CHAPTER 15

MORE STANDARD JAVA TYPES

The two preceding chapters have introduced standard Java classes. This chapter continues with the same theme. As we have now studied the basic features of Java, there are only the huge number of standard classes and other types that are left to be learned. Fortunately, you do not have know them all. You can study them gradually, as is necessary. I hope that after this book has introduced some of the standard classes, you will be able to learn the other standard classes from the electronic Java documentation and other sources.

`ArrayList` is a standard class that is useful in applications in which dynamic arrays are needed. A dynamic array is such that its length (or size) is not fixed. Instead, the array length may increase or decrease while the array is used in a program. Class **`ArrayList`**, the first subject of this chapter, provides methods which automatically increase or decrease the length of an array in operations that insert or remove new objects to/from an array.

Interfaces, and especially standard interfaces like **`Comparable`**, is the second subject of this chapter. Interfaces specify methods. A class can implement an interface, which means that the class is equipped with certain methods.

The third subject of this chapter is the standard class **`GregorianCalendar`**, the official Java type for handling information related to dates and time. **`GregorianCalendar`** is a subclass of a class named **`Calendar`**. You will be shown how class **`GregorianCalendar`** can be used instead of the **`Date`** class which was introduced in Chapter 11.

15.1 ArrayList class

We have learned a long time ago that an array of type **int[]** is a data structure that can contain a certain number of data items of type **int**. Similarly, an array whose type is, say, **President[]** is an array that can contain a certain number of references to objects of type **President**. These traditional Java arrays are created with statements like

```
int[]        array_of_integers  =  new  int[ 50 ] ;
President[]  president_table    =  new  President[ 80 ] ;
```

and individual array elements can be accessed by giving an index expression inside brackets:

```
array_of_integers[ integer_index ]  =  77 ;
president_table[ 0 ]  =  new  President( ... ) ;
```

When a traditional array is created, the length of the array is specified by the value that is given in brackets. The length of an array, which can be read from the array field **length**, is the number of array elements. For example, the above **array_of_integers** has 50 array elements to store values of type **int**, and its length is thus 50.

A shortcoming of a traditional array is that its length cannot be changed after the array has been created. A traditional array is efficient, which means that not much computing time is required to read and write the array elements, but in some programs the fixed length of an array causes problems, or at least makes programming difficult. For this reason, Java provides a standard array class named **ArrayList**. **ArrayList**-based arrays are less efficient than the traditional Java arrays, but in some programs they are very useful. The most important feature of **ArrayList**-based arrays is that they can grow dynamically, i.e., the memory space of an array is increased automatically if necessary.

ArrayList arrays can store only (references to) objects. When you create an **ArrayList** array, you should specify what kinds of objects you intend to store with the array. An **ArrayList**-based array can be created with a statement like

```
ArrayList<Integer>  array_of_integers  =
                                  new  ArrayList<Integer>() ;
```

By writing **<Integer>** after the class name you can specify that the array will store objects of type **Integer**. There is no need to specify a length or a capacity for the array because the array can grow automatically when necessary. **ArrayList** is a so-called *generic class* whose characteristics can be fine-tuned at the moment when an object of the class is created. By writing a class name inside angle brackets, < >, after the name **ArrayList**, we can stipulate what kinds of objects the array will store. The class name inside the angle brackets is a *type parameter* for the generic class. In general, an **ArrayList** array like

```
ArrayList<SomeClass>  some_array  =
                               new  ArrayList<SomeClass>() ;
```

can store objects of **SomeClass** or objects of some subclass of **SomeClass**.

The simplest way to add elements to an **ArrayList**-based array is to use the method **add()** which can add a new element to the end of an array. For example, the statements

```
array_of_integers.add( 123 ) ;
array_of_integers.add( 456 ) ;
array_of_integers.add( 789 ) ;
```

would add three elements of type **int** to the array of type **ArrayList<Integer>** that is created above. When these statements are executed right after the creation of the array, the number of array elements becomes 3. The integer values that are added to the array by the above statements are values, not objects. Therefore, automatic boxing operations happen when the above statements are executed. In a boxing operation, a value type is automati-

cally converted to an object. The above **int** values are converted to **Integer** objects. As a result of the execution of the above statements, the first three positions in the array reference "boxes" that contain the integer values.

