Objective

Arrays and Array Lists

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

Study following sections from your text book:
7.1 ,7.2, 7.7, and 7.8 from your text book.
Optional: study sections 7.3 … 7.6

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];
```

Arrays

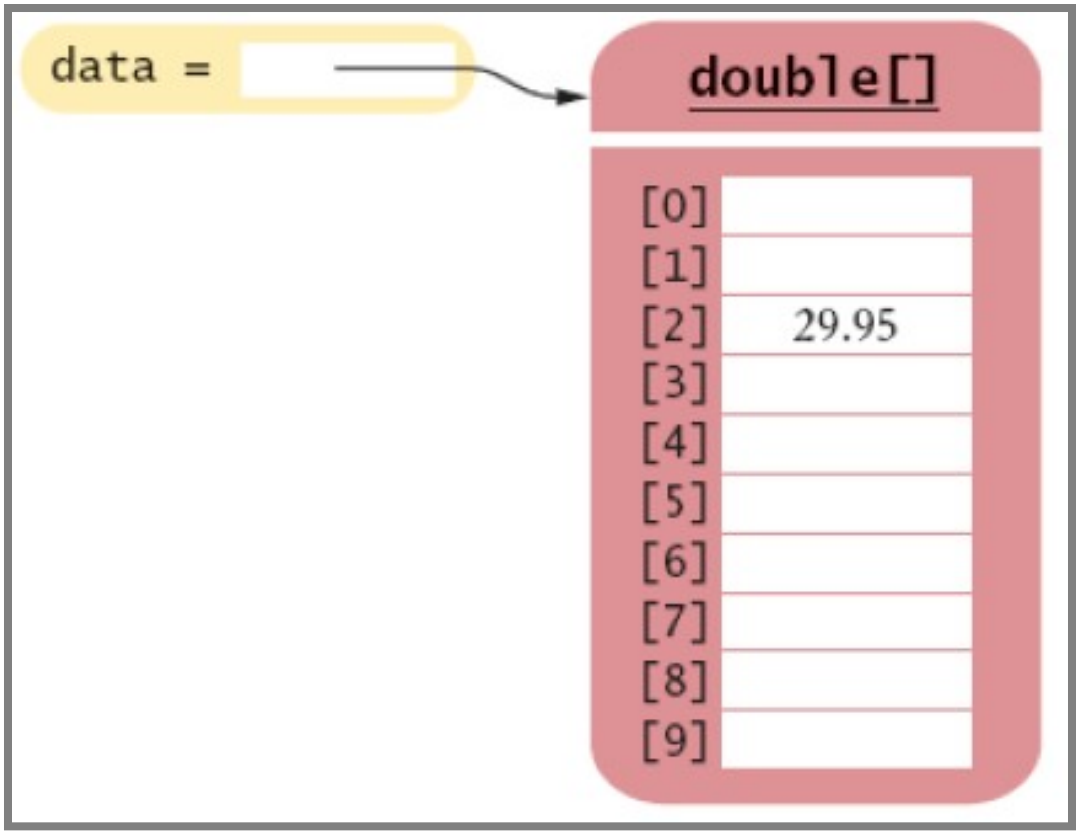
When array is created, all values are initialized depending on array type:

- Numbers: 0
- Boolean: `false`
- `char: '\u000'`
- Object References: `null`

Arrays

Use [ ] to access an element

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



Arrays

Using the value stored:

```
System.out.println("data[4]= " + data[4]);
```

Arrays are Object in Java

Get array length as `data.length` (Not a method!)

Index values range from `0` to `length - 1`

Arrays

- Accessing a nonexistent element results in a bounds error

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

- Limitation: Arrays have fixed length

Exercise

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]);
```

