CHAPTER 10

CLASSES AND OBJECTS

Although the keyword **`class`** has been used in every example program of this book, only now, after having studied the fundamentals of programming, we can really start exploring the nature of Java classes. Class is a very fundamental concept in modern computer programming. In the programs that we have studied in the previous chapters, classes have played a minor role. Because it is mandatory that all methods of a Java program are written inside some class, we always have had a class declaration in our programs. From now on, however, we'll start using more advanced classes in our programs. You will learn that classes can be your own (data) types with which you can create the kind of objects you want.

As classes are used to create objects, programming based on classes is called object-oriented programming. Therefore, this chapter is the beginning of Part III "Object-Oriented Programming" in this book. In this first object-oriented chapter, we'll examine simple classes that can be used to declare and create objects. In further chapters, the concept of a class will be elaborated. It is not possible to explain the concept of a class with a few words. Therefore, as the concept becomes clearer in further chapters, you might consider the rest of this book as a long answer to the question: "What is a class?"

10.1 Classes, fields, and instance methods

The classes that we have studied in the previous chapters have been just program structures that contain a set of static methods, often only the single static **`main()`** method. Now we are, finally, going to study classes that are the "real" classes that can be used to create objects, that represent (data) types specified by a programmer. Designing these kinds of classes can be called object-oriented programming, and, therefore, this chapter starts the object-oriented Part III of this book.

We need classes in order to create objects. We have already, in Chapter 8, studied objects of type **`String`**. **`String`** is a standard class of Java, and, as that class is automatically available for Java programs, we can specify a string reference with a statement like

```
String  some_string ;
```

Then a **`String`** object can be created, for example, with a statement like

```
some_string  =  "xxxxxx" ;
```

The above statement creates a **`String`** object whose content is "xxxxxx", a character string in which letter x is repeated 6 times. The statement also makes **`some_string`** reference the created object. When the string reference **`some_string`** references a **`String`** object, it stores the address of the object in the heap memory.

In the case of **`String`** objects, the class to create objects already exists, and that makes programming quite easy. When you want to create some special kinds of objects, you first have to declare a class that specifies the nature of your objects. Programs **Rectangles.java** and **BankSimple.java** are examples that demonstrate simple classes and the creation of objects that are based on the simple classes. The structure of both programs is the following

```
class SomeClass
{
   Declarations of data items (fields).

   Declarations of instance methods.
}

class SomeTesterClass
{
   public static void main( String[] not_in_use )
   {
      SomeClass  some_object  =  new  SomeClass() ;

      ...
   }
}
```

The program files **Rectangles.java** and **BankSimple.java** both contain two classes. First there is a "real" class that can be used to create objects, and then there is another class that contains the **`main()`** method which creates the objects.

The classes that are used to create objects usually have data declarations which are called fields in programming terminology. The class **`Rectangle`** of program **Rectangles.java** begins with the following lines

```
class Rectangle
{
   int  rectangle_width ;
   int  rectangle_height ;
   char  filling_character ;
```

on which **rectangle_width**, **rectangle_height**, and **filling_character** are (data) fields that belong to every object of type **Rectangle**. These fields specify how a simple rectangle looks when it is printed onto the screen of a computer.

The methods of a class that is used to create objects are usually non-static methods that are said to be instance methods. They are instance methods because objects of a class are also called instances of a class, and these methods can only be called in relation to an instance. For example, the method **print_rectangle()** in **Rectangles.java** is called with the statement

```
first_rectangle.print_rectangle() ;
```

where **first_rectangle** is a reference to an object of type **Rectangle** (an instance of class **Rectangle**) and method **print_rectangle()** is called in relation to the object referenced by **first_rectangle**. When **print_rectangle()** is called this way, it prints the **Rectangle** object for which it was called, and it uses those data fields (**rectangle_width**, **rectangle_height**, and **filling_character**) that exist inside the object referenced by **first_rectangle**. (The verb "invoke" is also used when the calling of an instance method is discussed. To describe the activity of calling an instance method for an object, it is commons to say that "a method is invoked for an object".)

It is important to understand that every object of a class contains copies of all data fields of the class. Every object contains instances of the fields of its class. For example, when we create an object of class **Rectangle** in the following way

```
Rectangle  first_rectangle  =  new  Rectangle() ;
```

we actually create a data structure that contains all the data fields declared in class **Rectangle**. The object referenced by **first_rectangle** contains its own **rectangle_width**, **rectangle_height**, and **filling_character**. These internal data fields are accessed through the methods of class **Rectangle**. For example, when the statement

```
first_rectangle.initialize_rectangle( 7, 4, 'Z' ) ;
```

is executed, method **initialize_rectangle()** sets the values of the fields inside the object referenced by **first_rectangle** so that **rectangle_width** is given the value 7, **rectangle_height** is set to 4, and **filling_character** is set to contain the character code of uppercase letter Z. Method **initialize_rectangle()** does not know the name of the object reference when it is executed, but the dot operator . binds it to the correct object in the call. The above call to method **initialize_rectangle()** could be explained in a longer way as "Go and execute the statements inside method **initialize_rectangle()** using the data fields inside the object referenced by **first_rectangle**."

As you already have studied arrays and strings, which also are objects, it should not be very difficult to understand how objects are created to the heap memory, and how they are referenced with a reference in the stack memory. A statement like

```
Rectangle  first_rectangle ;
```

declares an object reference that can be used to reference (or to point to) an object, but this statement does not yet create any objects. An object can be created by using the **new** operator in a statement like

```
first_rectangle  =  new  Rectangle() ;
```

This statement creates a **Rectangle** object to the heap memory, and makes **first_rectangle** reference the created object. An object reference references an object so that it stores the physical memory address of the object. Figure 10-1 shows how the **Rectangle** objects of program **Rectangles.java** are referenced by the references **first_rectangle** and **second_rectangle**. Figure 10-1 describes the situation right before the method **main()** of program **Rectangles.java** terminates.