An **ArrayList** array has a specific capacity in regard to the amount of elements it can hold. This capacity is consumed gradually when new array elements are added to the array. When the capacity to add new elements has been exhausted, it is enlarged automatically. The capacity of an array is enlarged so that a new and larger memory area is allocated for the array, the old array elements are copied to the new memory area, and the new memory area becomes the official internal memory area of the array.

Traditional arrays have the **length** field which can be read when we want to know how many elements the array contains. Because **ArrayList** arrays have, at least in theory, an unlimited capacity to store data, **ArrayList** arrays do not have the **length** field. Instead, the **ArrayList** class provides the **size()** method which returns a value that tells how many elements an array currently contains.

ArrayList-based arrays cannot be indexed with index expressions inside square brackets. To access individual array elements, the **ArrayList** class provides methods named **get()** and **set()**. With these methods an element in certain index position can be read or written.

We get a particularly interesting **ArrayList**-based array with the statement

```
ArrayList<Object>  miscellaneous_objects  =
                                  new  ArrayList<Object>() ;
```

This statement creates an array that can store objects of the standard class **Object** and objects of all subclasses of the **Object** class. Because **Object** is the superclass of all Java classes, the array created by the above statement can store all kinds of objects. All the following statements would thus be acceptable:

```
miscellaneous_objects.add( 555 ) ;
miscellaneous_objects.add( 66.77 ) ;
miscellaneous_objects.add( "This is a string literal." ) ;
miscellaneous_objects.add( new  Date( "03.02.2004" ) ) ;
```

These statements add objects of types **Integer**, **Double**, **String**, and **Date** to the same array, and each element of the array points to a different type of object.

It is easy to add different kinds of objects to the above array, but when we want to do something with the objects, the situation can be somewhat more complicated. For example, if we want to take a substring that contains the first three characters of the **String** object that is the third one added to the above array, we might first write

```
miscellaneous_objects.get( 2 ).substring( 0, 3 )
```

The compiler would not, however, accept this because the method call **get(2)** returns a reference to type **Object**, and class **Object** does not have a method named **substring()**. In order to make the above method call acceptable to the compiler, we would have to convert the array element to type **String** in the following way

```
((String) miscellaneous_objects.get( 2 )).substring( 0, 3 )
```

Things must not, fortunately, always be this difficult. For example, all the objects that are stored by the above array could be printed with the loop

```
for ( int object_index  =  0 ;
          object_index  <  miscellaneous_objects.size() ;
          object_index  ++ )
{
   System.out.print( "\n "
             +  miscellaneous_objects.get( object_index ) ) ;
}
```

In the case of the above loop, there are no problems with compilation because for all objects there is a method named **`toString()`**, and that method will be called automatically when the string concatenation operator (+) is applied to an object in the array. Inside the above loop, **`miscellaneous_objects.get( object_index )`** returns an element of an **`ArrayList`**-based array, and that element is an object reference of type **`Object`**. When the string concatenation operator (+) is used to concatenate something to **`"\n"`**, the **`toString()`** method is invoked for the referenced object. As **`toString()`** is a polymorphic method that is declared in class **`Object`** and redefined in its subclasses, the method is executed so that first the type of the referenced object is checked, and then the **`toString()`** method of the class of the referenced object is invoked.

On the following pages you can find example programs **ArrayListDemo.java**, **MorseCodes.java**, and **Translate.java** that demonstrate the use of **`ArrayList`**-based arrays and **`ArrayList`** methods. Program **Findreplace.java** in the previous chapter also uses class **`ArrayList`**. In the following section there is a program named **Events.java**, but because that program introduces a new concept called interface, it is presented in a new section. Briefly described, the following are the most important **`ArrayList`** methods:

- To add elements to an **`ArrayList`** array you can use methods **`add()`** and **`addAll()`**. When these methods are called, the number of elements in the **`ArrayList`** array is increased. The more powerful of these methods is **`addAll()`** because with that method you can insert all the elements of another **`ArrayList`** array or some other collection to any position of an **`ArrayList`** array. The operation of method **`addAll()`** is described in Figure 15-1. Method **`add()`** can insert only a single element to a specified position of an **`ArrayList`** array. There are two versions of both methods. One that adds elements only to the end of an array, and one that is capable of inserting elements in the middle of an array.
- To remove or delete elements from an **`ArrayList`** array, you can use methods **`remove()`** and **`clear()`**. When these methods are called, the number of array elements is reduced. There are two versions of method **`remove()`**. One version removes the first occurrence of a certain object, while the other removes the element of a certain array position. Method **`clear()`** removes all elements from an **`ArrayList`** array.
- To make copies of an **`ArrayList`** array, you can use methods **`clone()`** and **`toArray()`**. Method **`clone()`** creates a shallow copy of an **`ArrayList`** array. Method **`toArray()`** converts the whole **`ArrayList`** array to a conventional array.
- To search for a certain element from an **`ArrayList`** array, you can use methods **`indexOf()`**, **`lastIndexOf()`**, and **`contains()`**. Method **`indexOf()`** can search for an object in an **`ArrayList`** array and return its index. The value -1 is returned if the object is not found. Method **`lastIndexOf()`** works like **`indexOf()`** but it starts the search from the last position of an array and proceeds towards the beginning of the array. Method **`contains()`** is a boolean method that can be used to check whether a certain object is in an **`ArrayList`** array. Method **`contains()`** returns **`false`** in the same situation when method **`indexOf()`** returns -1.

By using the electronic Java documentation you can find more accurate information about the methods of class **`ArrayList`**. It is normal that sometimes it is hard to understand how a method of a class works. In such a situation, one possibility is to write a test program in which the method is used. You could, for example, use **ArrayListDemo.java** as a test program by adding new method calls to it.

In addition to the actual **`ArrayList`** methods, you can use the static methods of class **`Collections`** when you work with **`ArrayList`** arrays. Some of the **`Collections`** methods are used in the programs of this section. In the **javafilesextra** folder, there is the program **CollectionsMethods.java** which demonstrates some of the **`Collections`** methods.

