the parameter array and processes the values:

```
for (x : xs)  
{  
    sum = sum + x;  
}
```

Chapter Summary

- An **array** is a sequence of values of the same type
- You access array elements with an integer index, using the notation `a[i]`
- Index values of an array range from `0` to `length - 1`. Accessing a nonexistent element results in a **bounds error**
- Use the `length` field to find the number of elements in an array
- The **ArrayList** class manages a sequence of objects
- The `ArrayList` class is a generic class: **ArrayList<T>** collects objects of type `T`
- To treat primitive type values as objects, you must use **wrapper classes**
- The **enhanced for loop** traverses all elements of a collection
- To count values in an array list, check all elements and count the matches until you reach the end of the array list
- To find a value in an array list, check all elements until you have found a match

Continued...

Chapter Summary

- To compute the maximum or minimum value of an array list, initialize a candidate with the starting element. Then compare the candidate with the remaining elements and update it if you find a larger or smaller value
- **Two-dimensional arrays** form a tabular, two-dimensional arrangement. You access elements with an index pair `a[i][j]`
- An array variable stores a reference to the array. Copying the variable yields a second reference to the same array
- Use the `clone` method to copy the elements of an array
- Use the `System.arraycopy` method to copy elements from one array to another
- Avoid **parallel arrays** by changing them into arrays of objects