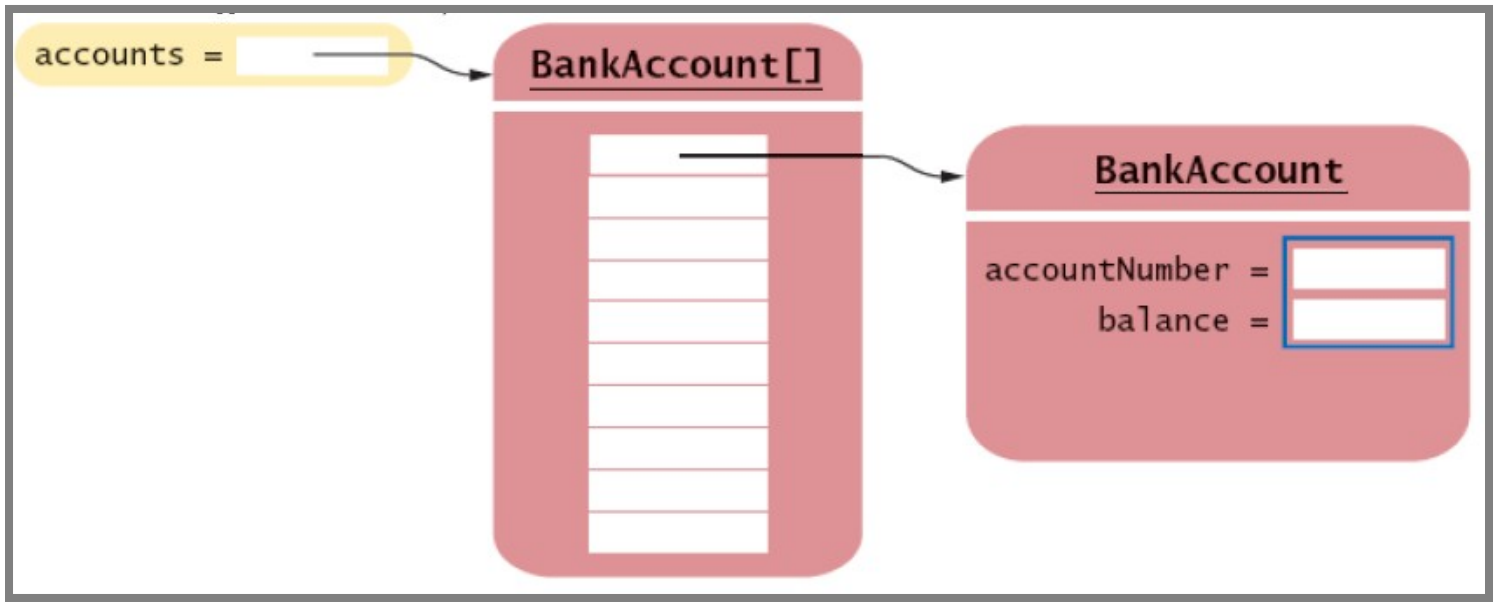
3.
```
double[] c;
System.out.println(c[0]);
```

Answers

- 0
- a run-time error: array index out of bounds
- a compile-time error: c is not initialized

Array of Objects

```
BankAccount[] accounts= new BankAccount[10];
accounts[0] = new BankAccount();
```



Array Lists

- The `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

Array Lists

The **`ArrayList`** class is a generic class:

`ArrayList<T>` collects objects of type **`T`**:

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

`size()` method yields number of elements

Adding Elements

`set` overwrites an existing value

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

`add` adds a new value before the index i

```
accounts.add(i, a)
```

`remove` removes an element at index i

```
Accounts.remove(i)
```

`get` returns the element at index i

```
Accounts.get(i)
```

Retrieving Array List Elements

Most common bounds error:

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

Example1: 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: {
       private int accountNumber;
67:    private double balance;
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:
```

Example1

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:    }
```

Example1

File: BankAccount.java

```
36:    /**
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:    /**
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;
55:    }
```

Example1

File: BankAccount.java

```
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:
68: }
```

Example1

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);
```

Example1

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: }
```