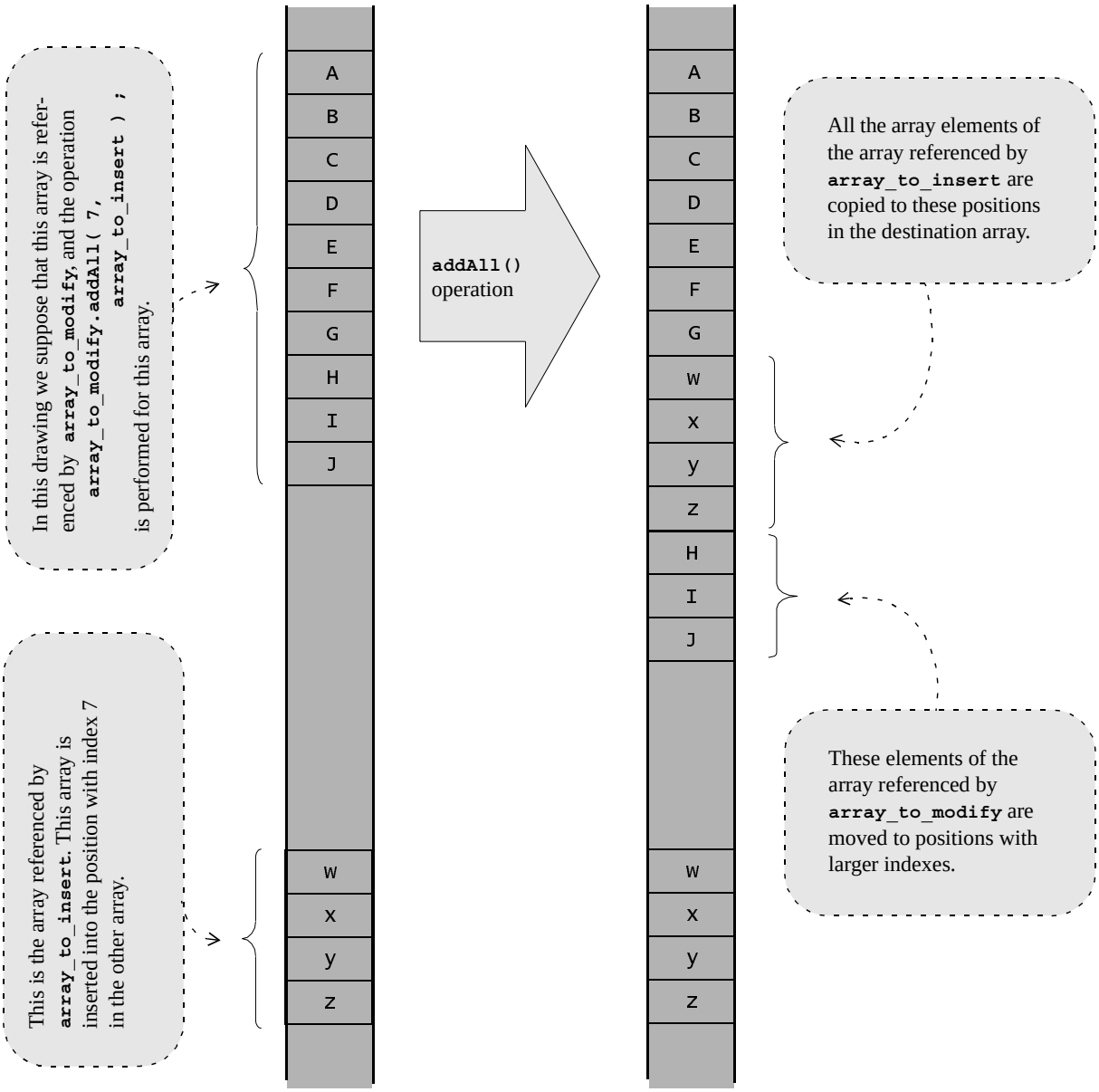

Figure 15-1. Performing an addAll() operation with ArrayList arrays.

"Old-fashioned" ArrayList arrays

In older Java versions the **`ArrayList`** class was not a generic class. This means that **`ArrayList`**-based arrays were declared without specifying the type of the objects that were intended to be stored to the array. To be compatible with older Java versions, the latest Java versions still accept **`ArrayList`** arrays that are created without a type parameter. For example, the array referenced by **`miscellaneous_objects`**, that is discussed in this section, can be declared and created with the statement

```
ArrayList  miscellaneous_objects  =  new  ArrayList() ;
```

This array works in the same way as an array that is created with the statement

```
ArrayList<Object>  miscellaneous_objects  =  new  ArrayList<Object>() ;
```

but an **`ArrayList`** array that is declared without a type parameter is less reliable in some situations, and, therefore, such arrays should not be used. Also the compiler prints warning messages when such arrays are declared.

```
D:\javafiles3>java ArrayListDemo

 Value 404 has index:  3

   202  101  505  404

   202  101  888  999  505  404

   202  101  888  999  404

   202  101  999  404

   777  666

   202  101  999  404  777  666

   202  101  999  411  777  666
```

Here, method **`remove()`** has removed the array element in position with index 4. Another version of **`remove()`** has been called to remove value 888 from the array.

One version of the **`addAll()`** method adds the elements of another array to the end of an **`ArrayList`** array. Here it has added the 2 elements from the array referenced by **`another_array`**.

ArrayListDemo.java - X. Experiments with ArrayList-based arrays.

```
//  ArrayListDemo.java

import java.util.* ;

class ArrayListDemo
{
   static void print_array( ArrayList<Integer> given_array )
   {
      System.out.print( "\n\n " ) ;

      for ( int element_index  =  0 ;
                element_index  <  given_array.size() ;
                element_index  ++ )
      {
         System.out.printf( "%5s", given_array.get( element_index ) ) ;
      }
   }
```

The **`size()`** method knows how many elements an **`ArrayList`** array currently contains.

Method **`print_array()`** prints the array versions that are created in method **`main()`**. When the array elements are printed with the format specifier **`%5s`**, they appear right-justified on the screen. The **`s`** in the format specifier refers to the word "string". When the elements are printed as strings, the **`toString()`** method is automatically invoked for the objects.

An element of an **`ArrayList`** array can be read with the **`get()`** method which takes an index as a parameter.

ArrayListDemo.java - 1: Demonstrating the standard class ArrayList.

