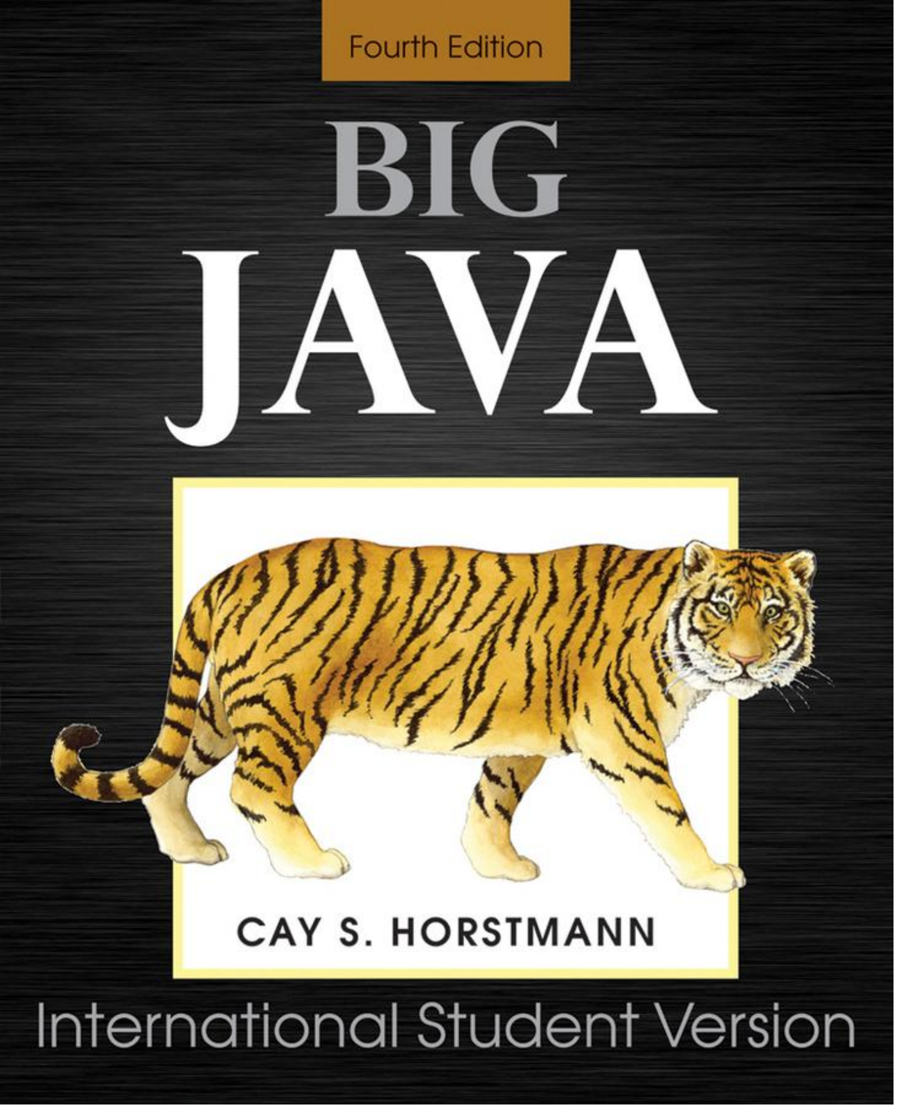


Chapter 6 – Arrays and Array Lists

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

T To understand the concept of regression testing

Arrays

- Array: Sequence of values of the same type
- Construct array:

  `new double[10]`

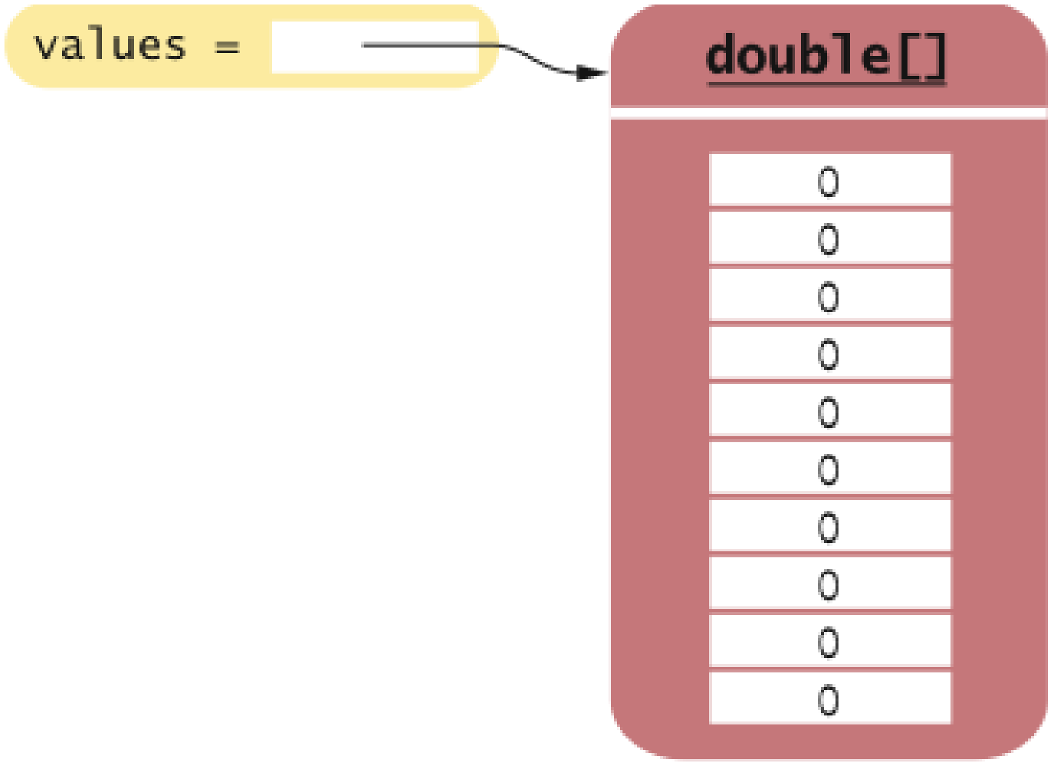
- Store in variable of type `double[]`:

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

- When array is created, all values are initialized depending on array type:
  - *Numbers:* `0`
  - *Boolean :* `false`
  - *Object References:* `null`

Arrays

Figure 1
An Array Reference
and an Array



Arrays

Use `[]` to access an element:

```
values[2] = 29.95;
```

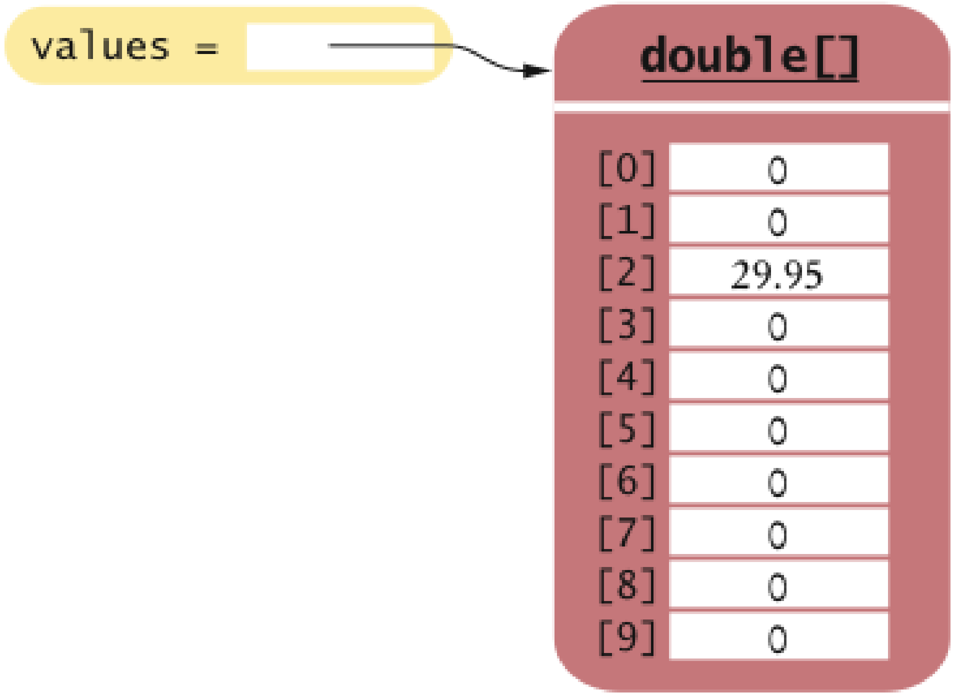


Figure 2
Modifying an Array Element

Arrays

- Using the value stored:

```
System.out.println("The value of this data item is "
   + values[2]);
```

- Get array length as `values.length` (Not a method!)
- Index values range from `0` to `length - 1`
- Accessing a nonexistent element results in a **bounds error**:

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

- Limitation: Arrays have fixed length

Declaring Arrays

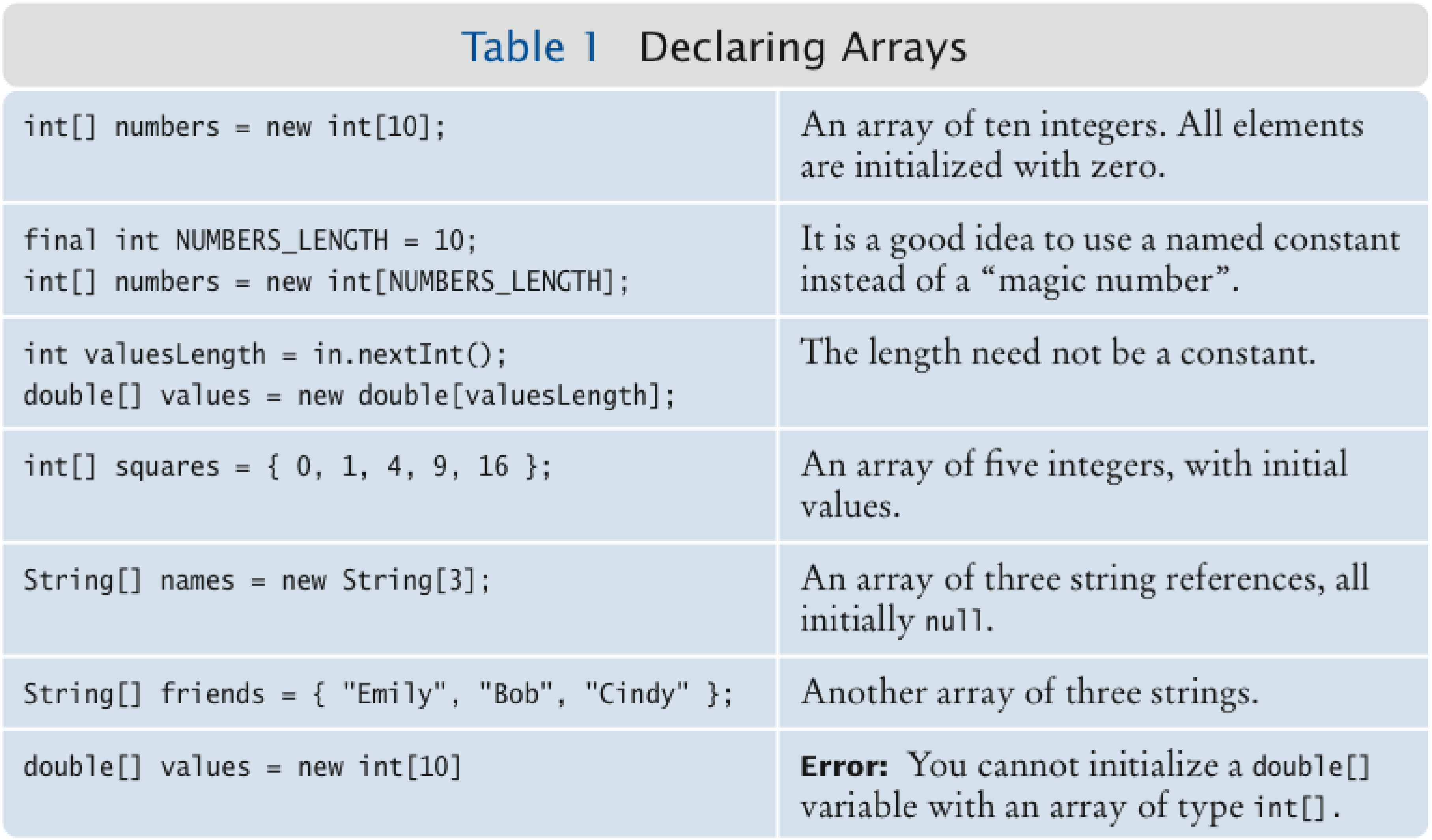


Table 1 Declaring Arrays

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.

`final int NUMBERS_LENGTH = 10;`
`int[] numbers = new int[NUMBERS_LENGTH];`
	It is a good idea to use a named constant instead of a “magic number”.

`int valuesLength = in.nextInt();`
`double[] values = new double[valuesLength];`
	The length need not be a constant.

`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] names = new String[3];`	An array of three string references, all initially `null`.
`String[] friends = { "Emily", "Bob", "Cindy" };`	Another array of three strings.
`double[] values = new int[10]`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

Self Check 6.1

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

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

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

Self Check 6.2

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.

```
a) double[] a = new double[10];
   System.out.println(a[0]);

b) double[] b = new double[10];
   System.out.println(b[10]);

c) double[] c;
   System.out.println(c[0]);
```

Answer:

a) 0

b) a run-time error: array index out of bounds

c) a compile-time error: c is not initialized

Make Parallel Arrays into Arrays of Objects

```
// Don't do this
int[] accountNumbers;
double[] balances;
```

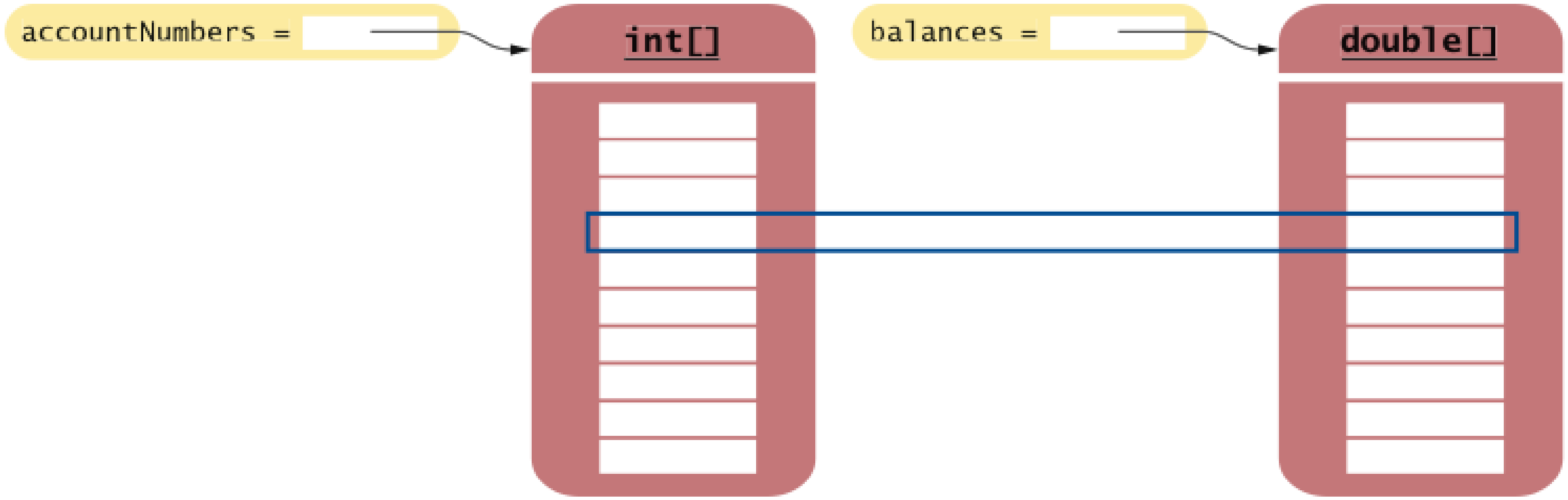


Figure 3 Avoid Parallel Arrays

Make Parallel Arrays into Arrays of Objects

Avoid parallel arrays by changing them into arrays of objects:

```
BankAccount[] accounts;
```