Output

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

Example1

Exercise

1. How do you construct an array of 10 strings? An ArrayList of strings?
2. 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);
```

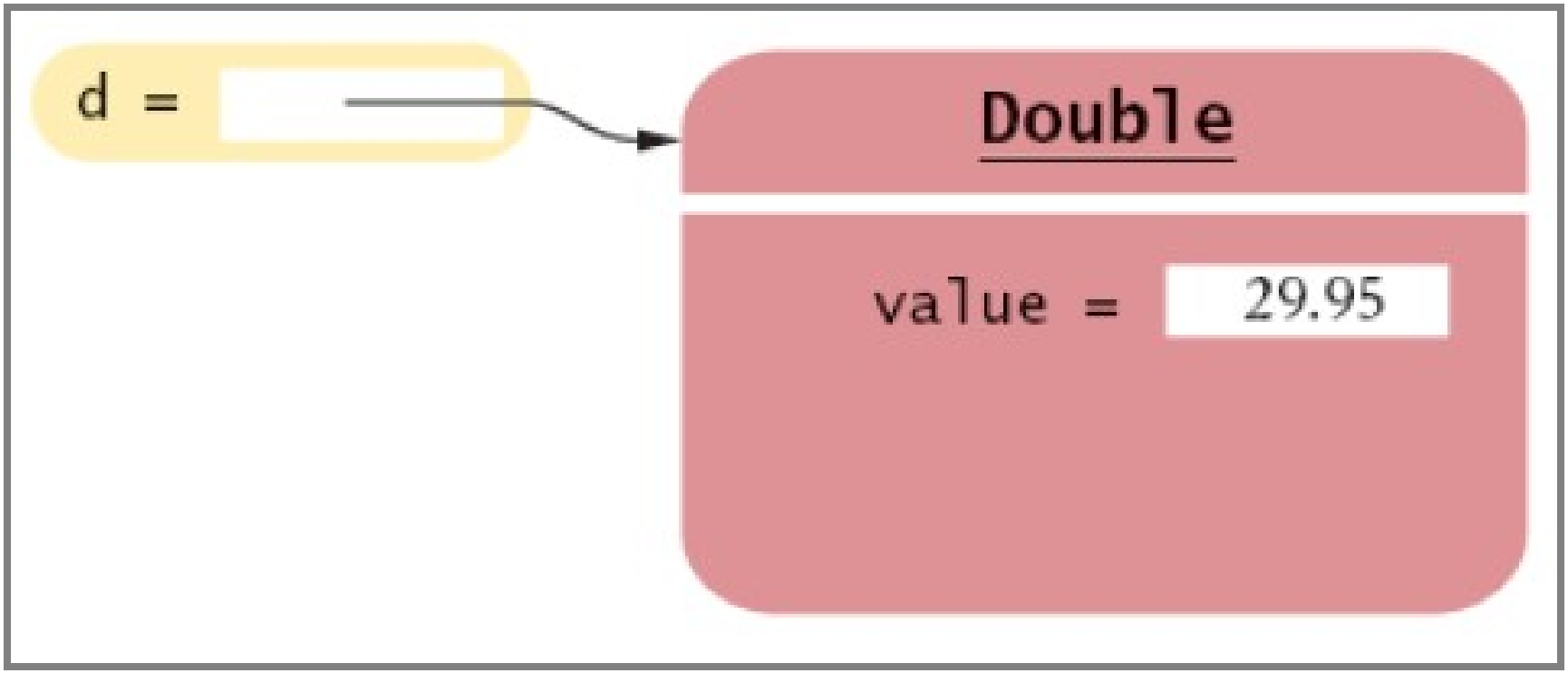
Wrappers

- You cannot insert primitive types directly into array lists
- To treat primitive type values as objects, you must use wrapper classes:

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

Continued…

Wrappers



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: Starting with Java 5.0, conversion between primitive types and the corresponding wrapper classes is automatic.

```
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();
```

Continued…

Wrappers

So we can use

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

Instead of

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

Continued…

Auto-boxing

Auto-boxing even works inside arithmetic expressions

```
Double d = 29.95;

Double e = d + 1;
```

Means:

- Auto-box double 29.95 into a Double object
- 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 **`e`**

Exercise

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

- Suppose data is an `ArrayList<Double>` of size > 0

How do you increment the element with index 0?

How do you increment each element by 1.

`for` Loop

Traditional:

```
double[] data = {1,2.6,4.5,7,7.0};
double sum = 0;
double e = 0;

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

`for` each Loop

Traverses all elements of a collection:

```
double[] data = {1,2.6,4.5,7,7.0};
double sum = 0;


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

You should read this loop as "for each e in data"

Note: There is no index element

Never combine for each loop with index element.

Continued…

for Loop

Traditional:

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

For each Loop

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

Continued…

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

1. Write a "for each" loop that prints all elements in the array data

```
double[] data = . . .;
```

2. 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;
```

Example

Count number of accounts that their balance is at least equal to *atLeast*.

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

}
```

Example: Finding a Value

Find the account with account number equal to *accountNumber*.

```
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
 }
   . . .
}
```

Example:
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

Continued…

Example:
Finding the Maximum or Minimum

```
if (accounts.size() == 0) // Why we need this?
   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;
```

Example2: 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: {
       private ArrayList<BankAccount> accounts;
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:    */
```

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:
```

Example2

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 Match
51:   }
52:   return matches;
53: }
54:
```

Example2

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)
          // Found a match
66:             return a;
67:       }
68:       return null; // No match in the entire array list
69:    }
70:
```

Example2

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:
90: }
```

Example2

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);
```

Example2

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: }
```

