Array Lists, Wrapper Classes, and Auto-Boxing

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

Retrieving Array List Elements

- Use `get` method
- Index starts at 0
-

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

- Bounds error if index is out of range

Retrieving Array List Elements

- Most common bounds error:

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

Adding Elements

- `set` overwrites an existing value

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

- `add` adds a new value before the index

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

Adding Elements

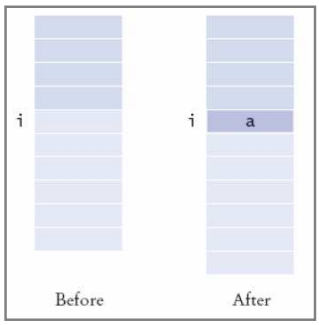

Figure 3:
Adding an Element in the Middle of an Array List

Removing Elements

- `remove` removes an element at an index

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

Removing Elements

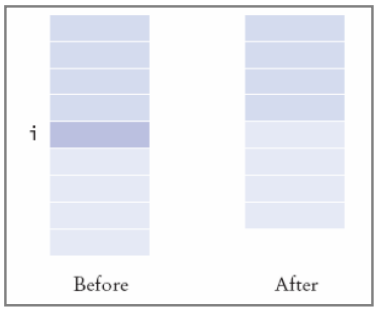

Figure 4:
Removing an Element in the Middle of an Array List

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
```

Question

- 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

Wrappers

- You cannot insert primitive types directly into array lists
- Cannot do:

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

- To treat primitive type values as objects, you must use wrapper classes:

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

Wrappers

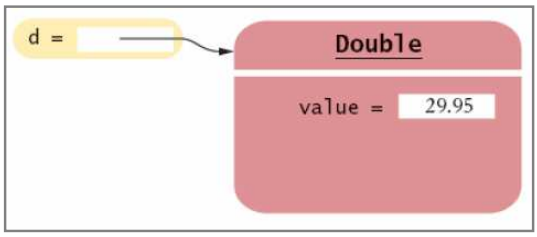

Figure 5:
An Object of a Wrapper Class

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();
```

Auto-boxing

- Auto-boxing even works inside arithmetic expressions

```
Double e = 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 e

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 Generalized `for` Loop

- Traverses all elements of a collection:

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

The Generalized `for` Loop

- Traditional alternative:

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

The Generalized `for` Loop

- Works for `ArrayLists` too:

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

The Generalized `for` Loop

- 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();
}
```

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

- Check all elements and count the matches until you reach the end of the array list.

```
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.

```
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 candidate with remaining elements
- Update it if you find a larger or smaller value

Simple Array Algorithms: Finding the Maximum or Minimum

- Example:

```
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;
```

Simple Array Algorithms: Finding the Maximum or Minimum

- 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);
. . .
```

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)
// Found a match
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