Class **Rectangle** differs from classes we have seen before so that data items are declared before the methods of the class. **rectangle_width**, **rectangle_height**, and **filling_character** are data items that belong to every object of type **Rectangle**. These data items are called fields in programming terminology. Fields are data members of a class.

```
//  Rectangles.java

class Rectangle
{
   int  rectangle_width ;
   int  rectangle_height ;
   char  filling_character ;

   public void initialize_rectangle( int  given_rectangle_width,
                                     int  given_rectangle_height,
                                     char given_filling_character )
   {
      rectangle_width    =  given_rectangle_width ;
      rectangle_height   =  given_rectangle_height ;
      filling_character  =  given_filling_character ;
   }

   public void print_rectangle()
   {
      for ( int number_of_rows_printed  =  0 ;
                number_of_rows_printed  <  rectangle_height ;
                number_of_rows_printed  ++ )
      {
         System.out.print( "\n      " ) ;

         for ( int number_of_characters_printed  =  0 ;
                   number_of_characters_printed  <  rectangle_width ;
                   number_of_characters_printed  ++ )
         {
            System.out.print( filling_character ) ;
         }
      }

      System.out.print( "\n" ) ;
   }
}
```

The methods of a class can freely read and write the data fields, the classwide data, that are declared at the beginning of the class.

Class **Rectangle** has two methods, **initialize_rectangle()** and **print_rectangle()**, which are written inside the class declaration in the same way as the methods that we have seen before. Because the keyword **static** is not used in the declaration of these methods, they are non-static instance methods than can only be called in relation to a **Rectangle** object according to the following statement syntax

```
object_reference_name.method_name( ... ) ;
```

Rectangles.java - 1: The declaration of class Rectangle.

Class **Rectangles** follows class **Rectangle** in file **Rectangles.java**. Class **Rectangles** exists only because it is logical to have a different class where the **main()** method may be placed. The **main()** method could alternatively be placed inside class **Rectangle**.

This statement declares a reference **first_rectangle** that can reference a **Rectangle** object, creates a **Rectangle** object, and makes **first_rectangle** reference the created object. This statement could be replaced with the statements

```
Rectangle  first_rectangle ;
first_rectangle  =  new  Rectangle() ;
```

```
class Rectangles
{
   public static void main( String[] not_in_use )
   {
      Rectangle  first_rectangle  =  new Rectangle() ;

      first_rectangle.initialize_rectangle( 7, 4, 'Z' ) ;
      first_rectangle.print_rectangle() ;

      Rectangle  second_rectangle  =  new Rectangle() ;

      second_rectangle.initialize_rectangle( 12, 3, 'X' ) ;
      second_rectangle.print_rectangle() ;
   }
}
```

After its creation, an object of type **Rectangle** contains the data fields **rectangle_width**, **rectangle_height**, and **filling_character**, but these fields contain only zeroes. Method **initialize_rectangle()** can be used to give meaningful values to these fields.

Rectangles.java - 2. The main() method of class Rectangles that creates two Rectangle objects.

As this chapter belongs to Part III of this book, the example programs are in a directory (folder) that has number 3 in its name.

```
D:\javafiles3>java Rectangles

      ZZZZZZZ
      ZZZZZZZ
      ZZZZZZZ
      ZZZZZZZ

      XXXXXXXXXXXX
      XXXXXXXXXXXX
      XXXXXXXXXXXX
```

This is the **Rectangle** object referenced by **second_rectangle**. The printed rectangle is 12 character positions wide, 3 rows high, and filled with character X.

Rectangles.java - X. The rectangles are made by printing a single character repeatedly.

The other simple program **BankSimple.java** shows how a simple bank account class can be declared and used. A banking program may be a useful example, because a large portion of the world's computing power is consumed making calculations related to money. Computers calculate, for example, wages, share prices, and maintain information about money stored in accounts in banks. While studying program **BankSimple.java**, you should bear in mind that real banking programs are much more complicated. The program could not be used in a real bank, but it does demonstrate some operations with objects.

The `BankAccount` objects created in program **BankSimple.java** are somewhat more complicated than the objects created in program **Rectangles.java**. The reason for this complication is that `BankAccount` objects have a string as a data field, and strings are objects themselves. The field `account_owner` of class `BankAccount` references a `String` object. When a `BankAccount` object is initialized with method `initialize_account()`, `account_owner` starts to reference a separate `String` object where the owner's name is stored. Figure 10-2 shows what the objects look like in the main memory of a computer when all the objects of program **BankSimple.java** have been created.

In order to design useful classes, we should learn to think in object-oriented way. In object-oriented thinking we should think first about data. After having thought what set of data fields could form an entity, an object, we should think what kinds of methods are needed to process that data. In the case of class `BankAccount` in program **BankSimple.java**, object-oriented thinking goes as follows:

- `BankAccount` objects are such that every object contains the name of the account owner (a string), the number of the account, and the balance of the account (i.e. how much money is currently stored in the account).
- As class `BankAccount` has three methods, there are three different possibilities to do something with `BankAccount` objects.
- By calling method `initialize_account()` for a `BankAccount` object, it is possible to initialize data fields `account_owner`, `account_number`, and `account_balance`.
- It is possible to increase the value of data field `account_balance` for a `BankAccount` object by calling method `deposit_money()`.
- By calling `show_account_data()` it is possible to see all data inside a `BankAccount` object.