Example2

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.
```

Example2

Exercise

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

Answers

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

Copying Arrays: Copying Array References

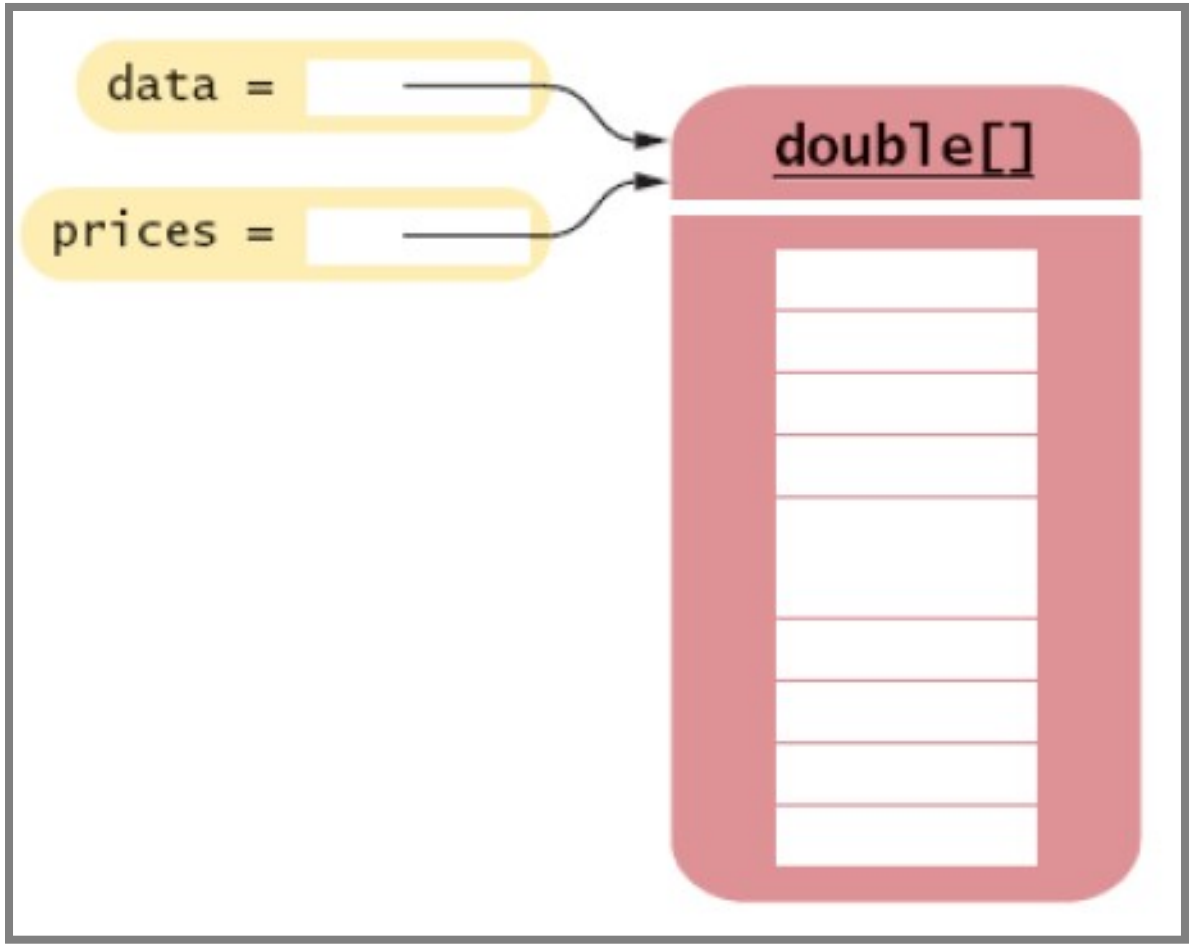
Copying an array variable yields a second reference to the same array

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

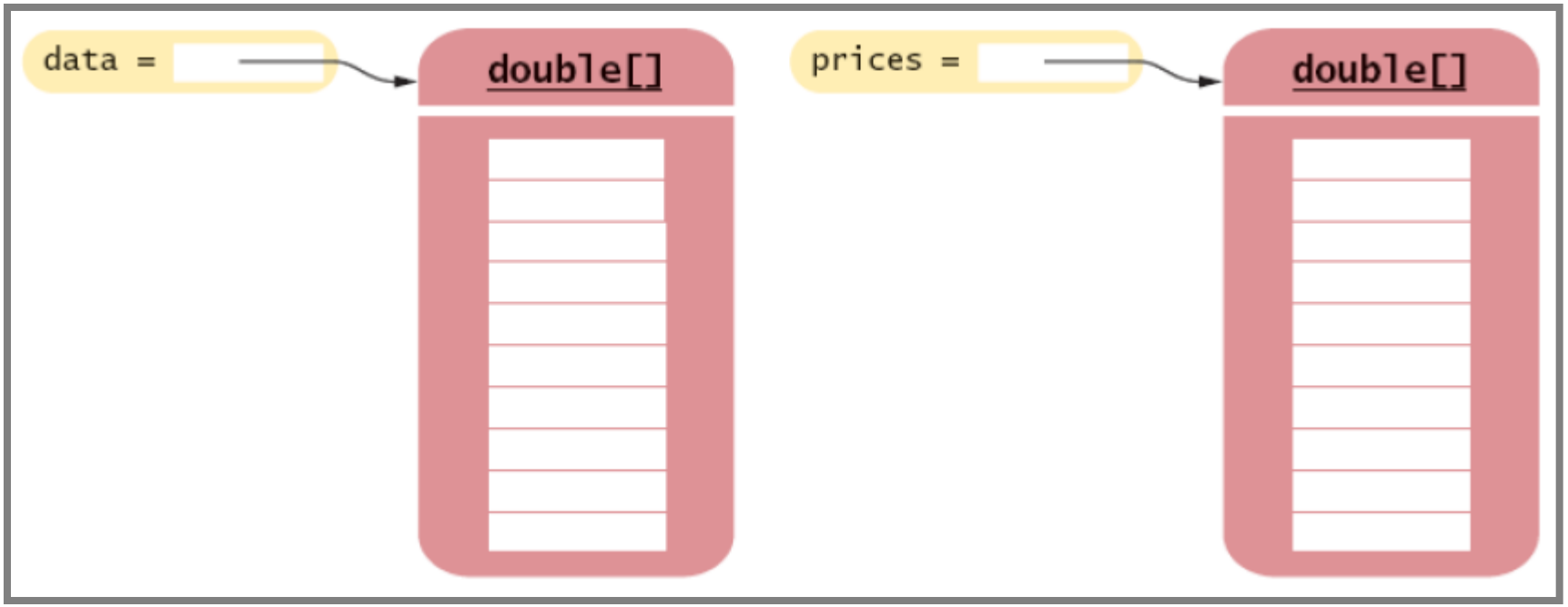
Shallow Copy

Continued…

Shallow Copy:
Copying Array References



Deep Copy : Cloning Arrays



Deep Copy: Cloning Arrays