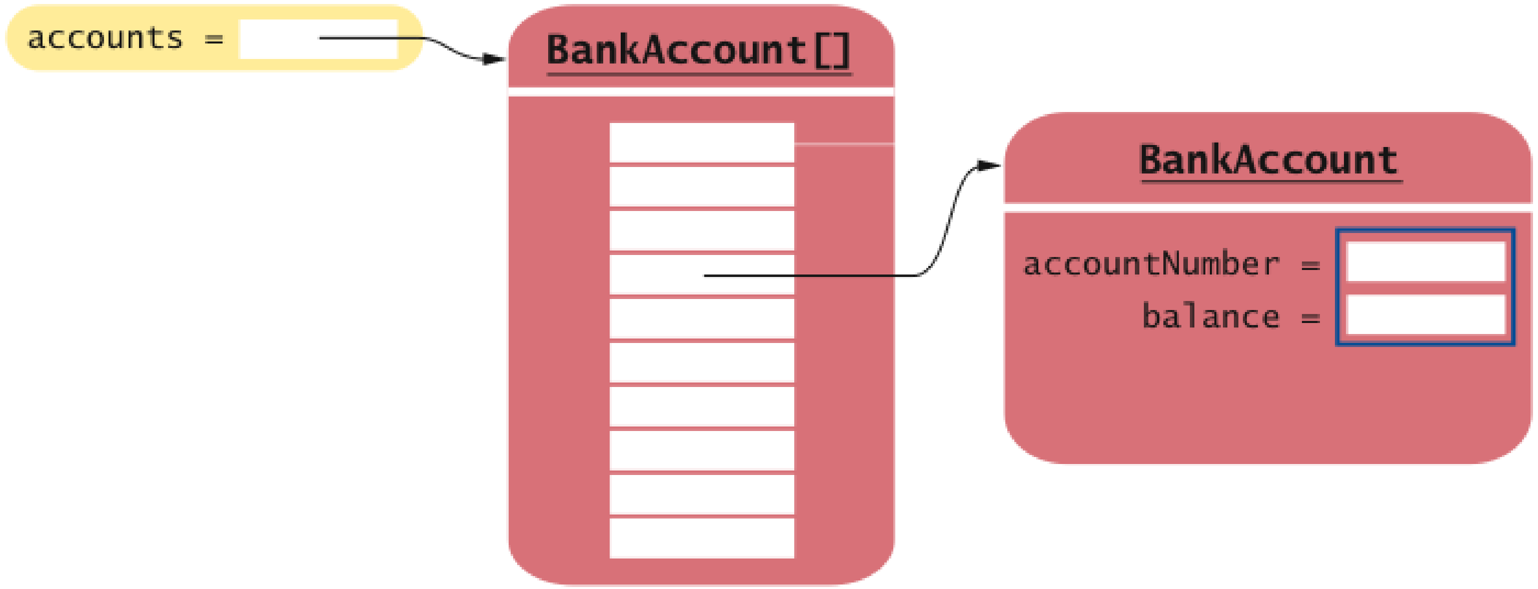
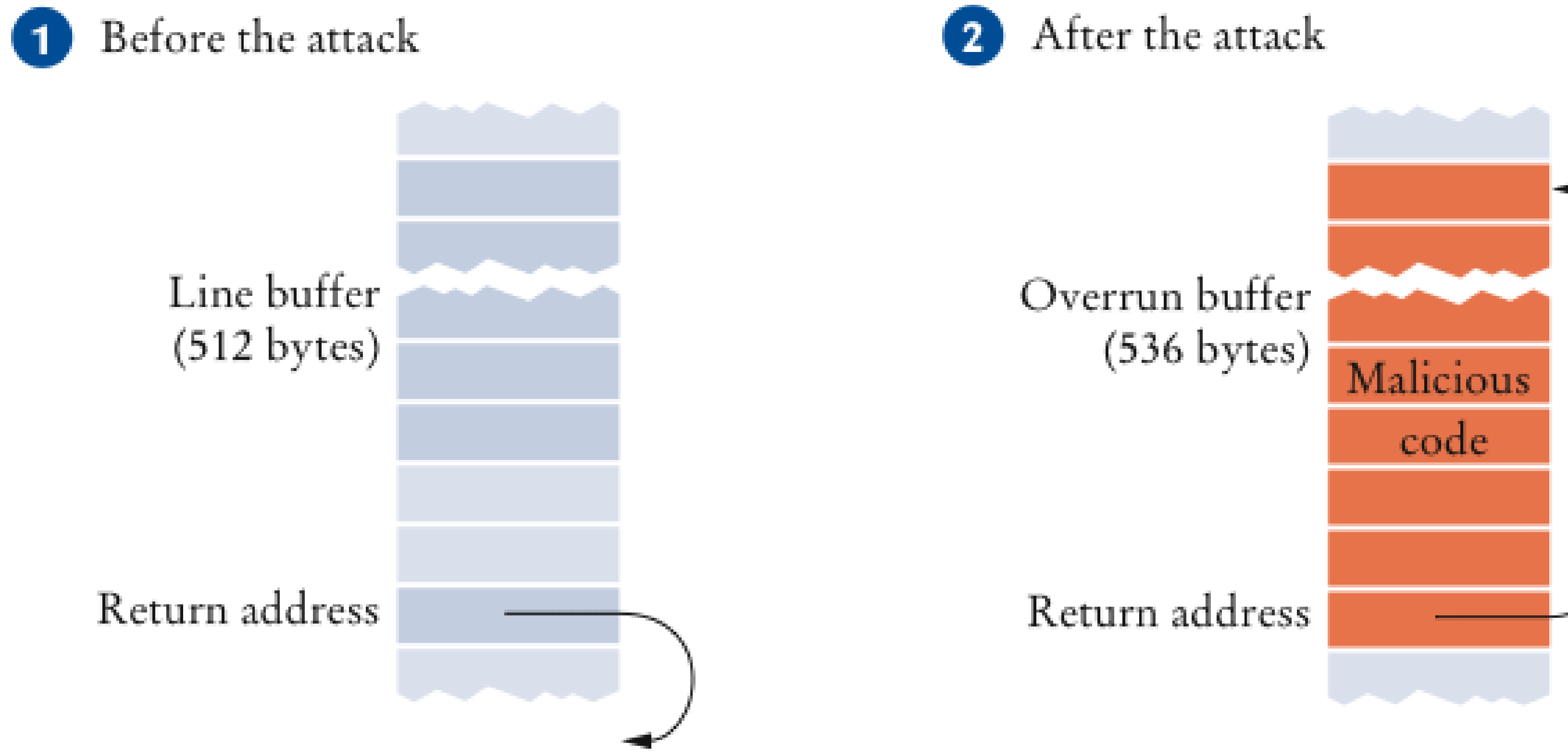


Figure 4 Reorganizing Parallel Arrays into an Array of Objects

An Early Internet Worm



A "Buffer Overrun" Attack

Array Lists

- `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- `ArrayList` is a **generic class**:

  `ArrayList<T>`

  collects objects of **type parameter** `T`:

  ```
  ArrayList<String> names = new ArrayList<String>();
  names.add("Emily");
  names.add("Bob");
  names.add("Cindy");
  ```
- `size` method yields number of elements

Adding Elements

To add an object to the end of the array list, use the `add` method:

```
names.add("Emily");
names.add("Bob"); 1
names.add("Cindy"); 2
```

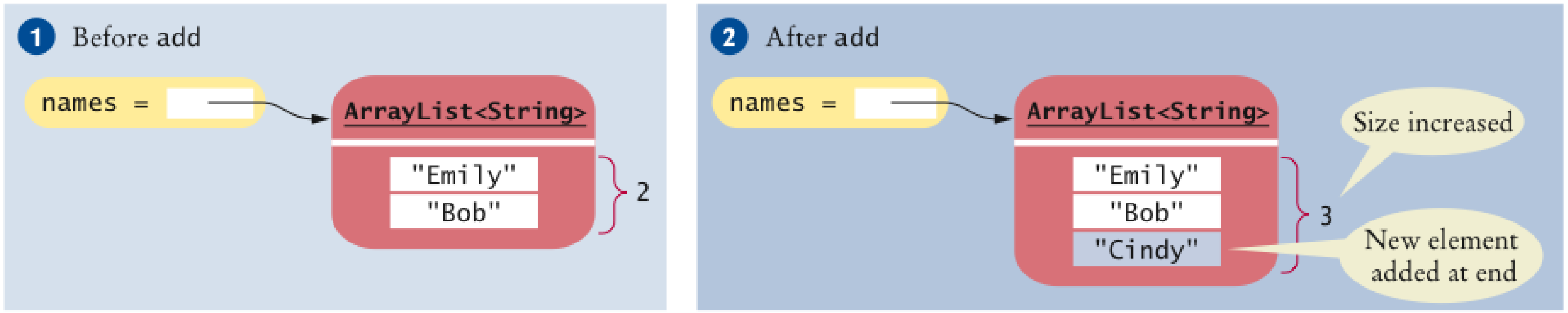


Figure 5 Adding an Element with add

Retrieving Array List Elements

- To obtain the value an element at an index, use the `get` method
- Index starts at 0
- `String name = names.get(2);`

```
// gets the third element of the array list
```

- Bounds error if index is out of range
- Most common bounds error:

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

Setting Elements

- To set an element to a new value, use the `set` method:

```
names.set(2, "Carolyn");
```

Removing Elements

- To remove an element at an index, use the `remove` method:

```
names.remove(1);
```

Adding and Removing Elements

```
names.add("Emily");
names.add("Bob");
names.add("Cindy");
names.set(2, "Carolyn"); 1
names.add(1, "Ann"); 2
names.remove(1); 3
```

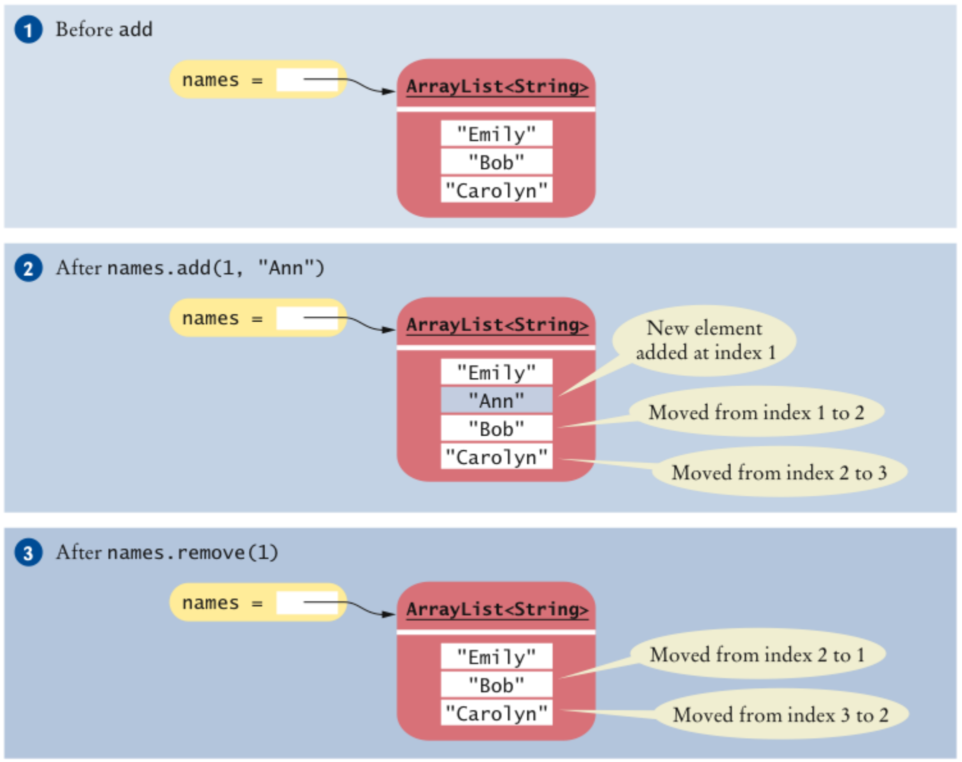


Figure 6 Adding and Removing Elements in the Middle of an Array List

Working with Array Lists

`ArrayList<String> names =`
`   new ArrayList<String>();`
	Constructs an empty array list that can hold strings.