A central idea in the design of classes is that data is encapsulated inside objects, and the data is accessed only through calls to methods. This principle is used both in program **Rectangles.java** and in **BankSimple.java**. Although the data fields of a class should be accessed only by the methods of the same class, it is possible to write programs in which data fields are accessed by the methods of a foreign class. For example, the data field `account_balance` of a `BankAccount` object can be accessed from method `main()`. The statement

```
first_account.account_balance  =
          first_account.account_balance  +  2222.11 ;
```

would be acceptable in the method `main()` of **BankSimple.java** in place of the method call

```
first_account.deposit_money(  2222.11 ) ;
```

If you want to prevent other classes from accessing the fields of a class, the fields should be declared with keyword `private`. On the other hand, fields declared with keyword `public` are automatically visible to all methods in all classes. Program **Person.java** provides an example of a class with `public` fields. Classes like the class `Person` in program **Person.java** are not, however, very object-oriented classes, and they should not be used too often. The accessibility of class members will be discussed more thoroughly on page 398.

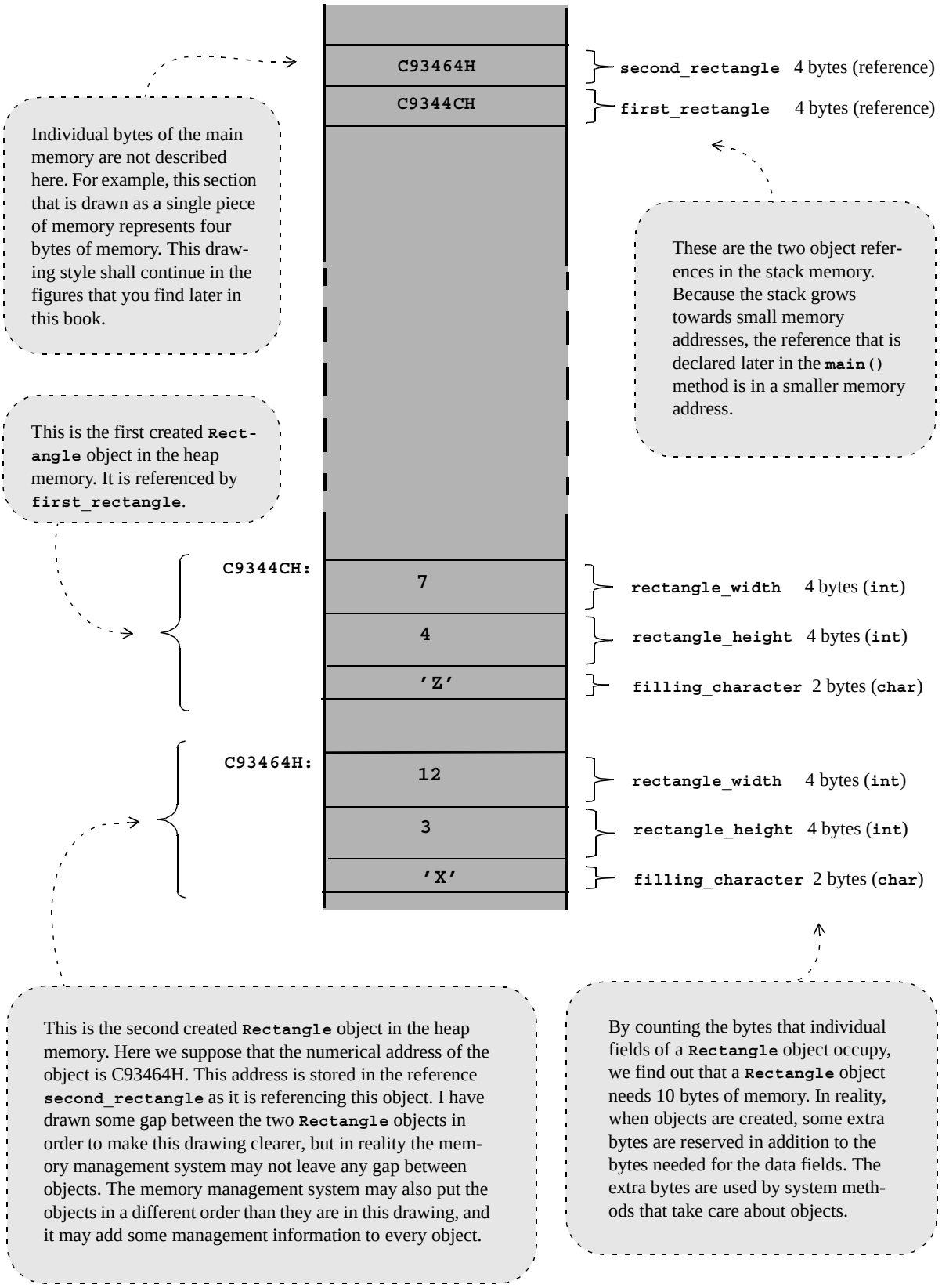


Figure 10-1. The objects of program Rectangles.java in the main memory.

These three data items, a string and two variables, are the data fields of class **BankAccount**. Every **BankAccount** object will have its own copy of these fields.

Method **initialize_account()** can be used to give initial values to the data fields **account_owner** and **account_number**. The field **account_balance** is set to zero. It is important to note that a method does not know for which object it was called. Method **initialize_account()** initializes the three fields, but it does not know which **BankAccount** object the fields belong to. Only the caller knows for which object it called the method.