```
Clone or use System.arraycopy

double[] b = (double[]) a.clone();
double[] to = new double[from.length];

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

Note: Clone is not a reliable method. Why?

Growing an Array

- Create a new, larger array.

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

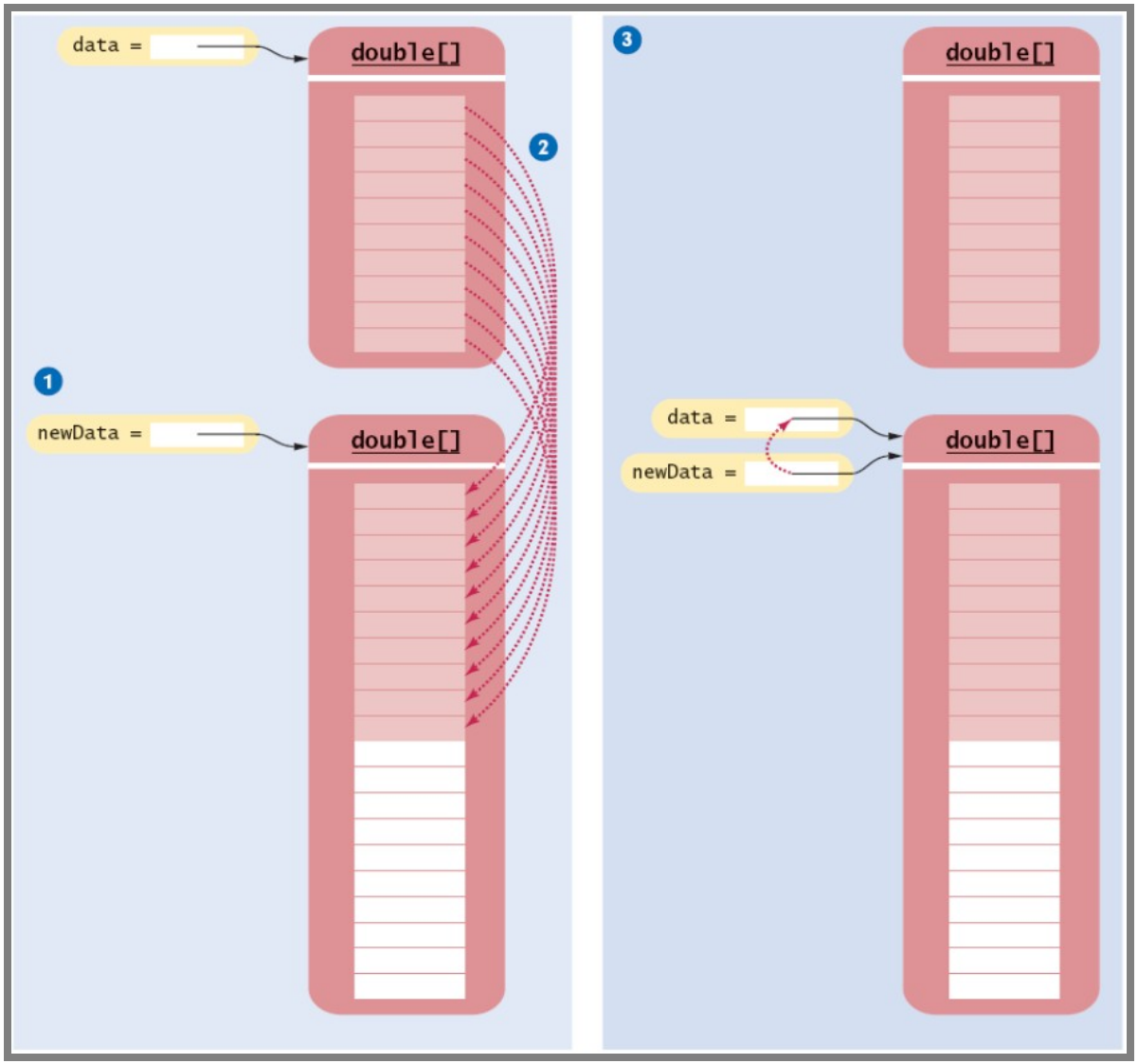
- Copy all elements into the new array

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

- Store the reference to the new array in the array variable

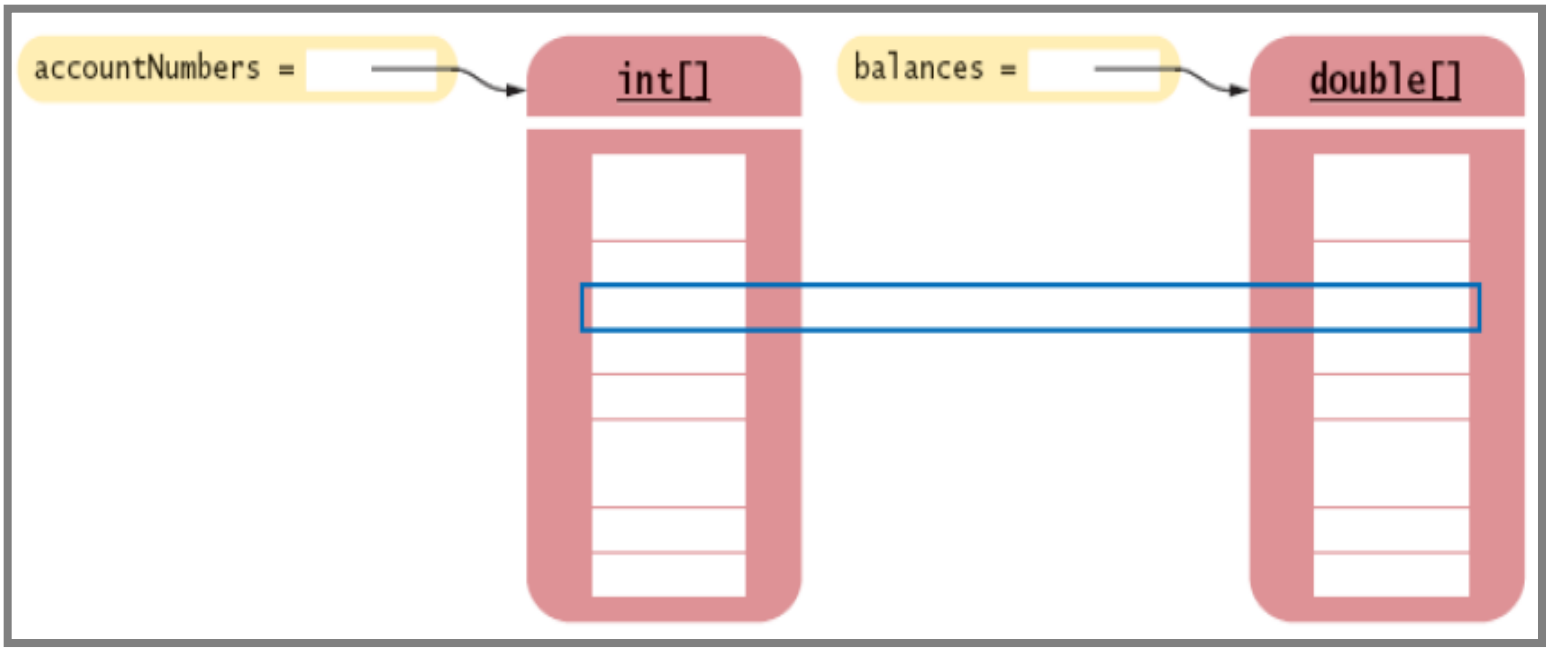
```
data = newData;