Here, the standard class **ArrayList** is used to create an array. As the type parameter is **<Integer>**, the array will store **Integer** objects. It is not necessary to specify any length or size for the array. Initially, this array does not contain any elements, but the array grows automatically when objects are added to it.

Method **add()** pushes an object to the end of the array. An **add()** operation increases the number of array elements by one. The **int** values that are given to the **add()** method are automatically boxed inside **Integer** objects.

```
    public static void main( String[] not_in_use )
    {
        ArrayList<Integer> array_of_integers  =  new  ArrayList<Integer>() ;

        array_of_integers.add( 202 ) ;
        array_of_integers.add( 101 ) ;
        array_of_integers.add( 505 ) ;
        array_of_integers.add( 404 ) ;

        System.out.print( "\n Value 404 has index:  "
                    +  array_of_integers.indexOf( 404 )  ) ;
        print_array( array_of_integers ) ;

        array_of_integers.add( 2, 999 ) ;
        array_of_integers.add( 2, 888 ) ;
        print_array( array_of_integers ) ;

        array_of_integers.remove( 4 ) ;
        print_array( array_of_integers ) ;

        array_of_integers.remove( new Integer( 888 ) ) ;
        print_array( array_of_integers ) ;

        ArrayList<Integer> another_array  =   new  ArrayList<Integer>() ;
        another_array.add( 777 ) ;
        another_array.add( 666 ) ;
        print_array( another_array ) ;

        array_of_integers.addAll( another_array ) ;
        print_array( array_of_integers ) ;

        array_of_integers.set( 3, array_of_integers.get( 3 )  +  7 ) ;
        print_array( array_of_integers ) ;
    }
}
```

With method **get()** it is possible to read an array element in certain array position, and the **set()** method writes a new value to a specified array position. As the index values start from zero, 3 refers to the fourth array element. This statement thus adds 7 to the fourth element.

Another version of the **add()** method is used to insert new array elements to the array position with index 2. As a result of the insertion operation, the elements in positions with indexes 2, 3, 4, ... are moved to positions with indexes 3, 4, 5, ..., respectively.

ArrayListDemo.java - 2. Using the methods of class ArrayList to modify arrays.

Iterators

Many standard classes in the Java class library provide the possibility to use so-called iterators. For example, class **ArrayList** has inherited a method named **iterator()** that returns an iterator that can be used to read the objects that are stored by an **ArrayList**-based array. The returned iterator implements the standard interface **Iterator**. Iterators can be used instead of index variables when **ArrayList** objects are processed. For example, the loop inside method **print_array()** of program **ArrayListDemo.java** could be rewritten by using an iterator in the following way:

```
Iterator  element_to_print  =  given_array.iterator() ;

while ( element_to_print.hasNext() == true )
{
   System.out.printf( "%5s", element_to_print.next() ) ;
}
```

An iterator is a kind of pointer or a special reference to the objects of an array. In the above loop, the iterator **element_to_print** points to the objects of an **ArrayList**-based array. You can think that after the creation of the iterator, it points to a position that is one position behind the first object in the array. When the **Iterator** method **next()** is called, the iterator is advanced to the next object in the array, and a reference to the object is returned. The first call to **next()** makes the iterator point to the first object of the array. With the **Iterator** method **hasNext()** it is possible to check whether the array has more elements, i.e., whether a call to the **next()** method will be successful.

By comparing the above loop to the corresponding loop in program **ArrayListDemo.java**, you can see that using an iterator can simplify loops, or at least make a loop shorter. Instead of iterators, however, it is better to use "foreach" loops which are shorter than traditional loops. The above program lines can be replaced, for example, with the following "foreach" loop:

```
for ( Object element_in_array : given_array )
{
   System.out.printf( "%5s",  element_in_array ) ;
}
```

Invented by Samuel Morse in the U.S. in 1844, Morse codes were the first widely-used method for transmitting textual information. Each letter of the alphabet is coded with a sequence of signals. A signal can be either short or long. If two communicating parties know the Morse codes, they can communicate, for example, with a flashlight. To transmit letter L, for example, you first show the light for a short time, then once for a longer time, and finally you show it twice for a shorter time. Before telephones and computers became popular, Morse codes were widely used to send textual messages through electric lines and radio waves. Although these codes have less importance these days, they remain an important invention in the history of information processing. (The "code" for the space character is my invention in this program.)

```
D:\javafiles3>java MorseCodes

  Type in your name: Kari Laitinen

  Your name in Morse codes is:

   -.-   .-   .-.   ..          .-..   .-   ..   -   ..   -.   .   -.
```

MorseCodes.java - X. The string "Kari Laitinen" written with Morse codes.