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

class  BankAccount
{
   String  account_owner ;
   long    account_number ;
   double  account_balance ;

   public void initialize_account( String given_name,
                                   long   given_account_number )
   {
      account_owner    =  given_name  ;
      account_number   =  given_account_number ;
      account_balance  =  0  ;
   }

   public void show_account_data()
   {
      System.out.print( "\n\nB A N K   A C C O U N T   D A T A : "
                     +  "\n   Account owner :  "  +  account_owner
                     +  "\n   Account number:  "  +  account_number
                     +  "\n   Current balance: "  +  account_balance ) ;
   }

   public void deposit_money( double amount_to_deposit )
   {
      System.out.print( "\n\nTRANSACTION FOR ACCOUNT OF " +  account_owner
                     +  " (Account number " +  account_number  + ")" ) ;
      System.out.print( "\n   Amount deposited: "  +  amount_to_deposit
                    +  "\n   Old account balance: "  +  account_balance ) ;
      account_balance  =  account_balance  +  amount_to_deposit ;
      System.out.print( "   New balance: "  +  account_balance  ) ;
   }
}
```

The basic banking operations are mathematically simple. An addition operation must be carried out in order to make a deposit to an account. **amount_to_deposit** is given as a parameter for this method. Instance methods handle parameters in the same way as the static methods that we studied in the previous chapter.

BankSimple.java - 1: The declaration of class BankAccount.

```
class BankSimple
{
   public static void main( String[] not_in_use )
   {
      BankAccount  first_account   =  new  BankAccount() ;
      BankAccount  second_account  =  new  BankAccount() ;

      first_account.initialize_account(  "James Bond", 77007007 ) ;
      second_account.initialize_account( "Philip Marlowe", 22003004 ) ;

      first_account.deposit_money(  5566.77 ) ;
      second_account.deposit_money( 9988.77 ) ;
      first_account.deposit_money(  2222.11 ) ;

      first_account.show_account_data() ;
      second_account.show_account_data() ;
   }
}
```

first_account and **second_account** are object references that are made to reference the two **BankAccount** objects that are created to the heap memory. Both objects contain the three data fields **account_owner**, **account_number**, and **account_balance**. The fields inside the objects are modified by calling methods for the objects.

BankSimple.java - 2. Class BankSimple that contains the method main().

```
D:\javafiles3>java BankSimple


TRANSACTION FOR ACCOUNT OF James Bond (Account number 77007007)
   Amount deposited: 5566.77
   Old account balance: 0.0   New balance: 5566.77

TRANSACTION FOR ACCOUNT OF Philip Marlowe (Account number 22003004)
   Amount deposited: 9988.77
   Old account balance: 0.0   New balance: 9988.77

TRANSACTION FOR ACCOUNT OF James Bond (Account number 77007007)
   Amount deposited: 2222.11
   Old account balance: 5566.77   New balance: 7788.880000000001

B A N K   A C C O U N T   D A T A :
   Account owner :  James Bond
   Account number:  77007007
   Current balance: 7788.880000000001

B A N K   A C C O U N T   D A T A :
   Account owner :  Philip Marlowe
   Account number:  22003004
   Current balance: 9988.77
```

BankSimple.java - X. The program always produces the same output.

The fields of class **Person** are declared with keyword **public**, which makes these fields accessible for methods in all other classes. The keyword **public** is an access modifier. Other access modifiers include keywords **private** and **protected**. Also a missing access modifier affects the visibility of a field. See page 398 for more information related to the visibility of class members.

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

class  Person
{
   public  String  person_name ;
   public  int     year_of_birth ;
   public  String  country_of_origin ;

   public void print_person_data()
   {
      System.out.print( "\n   " + person_name  + " was born in "
                  +  country_of_origin  + " in " + year_of_birth ) ;
   }
}

class  PersonTest
{
   public static void main( String[] not_in_use )
   {
      Person  computing_pioneer  =  new  Person() ;

      computing_pioneer.person_name        =  "Alan Turing" ;
      computing_pioneer.year_of_birth      =  1912 ;
      computing_pioneer.country_of_origin  =  "England" ;

      Person  another_computing_pioneer  =  new  Person() ;

      another_computing_pioneer.person_name        =  "Konrad Zuse" ;
      another_computing_pioneer.year_of_birth      =  1910 ;
      another_computing_pioneer.country_of_origin  =  "Germany" ;

      computing_pioneer.print_person_data() ;
      another_computing_pioneer.print_person_data() ;
   }
}
```

The fields inside a **Person** object can be referred to by using the dot operator (.) which is also used when methods are called for objects.

Person.java - 1. A class that has public data fields.

```
D:\javafiles3>java PersonTest

   Alan Turing was born in England in 1912
   Konrad Zuse was born in Germany in 1910
```

The class name **PersonTest** is written on the command line because that class contains the **main()** method.

Person.java - X. These lines are printed by calling method print_person_data() twice.