`names.add("Ann");`
`names.add("Cindy");`
	Adds elements to the end.

`System.out.println(names);`	Prints `[Ann, Cindy]`.
`names.add(1, "Bob");`	Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.
`names.remove(0);`	Removes the element at index 0. `names` is now `[Bob, Cindy]`.
`names.set(0, "Bill");`	Replaces an element with a different value. `names` is now `[Bill, Cindy]`.

Working with Array Lists (cont.)

`String name = names.get(i);`	Gets an element.

`String last =`
`   names.get(names.size() - 1);`
	Gets the last element.

`ArrayList<Integer> squares =`
`   new ArrayList<Integer>();`
`for (int i = 0; i < 10; i++)`
`{`
`   squares.add(i * i);`
`}`
	Constructs an array list holding the first ten squares.

ch06/arraylist/ArrayListTester.java

```
import java.util.ArrayList;

/**
   This program tests the ArrayList class.
*/
public class ArrayListTester
{
   public static void main(String[] args)
   {
      ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
      accounts.add(new BankAccount(1001));
      accounts.add(new BankAccount(1015));
      accounts.add(new BankAccount(1729));
      accounts.add(1, new BankAccount(1008));
      accounts.remove(0);

      System.out.println("Size: " + accounts.size());
      System.out.println("Expected: 3");
      BankAccount first = accounts.get(0);
      System.out.println("First account number: "
            + first.getAccountNumber());
      System.out.println("Expected: 1008");
      BankAccount last = accounts.get(accounts.size() - 1);
      System.out.println("Last account number: "
            + last.getAccountNumber());
      System.out.println("Expected: 1729");
   }
}
```

ch06/arraylist/BankAccount.java

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   private int accountNumber;
   private double balance;

   /**
      Constructs a bank account with a zero balance.
      @param anAccountNumber the account number for this account
   */
   public BankAccount(int anAccountNumber)
   {
      accountNumber = anAccountNumber;
      balance = 0;
   }

```

Continued

ch06/arraylist/BankAccount.java (cont.)

```
   /**
      Constructs a bank account with a given balance
      @param anAccountNumber the account number for this account
      @param initialBalance the initial balance
   */
   public BankAccount(int anAccountNumber, double initialBalance)
   {
      accountNumber = anAccountNumber;
      balance = initialBalance;
   }

   /**
      Gets the account number of this bank account.
      @return the account number
   */
   public int getAccountNumber()
   {
      return accountNumber;
   }

```

Continued

ch06/arraylist/BankAccount.java (cont.)

```
   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      double newBalance = balance + amount;
      balance = newBalance;
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      double newBalance = balance - amount;
      balance = newBalance;
   }

```

Continued

ch06/arraylist/BankAccount.java (cont.)

```
    /**
       Gets the current balance of the bank account.
       @return the current balance
    */
    public double getBalance()
    {
       return balance;
    }
 }
```

Program Run:

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

Self Check 6.3

How do you construct an array of 10 strings? An array list of strings?

Answer:

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

Self Check 6.4

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);
```

Answer: `names` contains the strings `"B"` and `"C"` at positions 0 and 1

Wrapper Classes

- For each primitive type there is a **wrapper class** for storing values of that type:

```
Double d = new Double(29.95);
```

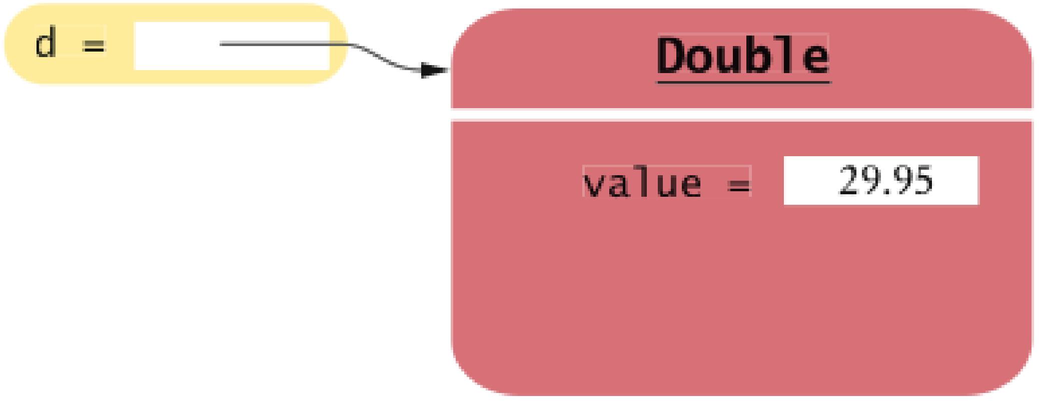


Figure 7 An Object of a Wrapper Class

- Wrapper objects can be used anywhere that objects are required instead of primitive type values:

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

Wrappers

There are wrapper classes for all eight primitive types:

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

Auto-boxing

- **Auto-boxing:** Automatic conversion between primitive types and the corresponding wrapper classes:

```
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();
```

- Auto-boxing even works inside arithmetic expressions:

```
d = d + 1;
```

  Means:

  - *auto-unbox* `d` *into a* `double`
  - *add* `1`
  - *auto-box the result into a new* `Double`
  - *store a reference to the newly created wrapper object in* `d`

Auto-boxing and Array Lists

- To collect numbers in an array list, use the wrapper type as the type parameter, and then rely on auto-boxing:

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

- Storing wrapped numbers is quite inefficient
  - *Acceptable if you only collect a few numbers*
  - *Use arrays for long sequences of numbers or characters*

Self Check 6.5

What is the difference between the types `double` and `Double`?

Answer: `double` is one of the eight primitive types. `Double` is a class type.

Self Check 6.6

Suppose `values` is an `ArrayList<Double>` of size > 0. How do you increment the element with index 0?

Answer:

```
values.set(0, values.get(0) + 1);
```

The Enhanced `for` Loop

- Traverses all elements of a collection:

```
double[] values = ...;
double sum = 0;
for (double element : values)
{
   sum = sum + element;
}
```

- Read the loop as “for each `element` in `values`”
- Traditional alternative:

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

The Enhanced `for` Loop

- Works for `ArrayLists` too:

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

- Equivalent to the following ordinary `for` loop:

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

The Enhanced `for` Loop

- The "for each loop" does not allow you to modify the contents of an array:

```
for (double element : values)
{
   element = 0;
   // ERROR—this assignment does not
   // modify array element
}
```

- Must use an ordinary `for` loop:

```
for (int i = 0; i < values.length; i++)
{
   values[i] = 0; // OK
}
```

Self Check 6.7

Write a "for each" loop that prints all elements in the array `values`

Answer:

```
for (double element : values)
    System.out.println(element);
```

Self Check 6.8

What does this "for each" loop do?

```
int counter = 0;
for (BankAccount a : accounts)
{
   if (a.getBalance() == 0) { counter++; }
}
```

Answer: It counts how many accounts have a zero balance.