With the static method **addAll()** of class **Collections**, all elements of a conventional Java array are added to the end of an empty **ArrayList** array. The array referenced by **array_of_morse_codes** is an array of strings, and, after this statement has been executed, the **ArrayList** array is an **ArrayList** version of the conventional array. Both arrays contain characters and their Morse codes as strings. The Morse code of a character is always in the following array position.

```
//  MorseCodes.java

import java.util.* ;

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

      String[]  array_of_morse_codes  =

       { "A", ".-",   "B", "-...", "C", "-.-.", "D", "-..", "E", ".",
         "F", "..-.", "G", " --.", "H", "....", "I", "..",  "J", ".---",
         "K", "-.-",  "L", ".-..", "M", "--",   "N", "-.",  "O", "---",
         "P", ".--.", "Q", "--.-", "R", ".-.",  "S", "...", "T", "-",
         "U", "..-",  "V", "...-", "W", ".--",  "X", "-..-","Y", " -.--",
         "Z", "--..", "1", ".----","2", "..---","3", "...--","4","....-",
         "5", ".....","6", "-....","7", "--...","8", "---..","9","----.",
         "0", "-----"," ", "     "  } ;

      ArrayList<String> arraylist_of_morse_codes = new ArrayList<String>() ;

      Collections.addAll( arraylist_of_morse_codes, array_of_morse_codes ) ;

      System.out.print( "\n  Type in your name: " ) ;

      String  given_name  =  keyboard.nextLine().toUpperCase() ;

      System.out.print( "\n  Your name in Morse codes is: \n\n" ) ;

      for ( int character_index  =  0 ;
                character_index  <  given_name.length() ;
                character_index  ++ )
      {
         int index_of_character_in_arraylist  =
                arraylist_of_morse_codes.indexOf(
                       ""  +  given_name.charAt( character_index ) ) ;

         if ( index_of_character_in_arraylist  !=  -1 )
         {
            System.out.print( "   "  +
                      arraylist_of_morse_codes.get(
                                index_of_character_in_arraylist + 1 ) ) ;
         }
      }
   }
}
```

Because the Morse code of a character always follows the character in the array, we add one to the index in order to get the Morse code.

MorseCodes.java - 1. Using class ArrayList to store String objects.

This program can make translations between words of two or three natural languages. **BilingualTranslation** is a class that is used to translate words between two natural languages like English and Spanish. Objects of class **BilingualTranslation** contain a pair of words which translate to each other.

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

import java.util.ArrayList ;

class  BilingualTranslation
{
   protected String first_word ;
   protected String second_word ;

   public BilingualTranslation() {}

   public BilingualTranslation( String  given_first_word,
                                String  given_second_word )
   {
      first_word   =  given_first_word ;
      second_word  =  given_second_word ;
   }

   public boolean translate( String given_word )
   {
      boolean  translation_was_successful  =  false ;

      if ( given_word.equals( first_word ) )
      {
         System.out.print( "\n \"" + given_word + "\" translates to \""
                         + second_word  + "\"" ) ;

         translation_was_successful  =  true ;
      }

      if ( given_word.equals( second_word ) )
      {
         System.out.print( "\n \"" + given_word + "\" translates to \""
                         + first_word  +  "\"" ) ;

         translation_was_successful  =  true ;
      }

      return  translation_was_successful ;
   }
}
```

Here, there must be a default constructor (i.e. a constructor which can be called without giving any parameters) because another class is derived from this class. The default constructor of this class is executed before the constructor of the derived class.

This method returns **true** if it can translate the given word. Translation is possible if the given word is the same as some of the words inside the object itself. A line of text is printed only if translation is possible.

Method **translate()** is a polymorphic method of which there exist several versions in this class hierarchy. In the derived class **TrilingualTranslation** there is a different version of this method.

Translate.java - 1: The declaration of class BilingualTranslation.

Being an enhanced version of its superclass, class **TrilingualTranslation** works with three natural words. The words are supplied to the constructor when a translation object is created.