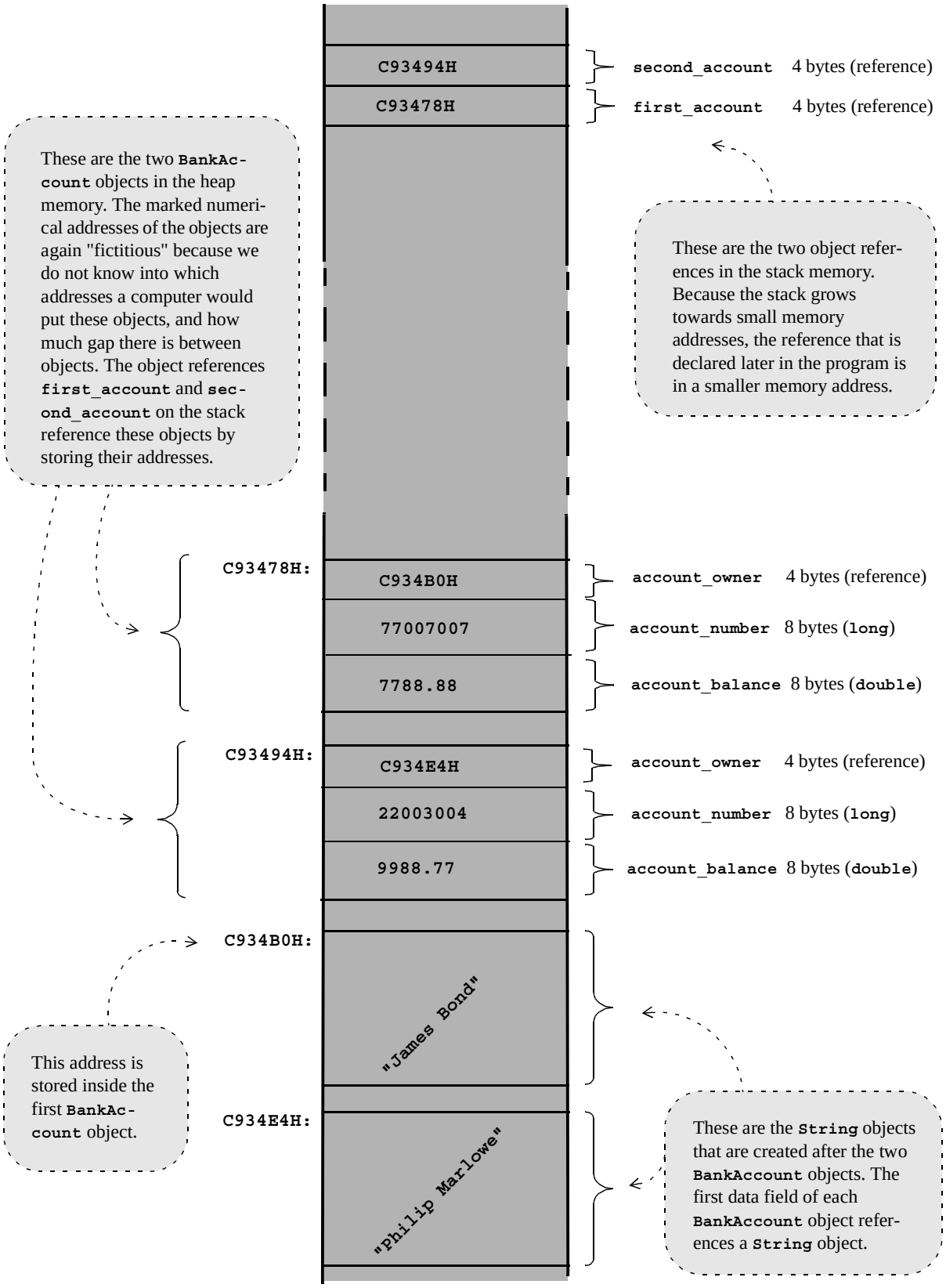


Figure 10-2. The objects of program BankSimple.java in the main memory.

10.2 Constructors are methods that build objects

Every class declaration introduces a new type, analogous to the basic built-in types **`char`**, **`int`**, **`double`**, etc. Once a class is declared, we can create "variables" based on the declared class. The "variables" based on class declarations are called objects because they are different from the traditional variables of type **`char`**, **`int`**, **`double`**, etc. Objects are more complex data items than conventional variables. In most cases, objects need to be initialized somehow when they are declared. We can say that objects need to be constructed before they are ready for use. In program **BankSimple.java**, there is a method to initialize objects. But because it is very common that objects have to be initialized, Java classes are usually equipped with special methods called constructors that can initialize objects. Initialization usually means that data fields are given certain initial values.

Program **BankBetter.java** has a **`BankAccount`** class that is equipped with a constructor. The **`BankAccount`** class in program **BankBetter.java** has the same data fields as the class in program **BankSimple.java**. The difference between these two programs is that the **`BankAccount`** class in program **BankBetter.java** has more methods, and it does not need the method **`initialize_account()`** because it has a constructor. Program **BankBetter.java** is an advanced version of our previous example since the methods of its **`BankAccount`** class allow money to be withdrawn from **`BankAccount`** objects, and money to be transferred between two **`BankAccount`** objects.

Constructors are like other methods of a class, and they are written according to the normal Java rules for methods. The following facts should be remembered about constructors:

- A constructor method must have the same name as the class where it is declared. A constructor of a class named **`ClassName`** is of the form

```
public ClassName( ... )
{
   ...
}
```

- Constructor methods may not have a type. They cannot even be of type **`void`**.
- The compiler generates a call to a constructor when it discovers that an object is being created in a program. For example, when the compiler finds a statement like

```
ClassName object_name  =  new  ClassName( "XXX", 222 ) ;
```

  it calls a constructor of class **`ClassName`**, and that class must have a constructor that takes a string and an integer value as parameters.
- As we shall soon see, a class can have several overloaded constructor methods which take different kinds of actual parameters.When an object is created, the compiler selects a constructor that has matching formal parameters.

Executing the internal statements of a constructor method is just one of the activities that happen when an object is created. The following is a longer list of activities in the process of the creation of an object

- Memory space is allocated from the heap memory for the object. The size of the reserved memory space depends on how much memory is needed by the data fields belonging to the object.
- A constructor is called. The constructor usually initializes the data fields, and takes care of other necessary initialization-related tasks. If the constructor does not initialize data fields, the fields are initialized by default with zeroes.
- The memory address of the object in the heap memory is returned and stored in an object reference in the stack memory. (In this book we can suppose that objects are referenced so that their addresses are stored in object references. This kind of logi-

cal thinking is correct from a programmer's point of view although the actual management of objects were more complicated. In reality, depending on how the used Java virtual machine and the automatic memory management system work, an object reference may store an indirect address to an object.)