```

Growing an Array



Make Parallel Arrays into Arrays of Objects

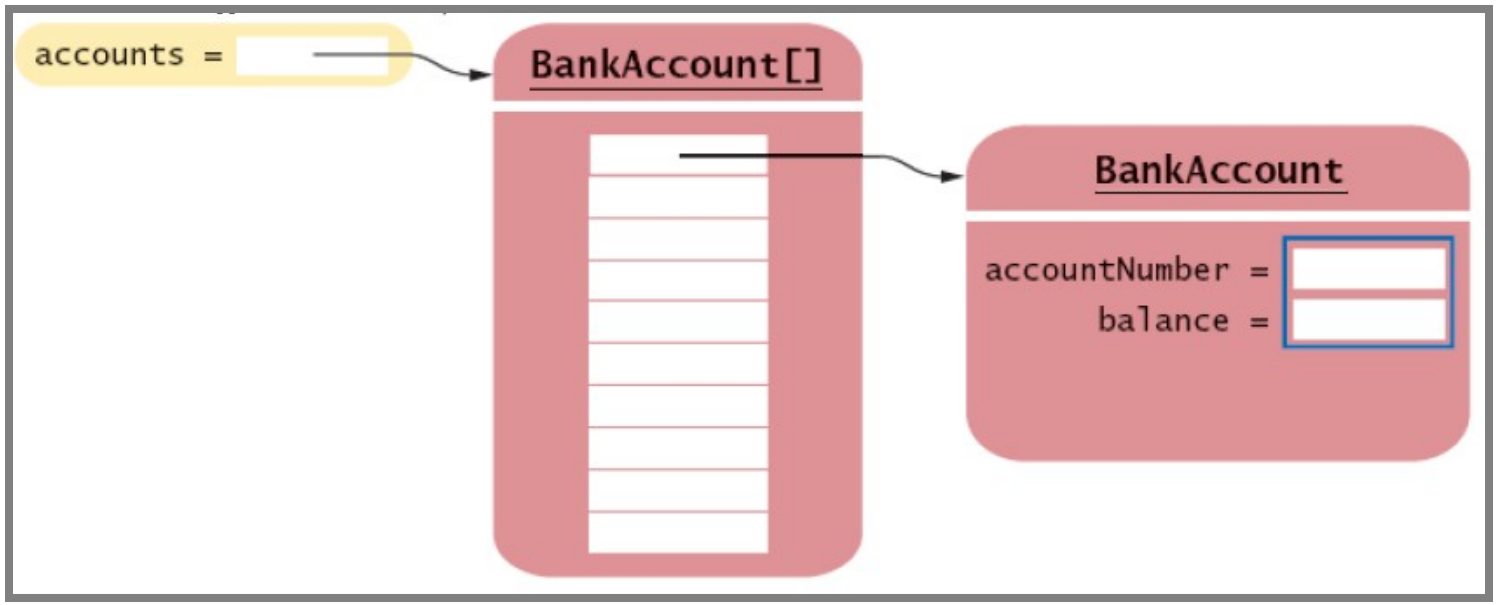
```
// Never use Parallel Arrays
int[] accountNumbers;
double[] balances;
```



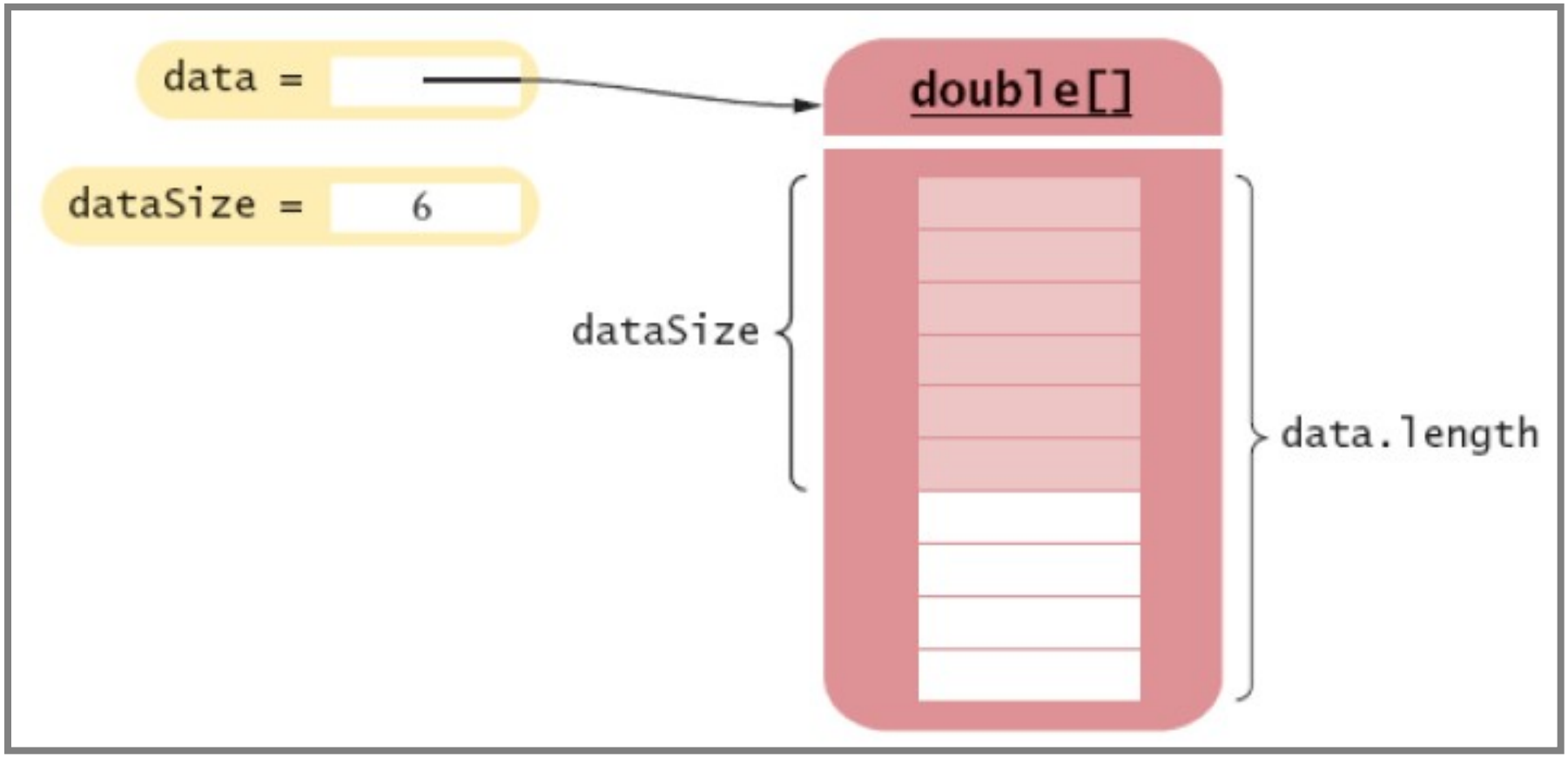
Make Parallel Arrays into Arrays of Objects