```
class  TrilingualTranslation  extends  BilingualTranslation
{
   protected String third_word ;

   public TrilingualTranslation( String given_first_word,
                                 String given_second_word,
                                 String given_third_word )
   {
      first_word   =  given_first_word ;
      second_word  =  given_second_word ;
      third_word   =  given_third_word ;
   }

   public boolean translate( String given_word )
   {
      boolean  translation_was_successful  =  false ;

      if ( given_word.equals( first_word ) )
      {
         System.out.print( "\n \"" + given_word + "\" translates to \""
                   + second_word  + "\" and \"" + third_word + "\"" ) ;

         translation_was_successful  =  true ;
      }

      if ( given_word.equals( second_word ) )
      {
         System.out.print( "\n \"" + given_word + "\" translates to \""
                   + first_word  + "\" and \"" + third_word  + "\"" ) ;

         translation_was_successful  =  true ;
      }

      if ( given_word.equals( third_word ) )
      {
         System.out.print( "\n \"" + given_word + "\" translates to \""
                   + first_word  + "\" and \"" + second_word  + "\"" ) ;

         translation_was_successful  =  true ;
      }

      return  translation_was_successful ;
   }
}
```

Because this method works with three natural languages, it prints longer lines of text than the corresponding method in class **BilingualTranslation**. Note that if you want to include double quote characters inside a string literal, you must use a backslash \ before the double quote character.

Translate.java - 2: Class TrilingualTranslation and its version of method translate().

Here, translation objects are created to the heap memory, and references to the objects are added to the end of **array_of_translations**.

An **ArrayList**-based array is used to store references to translation objects. When an **ArrayList** array is created this way, the array can store references to objects that are either objects of class **BilingualTranslation** or objects of some subclass of **BilingualTranslation**. As **TrilingualTranslation** is a subclass of **BilingualTranslation**, **TrilingualTranslation** objects can be stored as well.

```
class Translate
{
   public static void main( String[] command_line_parameters )
   {
      ArrayList<BilingualTranslation>  array_of_translations  =
                           new  ArrayList<BilingualTranslation>() ;

      array_of_translations.add(
             new BilingualTranslation( "week", "semana" ) ) ;
      array_of_translations.add(
             new TrilingualTranslation( "street", "calle", "rue" ) ) ;
      array_of_translations.add(
             new BilingualTranslation( "eat", "comer" ) ) ;
      array_of_translations.add(
             new TrilingualTranslation( "woman", "mujer", "femme" ) ) ;
      array_of_translations.add(
             new TrilingualTranslation( "man", "hombre", "homme" ) ) ;
      array_of_translations.add(
             new BilingualTranslation( "sleep", "dormir" ) ) ;

      if ( command_line_parameters.length  ==  1 )
      {
         int  translation_index  =  0 ;

         while ( translation_index  <  array_of_translations.size() )
         {
            array_of_translations.get( translation_index ).
                              translate( command_line_parameters[ 0 ] ) ;

            translation_index  ++  ;
         }

         System.out.print( "\n" ) ;
      }
      else
      {
         System.out.print( "\n Give a word on command line.\n\n" ) ;
      }
   }
}
```

Method **translate()** is called here to possibly produce a translation. Depending on what type of object is referenced by the array element of **array_of_translations**, the appropriate version of the two versions of method **translate()** is selected automatically.

Translate.java - 3. A simple translation application which uses the translation classes.