You may already have wondered that how it is possible that the program **BankSimple.java** contains statements like

```
BankAccount  first_account  =  new  BankAccount() ;
```

where there is clearly a constructor call, but the class `BankAccount` of program **BankSimple.java** does not have a constructor. The explanation of this inconsistency is that if there is no constructors declared in a class, the compiler automatically generates a so-called default constructor that can be called without supplying any parameters. The compiler generates these constructors in programs **BankSimple.java** and **Rectangles.java** where the classes do not have any constructor methods. The compiler-generated default constructors do not actually do anything. They just fulfill the requirement that here has to be a constructor in every class. It is also important to note that the default constructors are not generated if there is a constructor in a class. Therefore, if you try to insert the above statement into the `main()` method of program **BankBetter.java**, it will not work. It works only in **BankSimple.java** where no constructors are present.

Constructors are needed to build objects, but usually no special methods are needed to destroy objects. When an object is created inside a method in the following way

```
public void some_method( ... )
{
   SomeClass  some_object  =  new  SomeClass( ... ) ;

   ...
}
```

the object resides in the heap memory and `some_object` references (or points to) the object. When `some_method()` terminates, all its local data including the object reference `some_object` ceases to exist. The memory space reserved for local data is released from the stack memory when a method terminates. Thus, when `some_method()` above reaches its end, `some_object` simply stops referencing the object in the heap memory, and the object becomes an unnecessary object that is not referenced any more. In such a situation, the object does not need to exist in the heap memory. Therefore, a separate memory management mechanism called the garbage collector sees to it that the object is removed from the heap memory and its memory space is freed for other purposes. The garbage collector is a background program that runs automatically together with Java programs and takes care of automatic memory management activities.

A method that has the same name as the class itself is a constructor method of the class. The compiler generates a call to a constructor when an object is created. Constructors are typeless methods. Not even the type **void** may be specified for them. This constructor simply copies the values of its parameters to the fields of the class.

```
//  BankBetter.java   (c) 2005 Kari Laitinen

class  BankAccount
{
   String  account_owner ;
   long    account_number ;
   double  account_balance ;

   public BankAccount( String  given_account_owner,
                       long    given_account_number,
                       double  initial_balance )
   {
      account_owner    =  given_account_owner ;
      account_number   =  given_account_number ;
      account_balance  =  initial_balance  ;
   }

   public void show_account_data()
   {
      System.out.print( "\n\nB A N K   A C C O U N T   D A T A : "
                     +  "\n   Account owner :  "  +  account_owner
                     +  "\n   Account number:  "  +  account_number
                     +  "\n   Current balance: "  +  account_balance ) ;
   }

   public void deposit_money( double amount_to_deposit )
   {
      System.out.print( "\n\nTRANSACTION FOR ACCOUNT OF " +  account_owner
                     +  " (Account number " +  account_number  + ")" ) ;
      System.out.print( "\n   Amount deposited: "  +  amount_to_deposit
                    +  "\n   Old account balance: "  +  account_balance ) ;
      account_balance  =  account_balance  +  amount_to_deposit ;
      System.out.print( "   New balance: "  +  account_balance  ) ;
   }
```

These two methods are the same as in program **BankSimple.java**.

BankBetter.java - 1: A program with a BankAccount class that has a constructor.

```
    public void withdraw_money( double amount_to_withdraw )
    {
       System.out.print( "\n\nTRANSACTION FOR ACCOUNT OF " +  account_owner
                     +  " (Account number " +  account_number  + ")" ) ;

       if ( account_balance  <  amount_to_withdraw )
       {
          System.out.print("\n   -- Transaction not completed: "
                   + "Not enough money to withdraw " + amount_to_withdraw ) ;
       }
       else
       {
          System.out.print("\n   Amount withdrawn:    "  +  amount_to_withdraw
                     + "\n   Old account balance: "  +  account_balance ) ;
          account_balance  =  account_balance  -  amount_to_withdraw ;
          System.out.print("   New balance: "  +  account_balance ) ;
       }
    }

    public void transfer_money_to( BankAccount  receiving_account,
                                   double       amount_to_transfer )
    {
       System.out.print( "\n\nTRANSACTION FOR ACCOUNT OF " +  account_owner
                     +  " (Account number " +  account_number  + ")" ) ;

       if ( account_balance  >=  amount_to_transfer )
       {
          receiving_account.account_balance  =
             receiving_account.account_balance  +  amount_to_transfer ;

          System.out.print(
                "\n   " + amount_to_transfer + " was transferred to "
              +  receiving_account.account_owner  +  " (Account no. "
              +  receiving_account.account_number +  ")."
              +  "\n   Balance before transfer: " +  account_balance ) ;
          account_balance  =  account_balance  -  amount_to_transfer ;
          System.out.print( "   New balance:  " +  account_balance ) ;
       }
       else
       {
          System.out.print( "\n   -- Not enough money for transfer." ) ;
       }
    }
}
```

There must be enough money for the withdrawal.

This statement transfers money from "this" account to a receiving account. **`receiving_account`** is a reference to a **`BankAccount`** object that is given as a parameter for this method. Because this is a method of class **`BankAccount`**, it is allowed to access the data fields of another **`BankAccount`** object by using the syntax

`object_reference_name.data_field_name`

BankBetter.java - 2: The other part of class BankAccount.

Here two **BankAccount** objects are created. When the Java compiler sees these **new** operations, it generates calls to the constructor method of class **BankAccount**, and passes the data given in parentheses as parameters to the constructor method.