- Avoid parallel arrays by changing them into arrays of objects:

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



Partially Filled Arrays



Notes

```
import java.util.Arrays;
String[] arr={"one","two","three","four"};
System.out.println(Arrays.toString(arr));
[one, two, three, four]

ArrayList<Integer> numbers = new
ArrayList<Integer>();
numbers.add(10);
numbers.add(15);
numbers.add(20);
System.out.println(numbers);
[10, 15, 20]
```

Notes

```
import java.util.Arrays;
int[] arr = {2, 4, 1, 14, 14, 11, 3};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
[1, 2, 3, 4, 11, 14, 14]

Arrays.binarySearch(arr,key);
```

Notes

```
import java.util.ArrayList;
import java.util.Collections;
ArrayList<String> listOfCountries = new
                                ArrayList<String>();
listOfCountries.add("India");
listOfCountries.add("Canada");
listOfCountries.add("China");
listOfCountries.add("Denmark");
Collections.sort(listOfCountries);
listOfCountries.contains("China");
// returns true
```

Regression Testing

Save test cases
Use saved test cases in subsequent versions
A test suite is a set of tests for repeated testing
Cycling: Bugs have nasty habit of recycling. Those that are fixed in early versions, may reappear in later versions.
Regression testing: Repeating previous tests to ensure that known failures of prior versions do not appear in new versions

Organize test suite

1. Produce multiple tester classes, where each runs with a separate set of test data (more about this later).
2. A better way: Provide a generic tester, and feed it input from multiple files.

Example

```
import java.util.Scanner;
public class ParFilledArrayTester{
  public static void main (String[] args)  {
   double x;
   final int SIZE=5;
    Scanner myInput = new Scanner(System.in);
   ParFilledArray sample = new ParFilledArray(SIZE);
   while(true){
      x = myInput.nextDouble();
      if(!sample.addElement(x)){
         System.out.println(x+" rejected. Arra is full");
         break;
      }
   }
   System.out.println("Sum: "+sample.getSum());
  }
}
```

Check ParFilledArray.zip file in

Example(continue)

The program test class ParFilledArray by getting double numbers from keyboard and adding them to class ParFilledArray. The code of the program provided in Examples in Extra Notes.

Example(continue)

Typing the values by hand everything running the program is tedious.
Solution: use redirection method:
Save the sample data in a text file pass them to program at run time using following command:

```
Java ParFilledArrayTester <testData.txt
```

The program executed, but no longer reads input from keyboard. Instead System.in object gets the input from the file testData.txt.

Summary

- An array collects a sequence of values of the same type.
- Use for each loop if you do not need the index values in the loop body.
- An ArrayList stores sequence of values whose size can change.
- The ArrayList class is a generic class: ArrayList<Type>
- To collect numbers in array list, you must use wrapper classes.

Suggested Exercises:

E7.11, E7.12,E7.16
P7.1, P7.5