Partially Filled Arrays

- Array length = maximum number of elements in array
- Usually, array is partially filled
- Need companion variable to keep track of current size
    - *Uniform naming convention:*

    ```
    final int VALUES_LENGTH = 100;
    double[] values = new double[VALUES_LENGTH];
    int valuesSize = 0;
    ```

- Update `valuesSize` as array is filled:

```
values[valuesSize] = x;
valuesSize++;
```

Partially Filled Arrays

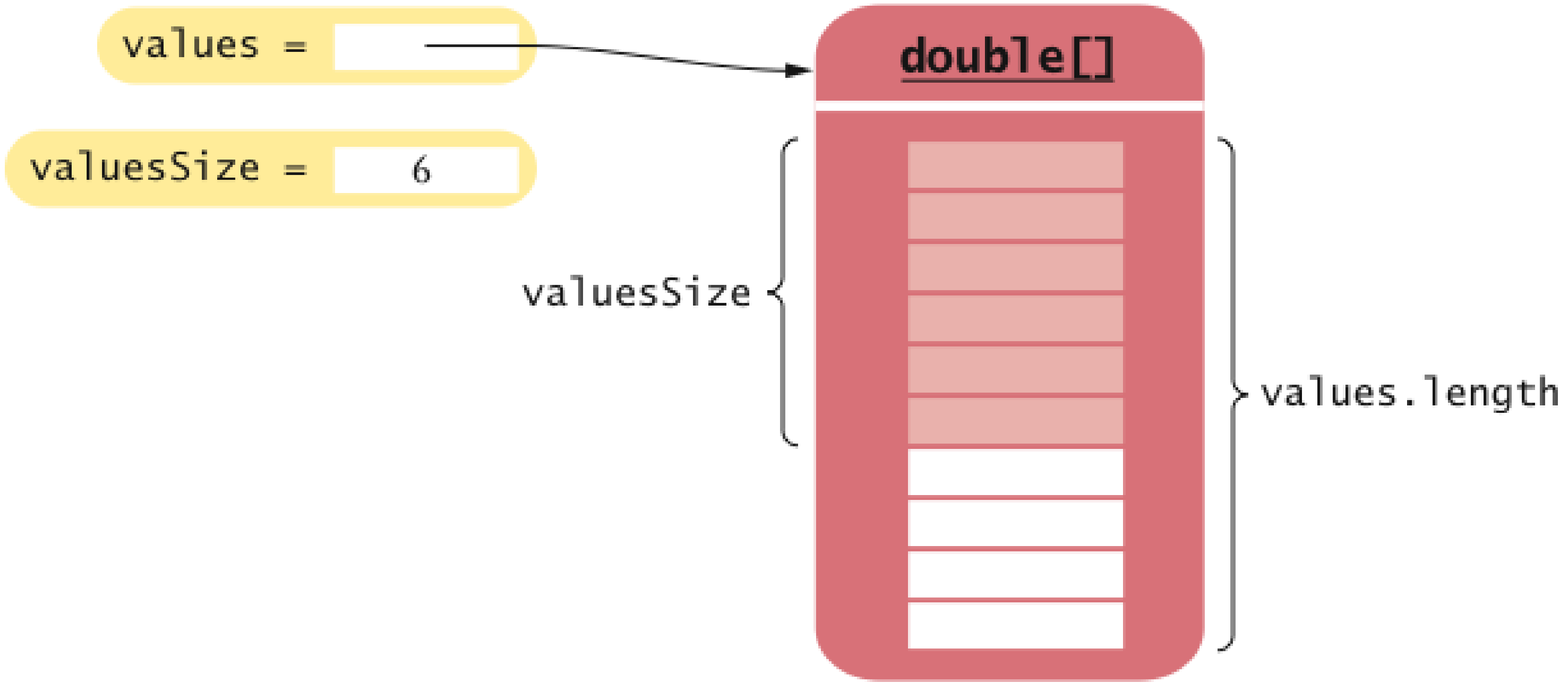


Figure 8 A Partially Filled Array

Partially Filled Arrays

- Example: Read numbers into a partially filled array:

```
int valuesSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
   if (valuesSize < values.length)
   {
      values[valuesSize] = in.nextDouble();
      valuesSize++;
   }
}
```

- To process the gathered array elements, use the companion variable, not the array length:

```
for (int i = 0; i < valuesSize; i++)
{
   System.out.println(values[i]);
}
```

Self Check 6.9

Write a loop to print the elements of the partially filled array `values` in reverse order, starting with the last element.

Answer:

```
for (int i = valuesSize - 1; i >= 0; i--)
   System.out.println(values[i]);
```

Self Check 6.10

How do you remove the last element of the partially filled array `values`?

Answer:

```
valuesSize--;
```

Self Check 6.11

Why would a programmer use a partially filled array of numbers instead of an array list?

Answer: You need to use wrapper objects in an `ArrayList<Double>`, which is less efficient.

Common Array Algorithm: Filling

- Fill an array with zeroes:

```
for (int i = 0; i < values.length; i++)
{
   values[i] = 0;
}
```

- Fill an array list with squares (0, 1, 4, 9, 16, ...):

```
for (int i = 0; i < values.size(); i++)
{
   values.set(i, i * i;
}
```

Common Array Algorithm: Computing Sum and Average

- To compute the sum of all elements, keep a running total:

```
double total = 0;
for (double element : values)
{
   total = total + element;
}
```

- To obtain the average, divide by the number of elements:

```
double average = total /values.size();
// for an array list
```

- Be sure to check that the size is not zero

Common Array Algorithm: Counting Matches

- Check all elements and count the matches until you reach the end
- Example: Count the number of accounts whose balance is at least as much as a given threshold:

```
public class Bank
{
   private ArrayList<BankAccount> accounts;

   public int count(double atLeast)
   {
      int matches = 0;
      for (BankAccount account : accounts)
      {
         if (account.getBalance() >= atLeast) matches++;
      }
      return matches;
   }
   . . .
}
```

Common Array Algorithm: Finding the Maximum or Minimum

- Initialize a candidate with the starting element
- Compare candidate with remaining elements
- Update it if you find a larger or smaller value

Common Array Algorithm: Finding the Maximum or Minimum

- Example: Find the account with the largest balance in the bank:

```
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;
```

- Works only if there is at least one element in the array list — if list is empty, return `null`:

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

Common Array Algorithm: Searching for a Value

- Check all elements until you have found a match
- Example: Determine whether there is a bank account with a particular account number in the bank:

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

Common Array Algorithm: Searching for a Value

- The process of checking all elements until you have found a match is called a **linear search**

Common Array Algorithm: Locating the Position of an Element

- Problem: Locate the position of an element so that you can replace or remove it
- Use a variation of the linear search algorithm, but remember the position instead of the matching element
- Example: Locate the position of the first element that is larger than 100:

```
int pos = 0;
boolean found = false;
while (pos < values.size() && !found)
{
   if (values.get(pos) > 100) { found = true; }
   else { pos++; }
}
if (found) { System.out.println("Position: " + pos); }
else { System.out.println("Not found"); }
```

Common Array Algorithm: Removing an Element

- Array list ⇒ use method `remove`
- Unordered array ⇒
    1. *Overwrite the element to be removed with the last element of the array*
    2. *Decrement the variable tracking the size of the array*

```
values[pos] = values[valuesSize - 1];
valuesSize--;
```

Common Array Algorithm: Removing an Element

- Ordered array ⇒

   1. *Move all elements following the element to be removed to a lower index*
   2. *Decrement the variable tracking the size of the array*

```
for (int i = pos; i < valuesSize - 1; i++)
{
   values[i] = values[i + 1];
}
valuesSize--;
```

Common Array Algorithm: Removing an Element

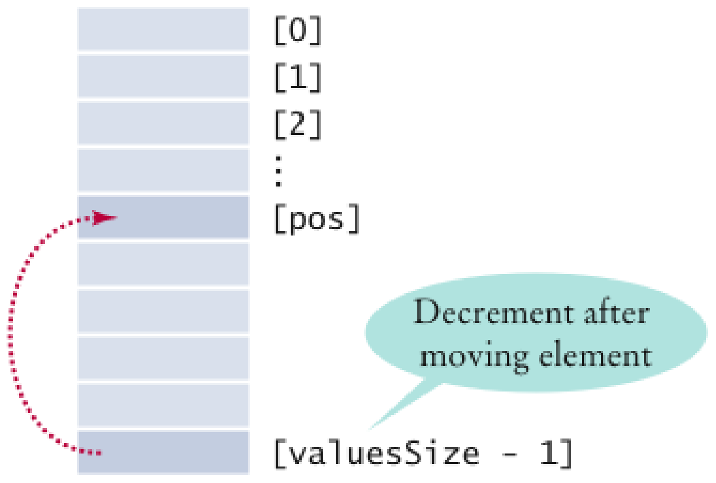


Figure 9
Removing an Element in an Unordered Array

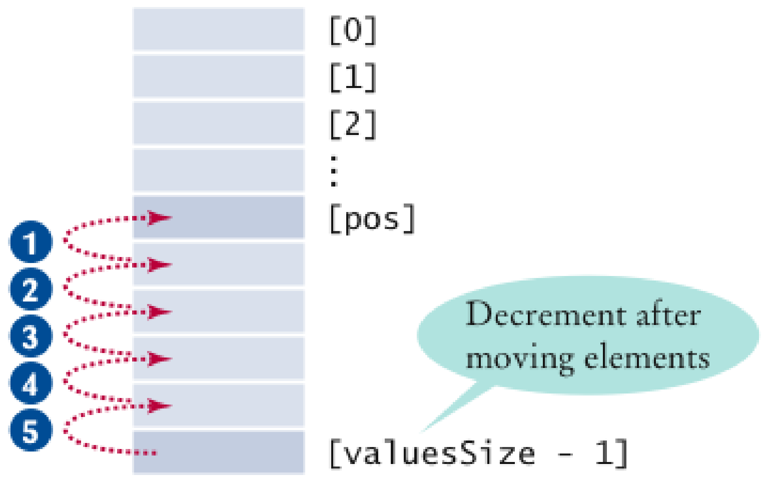


Figure 10
Removing an Element in an Ordered Array

Common Array Algorithm: Inserting an Element

- Array list ⇒ use method `add`
- Unordered array ⇒
    1. *Insert the element as the last element of the array*
    2. *Increment the variable tracking the size of the array*

```
if (valuesSize < values.length)
{
   values[valuesSize] = newElement;
   valuesSize++;
}
```