An object reference is given as a parameter for method **transfer_money_to()**. Inside the method, **jazz_player_account** is referenced with reference **receiving_account**, and **moon_walker_account** is the "this" account, the account for which the method was called.

```
class BankBetter
{
   public static void main( String[] not_in_use )
   {
      BankAccount  jazz_player_account  =
                       new  BankAccount( "Louis Armstrong", 121212, 0 ) ;
      BankAccount  moon_walker_account  =
                       new  BankAccount( "Neil Armstrong",  191919,
                                                            7777.77 ) ;
      jazz_player_account.deposit_money( 3333.33 ) ;

      jazz_player_account.withdraw_money( 4444.44 ) ;

      moon_walker_account.transfer_money_to( jazz_player_account,
                                             2222.22 ) ;

      moon_walker_account.show_account_data() ;
      jazz_player_account.show_account_data() ;
   }
}
```

BankBetter.java - 3. Method main() that creates and uses two BankAccount objects.

Exercises with program BankBetter.java

Exercise 10-1. Write a new method **withdraw_all_money()** to class **BankAccount** in program **BankBetter.java**. The new method should take out all the money from a **BankAccount** object. It should also inform the user how much money was withdrawn. The following method calls could be written in method **main()** to test the new method

```
jazz_player_account.withdraw_all_money() ;
moon_walker_account.withdraw_all_money() ;
```

Exercise 10-2. Write a new method **transfer_money_from()** to class **BankAccount** in program **BankBetter.java**. The new method should transfer money from the other account to "this" account. It should move money in the opposite direction to the direction that the existing method **transfer_money_to()** moves. The new method could be called from method **main()** in the following way

```
jazz_player_account.transfer_money_from(
                              moon_walker_account, 333.33 ) ;
```

```
D:\javafiles3>java BankBetter


TRANSACTION FOR ACCOUNT OF Louis Armstrong (Account number 121212)
   Amount deposited: 3333.33
   Old account balance: 0.0   New balance: 3333.33

TRANSACTION FOR ACCOUNT OF Louis Armstrong (Account number 121212)
   -- Transaction not completed: Not enough money to withdraw 4444.44

TRANSACTION FOR ACCOUNT OF Neil Armstrong (Account number 191919)
   2222.22 was transferred to Louis Armstrong (Account no. 121212).
   Balance before transfer: 7777.77   New balance:  5555.550000000001

B A N K   A C C O U N T   D A T A :
   Account owner :  Neil Armstrong
   Account number:  191919
   Current balance: 5555.550000000001

B A N K   A C C O U N T   D A T A :
   Account owner :  Louis Armstrong
   Account number:  121212
   Current balance: 5555.549999999999
```

BankBetter.java - X. The output of the program is always the same.

Destructors do not exist in Java

Some other programming languages (e.g. C++) have classes that contain destructors in addition to constructors. Destructors are methods that are called when objects are destroyed. So, if you are familiar with C++, you might expect me to explain something about destructors. Unfortunately, or luckily, there is nothing to be explained because destructors do not belong to Java classes. As the automatic memory management system with the Garbage Collector automatically destroys objects which are no longer referenced, there is no need to have destructors in Java classes.

Because, in large and complicated programs, it is possible that something has to be done to objects before they are destroyed from the heap memory, Java provides a possibility to write a method that will be called automatically before an object is destroyed. The name of such a method must be **`finalize()`** and it is written like this

```
public void finalize()
{
   // Actions needed before the destruction of an object.

}
```

If you put this kind of method to a class, the method will be called automatically before the Garbage Collector destroys the object and deallocates the memory space of the object.

In the programs of this book, we are not going to use **`finalize()`** methods. If you need more information on this topic, please take a look at program **ObjectClassTests.java** in the **javafilesextra** folder.

10.3 Several constructors in a class

The overloading of method names was a subject discussed in Chapter 9. Overloading means that two or more methods may have the same name if their parameters differ sufficiently. The Java compiler can make a distinction between two methods with the same name if parameters have different types, or there is a different number of parameters. Let us, for example, suppose that we have two methods with the declarators

```
void print_numbers( int first_number, int second_number )
void print_numbers( int some_number )
```

If there was the method call

```
print_numbers( 77 ) ;
```

in some other method in the same class, the compiler would call the latter method above, because that takes a single parameter of type **int**.

Overloading is very common in the case of constructor methods. Classes often need to have several constructors because objects need to be constructed in different ways. As the constructor method must always have the same name as the class itself, constructor methods must be overloaded when several constructors are needed.

Program **Animals.java** contains a class declaration that has two constructors. The name of the class is **Animal**. **Animal** objects are quite fictitious, bearing little similarity to real animals. **Animal** objects are such that they can be fed and made to speak. When an **Animal** object is fed, it takes food into its stomach in the form of a string. When an **Animal** object is made to speak, it tells its species' name and what it has eaten. The data fields of class **Animal** are two strings that contain the name of the animal species and maintain information about stomach contents.

The declarators of the two constructors of class **Animal** are

```
public Animal( String  given_species_name )
public Animal( Animal  another_animal )
```

The compiler can distinguish these two methods having the same name since their parameters are of a different type. The first method takes a string reference as a parameter. The latter method takes a reference to type **Animal**. The first constructor initializes the **Animal** object with the given species name. The latter method makes a new copy of the other **Animal** object. It is possible to duplicate, or clone, **Animal** objects with the latter constructor. (Cloning real animals is much more difficult and dubious.)

Constructors that make copies of objects are called copy constructors. The latter constructor above is a copy constructor. It is very common, and sometimes even necessary, that classes are equipped with copy constructors. The copy constructor of a class takes a single parameter that is a reference to an object of the class itself. Thus, the copy constructor inside class **SomeClass** would look like

```
class SomeClass
{
   // declarations of data fields

   public SomeClass( SomeClass object_to_be_copied )
   {
      ...
   }
   // other constructors and methods
}
```

Another common constructor is the default constructor. Default constructors do not require any parameters. If the hypothetical class **SomeClass** above were equipped with the constructor

```
public SomeClass()
{
   ...
}
```

the class would have a default constructor. As was discussed earlier, the compiler automatically generates a default constructor if no constructors are declared in a class.