```
D:\javafiles3>java Translate week

 "week" translates to "semana"

D:\javafiles3>java Translate woman

 "woman" translates to "mujer" and "femme"

D:\javafiles3>java Translate rue

 "rue" translates to "street" and "calle"
```

In this case the given word was found as the last word inside an object of type **`TrilingualTranslation`**.

Translate.java - X. The program is executed three times here.

Exercises with program Translate.java

Exercise 15-1. Modify program **Translate.java** so that it informs the user if it is not able to translate the given word. Method **`translate()`** returns **`true`** or **`false`** depending on whether or not the translation was successful, but in the current version of the program that information is ignored. (It is possible to call a non-**`void`** method in the same way as methods of type **`void`** are called.) You could declare a variable like

```
boolean  word_has_been_translated  =  false ;
```

in method **`main()`** and set that to value **`true`** when a translation has been made.

Exercise 15-2. Improve program **Translate.java** so that it is capable of translating between four different languages. You can derive a class named **`FourLanguageTranslation`** from class **`TrilingualTranslation`**.

Exercise 15-3. Make program **Translate.java** to read its translation data from a file. You could convert the program to a translator application with which it would be possible to add new word combinations to the translation data. The program could be a menu-based application similar to program **Collect.java**. If you can find, for example from the Internet, an existing file which contains translations of words from one language to another, you could make a translation program which uses those existing translations. To make this task simpler, it might be best to translate only between two languages.

Exercises related to ArrayList-based arrays

Exercise 15-4. In Chapter 7, a program named **Reverse.java** is introduced. That program uses a conventional Java array to store values of type **`int`**. Rewrite the program so that **`int`** values are stored to an **`ArrayList`** array instead of a conventional array.

Exercise 15-5. An **`ArrayList`**-based array is particularly useful in an application in which objects are dynamically inserted to an array and removed from an array. Program **Collect.java** is this kind of application. Rewrite program **Collect.java** so that you use inside class **`Collection`** an **`ArrayList`** array in place of the conventional array. This modification should simplify the methods of class **`Collection`**. (If you have developed some other program that is similar to **Collect.java**, it might be a useful idea to use an **`ArrayList`**-based array in that program.)

15.2 Comparable and other interfaces

A Java class can inherit the members of only one immediate superclass. A class cannot have several immediate superclasses. Sometimes, however, it is necessary that classes can be specified so that they possess certain qualities in addition to the qualities that are inherited from the immediate superclass and classes that are superclasses of the immediate superclass. The concept of *interface* has been invented to specify additional qualities of classes.

When a class is declared, it can inherit one class, and in addition it can *implement* one or more interfaces. Java has the reserved keyword **`implements`** which can be used in the following way

```
class SomeClass  extends  SomeSuperclass
                implements SomeInterface, SomeOtherInterface
{
   ...
```

In this case **`SomeClass`** implements two named interfaces **`SomeInterface`** and **`SomeOtherInterface`**. When a class implements several interfaces, the names of the interfaces are separated by commas in the class declaration.

An interface usually specifies a set of methods. In addition an interface can specify constants. An interface contains only method declarators (method headers). It does not provide implementations (i.e. method bodies) for the specified methods. It is the responsibility of the class that implements an interface to provide implementations for the specified methods.

To explore the nature of interfaces, let's suppose that the interface **`SomeInterface`**, which is implemented by **`SomeClass`** above, is declared in the following way:

```
interface SomeInterface
{
   int calculate_something( int given_value ) ;
   void do_something() ;
}
```

When **`SomeClass`** implements this interface, it means that **`SomeClass`** must have a method named **`calculate_something()`** which takes an **`int`** value as a parameter and returns an **`int`** value, and it must have a method named **`do_something()`** that neither takes parameters nor returns anything. As the above **`SomeClass`** also implements the other interface **`SomeOtherInterface`**, it means that **`SomeClass`** contains also all the methods specified by that interface.

Interface declarations clearly resemble class declarations. The keyword **`interface`** is used in place of the **`class`** keyword. Like classes, interfaces are usually written to their own source program files, and the file name must correspond to the name of the interface. An interface named **`SomeInterface`** should be kept in a file named **SomeInterface.java**.

Java provides standard interfaces in addition to standard classes. Many of the standard interfaces are *generic interfaces*, which means that when a class implements an interface, it is necessary to specify the type of objects with which the methods of the interface will operate. The class **`Event`** in program **Events.java** implements the standard interface **`Comparable`** in the following way

```
class  Event  extends Date
              implements Comparable<Event>
{
   ...
```

This class declaration means that class **`Event`** has a method named **`compareTo()`** which can compare **`Event`** objects. By writing **`<Event>`** after the interface name we specify that the **`compareTo()`** method, that is the only method required by the **`Comparable`** interface,