Common Array Algorithm: Inserting an Element

- Ordered array ⇒
  1. *Start at the end of the array, move that element to a higher index, then move the one before that, and so on until you finally get to the insertion location*
  2. *Insert the element*
  3. *Increment the variable tracking the size of the array*

```
if (valuesSize < values.length)
{
   for (int i = valuesSize; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = newElement;
   valuesSize++;
}
```

Common Array Algorithm: Inserting an Element

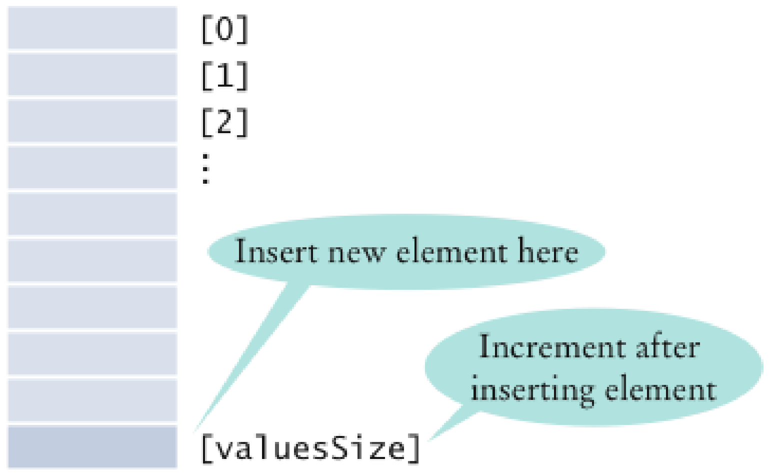


Figure 11
Inserting an Element in an Unordered Array

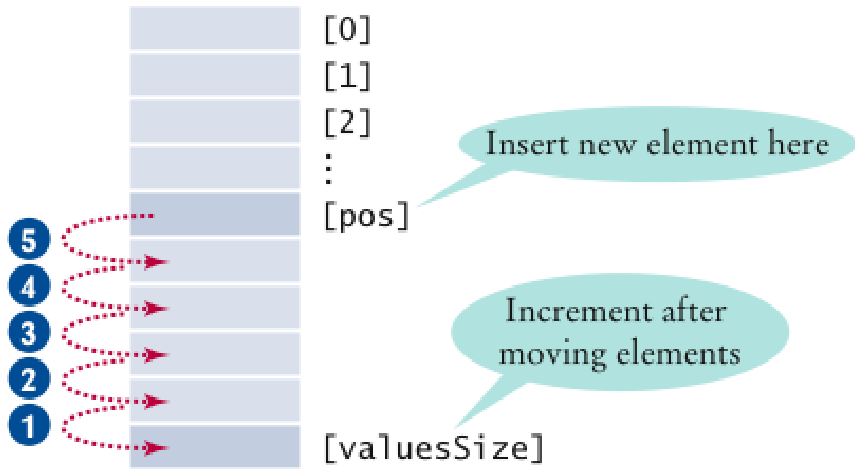
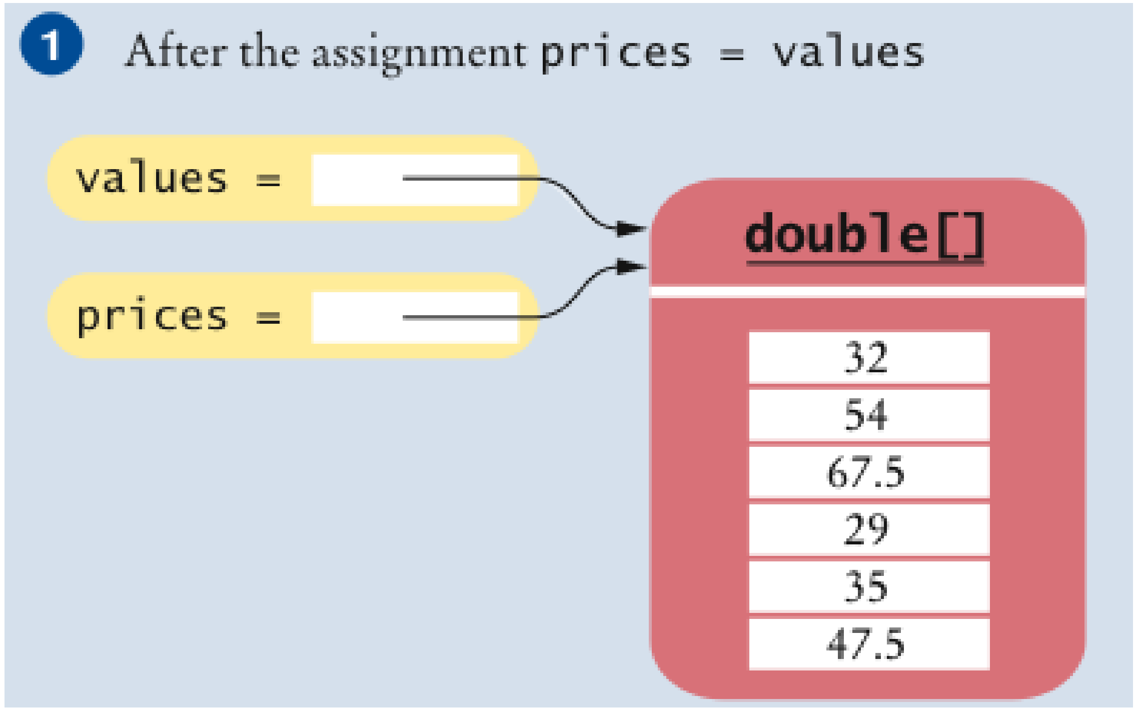


Figure 12
Inserting an Element in an Ordered Array

Common Array Algorithm: Copying an Array

- Copying an array variable yields a second reference to the same array:

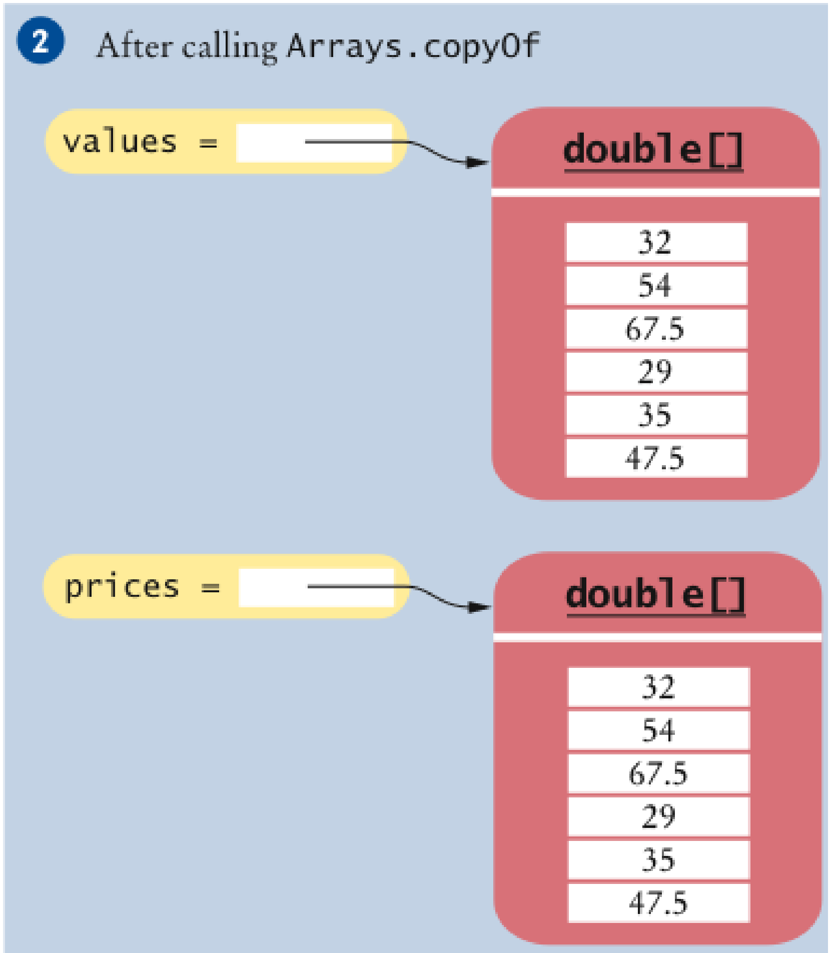
```
double[] values = new double[6];
. . .  // Fill array
double[] prices = values; 1
```



Common Array Algorithm: Copying an Array

- To make a true copy of an array, call the `Arrays.copyOf` method:

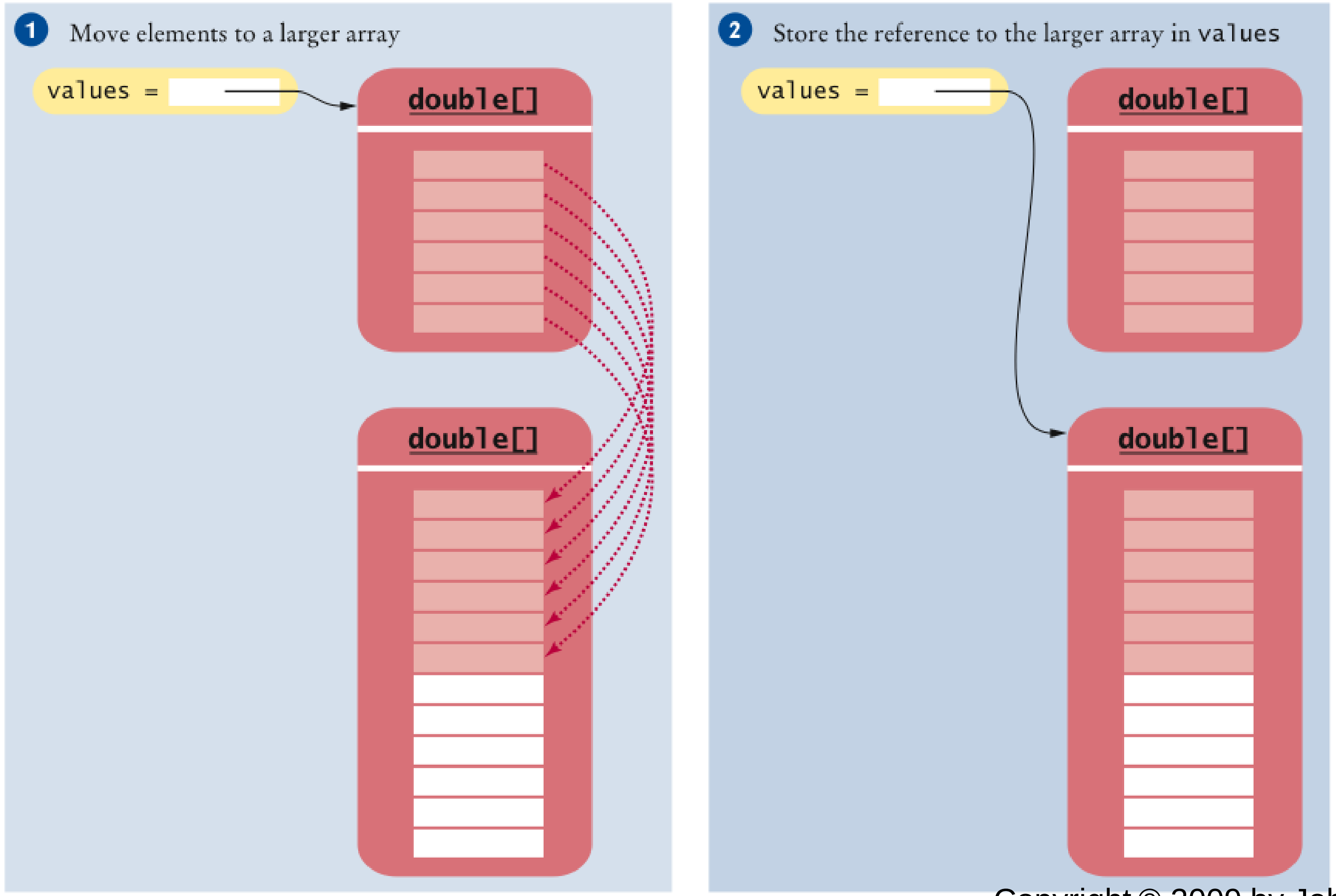
```
double[] prices = Arrays.copyOf(values, values.length);
```



Common Array Algorithm: Copying an Array

- To grow an array that has run out of space, use the `Arrays.copyOf` method:

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



1 Move elements to a larger array
values =
double[]
double[]
2 Store the reference to the larger array in values
values =
double[]
double[]

Figure 14 Growing an Array

Common Array Algorithm: Growing an Array

- Example: Read an arbitrarily long sequence numbers into an array, without running out of space:

```
int valuesSize = 0;
while (in.hasNextDouble())
{
   if (valuesSize == values.length)
      values = Arrays.copyOf(values, 2 * values.length);
   values[valuesSize] = in.nextDouble();
   valuesSize++;
}
```

Common Array Algorithm: Printing Element Separators

- When you display the elements of an array or array list, you usually want to separate them:

```
Ann | Bob | Cindy
```

- When you display the elements of an array or array list, you usually want to separate them
- Print the separator before each element *except the initial one* (with index 0):

```
for (int i = 0; i < names.size(); i++)
{
   if (i > 0)
   {
      System.out.print(" | ");
   }
   System.out.print(names.get(i));
}
```

ch06/bank/Bank.java

- `Bank` class stores an array list of bank accounts
- Methods of the `Bank` class use some of the previous algorithms:

```
import java.util.ArrayList;

/**
   This bank contains a collection of bank accounts.
*/
public class Bank
{
   private ArrayList<BankAccount> accounts;

   /**
      Constructs a bank with no bank accounts.
   */
   public Bank()
   {
      accounts = new ArrayList<BankAccount>();
   }

```

Continued

ch06/bank/Bank.java (cont.)

```
   /**
      Adds an account to this bank.
      @param a the account to add
   */
   public void addAccount(BankAccount a)
   {
      accounts.add(a);
   }

   /**
      Gets the sum of the balances of all accounts in this bank.
      @return the sum of the balances
   */
   public double getTotalBalance()
   {
      double total = 0;
      for (BankAccount a : accounts)
      {
         total = total + a.getBalance();
      }
      return total;
   }

```

Continued

ch06/bank/Bank.java (cont.)

```
   /**
      Counts the number of bank accounts whose balance is at
      least a given value.
      @param atLeast the balance required to count an account
      @return the number of accounts having least the given balance
   */
   public int countBalancesAtLeast(double atLeast)
   {
      int matches = 0;
      for (BankAccount a : accounts)
      {
         if (a.getBalance() >= atLeast) matches++; // Found a match
      }
      return matches;
   }

```

Continued

ch06/bank/Bank.java (cont.)

```
   /**
      Finds a bank account with a given number.
      @param accountNumber the number to find
      @return the account with the given number, or null if there
      is no such account
   */
   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
   }

```

Continued

ch06/bank/Bank.java (cont.)

```
   /**
      Gets the bank account with the largest balance.
      @return the account with the largest balance, or null if the
      bank has no accounts
   */
   public BankAccount getMaximum()
   {
      if (accounts.size() == 0) return null;
      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;
   }
}
```

ch06/bank/BankTester.java

```java
/**
    This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      double threshold = 15000;
      int count = firstBankOfJava.countBalancesAtLeast(threshold);
      System.out.println("Count: " + count);
      System.out.println("Expected: 2");

```

Continued

ch06/bank/BankTester.java (cont.)

```java
      int accountNumber = 1015;
      BankAccount account = firstBankOfJava.find(accountNumber);
      if (account == null)
         System.out.println("No matching account");
      else
         System.out.println("Balance of matching account: "
               + account.getBalance());
      System.out.println("Expected: 10000");

      BankAccount max = firstBankOfJava.getMaximum();
      System.out.println("Account with largest balance: "
            + max.getAccountNumber());
      System.out.println("Expected: 1001");
   }
}
```

Program Run:

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

Self Check 6.12

What does the `find` method do if there are two bank accounts with a matching account number?

Answer: It returns the first match that it finds.

Self Check 6.15

The following replacement has been suggested for the algorithm that prints element separators:

```
System.out.print(names.get(0));
for (int i = 1; i < names.size(); i++)
   System.out.print(" | " + names.get(i));
```

What is problematic about this suggestion?

Answer: If `names` happens to be empty, the first line causes a bounds error.

Regression Testing

- **Test suite:** a set of tests for repeated testing
- **Cycling:** bug that is fixed but reappears in later versions
- **Regression testing:** repeating previous tests to ensure that known failures of prior versions do not appear in new versions

ch06/regression/BankTester.java

```java
import java.util.Scanner;

/**
   This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      Scanner in = new Scanner(System.in);

      double threshold = in.nextDouble();
      int c = firstBankOfJava.count(threshold);
      System.out.println("Count: " + c);
      int expectedCount = in.nextInt();
      System.out.println("Expected: " + expectedCount);

```

Continued

ch06/regression/BankTester.java (cont.)

```
      int accountNumber = in.nextInt();
      BankAccount a = firstBankOfJava.find(accountNumber);
      if (a == null)
         System.out.println("No matching account");
      else
      {
         System.out.println("Balance of matching account: " + a.getBalance());
         int matchingBalance = in.nextInt();
         System.out.println("Expected: " + matchingBalance);
      }
   }
}
```

Regression Testing: Input Redirection

- Store the inputs in a file
- ch06/regression/input1.txt:

```
15000
2
1015
10000
```

- Type the following command into a shell window:

```
java BankTester < input1.txt
```

- Program Run:

```
Count: 2
Expected: 2
Balance of matching account: 10000
Expected: 10000
```

Regression Testing: Output Redirection

- Output redirection:

```
java BankTester < input1.txt > output1.txt
```

Self Check 6.16

Suppose you modified the code for a method. Why do you want to repeat tests that already passed with the previous version of the code?

> **Answer:** It is possible to introduce errors when modifying code.

Self Check 6.17

Suppose a customer of your program finds an error. What action should you take beyond fixing the error?

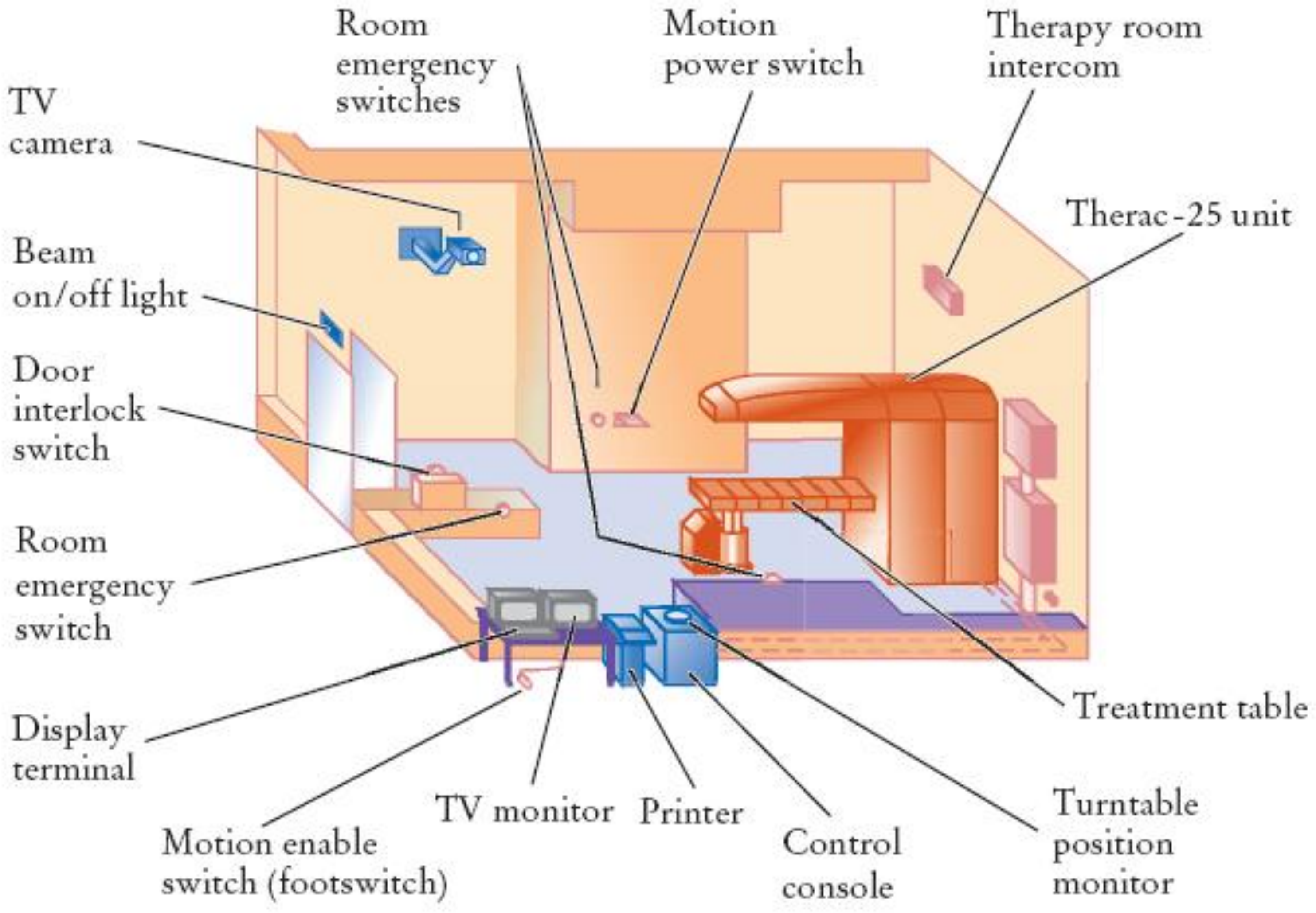
> **Answer:** Add a test case to the test suite that verifies that the error is fixed.

Self Check 6.18

Why doesn’t the `BankTester` program contain prompts for the inputs?

Answer: There is no human user who would see the prompts because input is provided from a file.

Therac-25 Facility



Typical Therac-25 Facility

Two-Dimensional Arrays

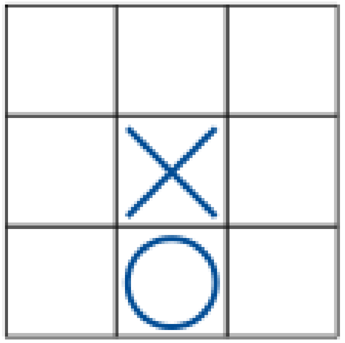


Figure 15 A Tic-Tac-Toe Board

- When constructing a two-dimensional array, specify how many rows and columns are needed:

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

- Access elements with an index pair:

```
board[1][1] = "x";
board[2][1] = "o";
```

Traversing Two-Dimensional Arrays

- It is common to use two nested loops when filling or searching:

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

Traversing Two-Dimensional Arrays

- You can also recover the array dimensions from the array variable:
  - *board.length is the number of rows*
  - *board[0].length is the number of columns*
- Rewrite the loop for filling the tic-tac-toe board:

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

ch06/twodim/TicTacToe.java

```
/**
   A 3 x 3 tic-tac-toe board.
*/
public class TicTacToe
{
   private String[][] board;
   private static final int ROWS = 3;
   private static final int COLUMNS = 3;

   /**
      Constructs an empty board.
   */
   public TicTacToe()
   {
      board = new String[ROWS][COLUMNS];
      // Fill with spaces
      for (int i = 0; i < ROWS; i++)
         for (int j = 0; j < COLUMNS; j++)
            board[i][j] = " ";
   }

```

Continued

ch06/twodim/TicTacToe.java (cont.)

```
   /**
      Sets a field in the board. The field must be unoccupied.
      @param i the row index
      @param j the column index
      @param player the player ("x" or "o")
   */
   public void set(int i, int j, String player)
   {
      if (board[i][j].equals(" "))
         board[i][j] = player;
   }

```

Continued

ch06/twodim/TicTacToe.java (cont.)

```
         Creates a string representation of the board, such as
         |x  o|
         | x |
         |  o|
         @return the string representation
      */
      public String toString()
      {
         String r = "";
         for (int i = 0; i < ROWS; i++)
         {
            r = r + "|";
            for (int j = 0; j < COLUMNS; j++)
               r = r + board[i][j];
            r = r + "|\n";
         }
         return r;
      }
   }
```

ch06/twodim/TicTacToeRunner.java

```java
import java.util.Scanner;

/**
   This program runs a TicTacToe game. It prompts the
   user to set positions on the board and prints out the
   result.
*/
public class TicTacToeRunner
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      String player = "x";
      TicTacToe game = new TicTacToe();
```

Continued

ch06/twodim/TicTacToeRunner.java (cont.)

```java
      boolean done = false;
      while (!done)
      {
         System.out.print(game.toString());
         System.out.print(
               "Row for " + player + " (-1 to exit): ");
         int row = in.nextInt();
         if (row < 0) done = true;
         else
         {
            System.out.print("Column for " + player + ": ");
            int column = in.nextInt();
            game.set(row, column, player);
            if (player.equals("x"))
               player = "o";
            else
               player = "x";
         }
      }
   }
}
```

ch06/twodim/TicTacToeRunner.java (cont.)

Program Run:

```
|     |
|     |
|     |
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
```

Self Check 6.19

How do you declare and initialize a 4-by-4 array of integers?

Answer:

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

Self Check 6.20

How do you count the number of spaces in the tic-tac-toe board?

Answer:

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

Exercises for Chapter 6: Arrays & ArrayLists

Aufgabe 1: Implementieren Sie ein dynamisches Array für einen primitiven Wertetyp als Klasse

In diesem Kapitel wird so ein Array “Partially Filled Array” genannt. Ein dynamisches Array kann automatisch wachsen und man kann Elemente einfügen und Löschen. Implementieren Sie folgende Methoden:

•Konstruktor ArrayListInt()
•Konstruktor ArrayListInt(int initialLength)
•get(int indexOfElementToGet)
•set(int indexOfElementToSet, type elementToBeSet)
•add(type elementToBeAddedAtTheEnd)
•insert(int index, type elementToBeInserted)
•remove(int indexOfElementToBeRemoved)
•toString() zur Konsolenausgabe des ganzen Arrays

Implementieren Sie ein Wachsen und Schrumpfen z.B. indem Sie die Grösse jeweils verdoppeln oder halbieren.

Vergleichen Sie die Performance von Ihrer Klasse mit einer ArrayList desselben Typs als Wrapper-Klasse für die kritischen Methoden add, insert und remove.

Exercises for Chapter 6: Arrays & ArrayLists

Aufgabe 2: Lösen Sie je 2 Aufgaben aus den CodingBat Gruppen Array-1 und Array-2