Exercises with program Animals.java

Exercise 10-3. Add the new data field

```
String  animal_name ;
```

to class **Animal** in program **Animals.java**. You have to modify the first constructor of the class so that an **Animal** object can be created by writing

```
Animal  named_cat  =  new  Animal( "cat", "Ludwig" ) ;
```

You also need to modify the copy constructor so that it copies the new data field. Method **make_speak()** must be modified so that it prints something like

```
Hello, I am a cat called Ludwig.
I have eaten: ...
```

Exercise 10-4. Modify method **make_speak()** in program **Animals.java** so that it prints something like

```
Hello, I am ...
My stomach is empty.
```

in the case when **stomach_contents** references just an empty string. The stomach is empty as long as method **feed()** has not been called for an **Animal** object. You can use the standard string method **length()** to check if the stomach is empty. Method **length()** can be used, for example, in the following way

```
if ( stomach_contents.length() == 0 )
{
   // stomach_contents references an empty string.
   ...
```

Exercise 10-5. Write a default constructor for class **Animal** in program **Animals.java**. A default constructor is such that it can be called without giving any parameters. The default constructor should initialize the data fields so that the program lines

```
Animal  some_animal  =  new  Animal();
some_animal.make_speak() ;
```

would produce the following output on the screen

```
Hello, I am a default animal called no name.
...
```

Exercise 10-6. Write a new method named **make_stomach_empty()** to class **Animal** in **Animals.java**. The new method could be called

```
animal_object.make_stomach_empty() ;
```

and it should make **stomach_contents** reference an empty string "".

The encapsulated data inside objects of class **Animal** consist of the name of the animal species, and of a stomach where food is put when an **Animal** object is fed.

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

class  Animal
{
   String  species_name ;
   String  stomach_contents ;

   public Animal( String  given_species_name )
   {
      species_name      =  given_species_name ;
      stomach_contents  =  "" ;
   }

   public Animal( Animal  another_animal )
   {
      species_name      =  another_animal.species_name ;
      stomach_contents  =  another_animal.stomach_contents ;
   }

   public void feed( String  food_for_this_animal )
   {
      stomach_contents  =
          stomach_contents  +  food_for_this_animal  +  ", "  ;
   }

   public void make_speak()
   {
      System.out.print( "\n Hello, I am a " + species_name     + "."
                     + "\n I have eaten: " + stomach_contents + "\n" ) ;
   }
}
```

Animal objects are fed by concatenating (appending) the food string to previous stomach contents. Operator + joins a new string to the end of an existing string. **stomach_contents** references a new **String** object after this operation.

The second constructor simply copies the fields of the object referenced by **another_animal**. As **another_animal** references an **Animal** object, it is possible to access the object's data fields with the dot operator. Note that the name of "this" object, the object for which the constructor was called, is not visible inside methods. The names **species_name** and **stomach_contents** automatically refer to the data fields of "this" object.

To be accurate, this copy constructor does not make a deep copy of the object referenced by **another_animal**. After this constructor has done its job, both "this" object and the object referenced by **another_animal** reference the same **String** objects that represent the stomach contents and species name. However, when "this" object is fed later with method **feed()**, the **feed()** method makes **stomach_contents** reference a new **String** object.

Animals.java - 1: Class Animal with two constructors and two other methods.

The first constructor of class **Animal** is called when these statements create objects. The compiler finds out that a string literal is given as a parameter, and that type of parameter is accepted by the first constructor. That constructor initializes the stomachs of the **Animal** objects with an empty string.

```
class  Animals
{
   public static void main( String[] not_in_use )
   {
      Animal  cat_object  =  new Animal( "cat" ) ;
      Animal  dog_object  =  new Animal( "vegetarian dog" ) ;

      cat_object.feed( "fish" ) ;
      cat_object.feed( "chicken" ) ;

      dog_object.feed( "salad" ) ;
      dog_object.feed( "potatoes" ) ;

      Animal  another_cat  =  new Animal( cat_object ) ;

      another_cat.feed( "milk" ) ;

      cat_object.make_speak() ;
      dog_object.make_speak() ;
      another_cat.make_speak() ;
   }
}
```

When **another_cat** is made to speak here, it is no longer an identical copy of **cat_object** because it was fed with milk after the cloning operation.

This object creation invokes the second constructor of class **Animal**. The object referenced by **another_cat** becomes a shallow copy of the object referenced by **cat_object**. The copy operation is shallow because the **String** objects that are referenced by the fields **species_name** and **stomach_contents** are not duplicated.

Animals.java - 2. Class Animals whose method main() creates and uses Animal objects.

```
D:\javafiles3>java Animals

 Hello, I am a cat.
 I have eaten: fish, chicken,

 Hello, I am a vegetarian dog.
 I have eaten: salad, potatoes,

 Hello, I am a cat.
 I have eaten: fish, chicken, milk,
```

Animals.java - X. All these lines are generated through calls to method make_speak().

10.4 Arrays containing references to objects

An array is a data structure where many data items of the same type can be stored. We have already studied arrays of the basic types **char**, **int**, **double**, etc. For example, we get an array whose type is **int[]** when we first declare an array reference like

```
int[]  array_of_integers ;
```

and then create an array with the **new** operator in the following way

```
array_of_integers  =  new  int[ 50 ] ;
```

It is common to combine the array declaration and creation operations into a single statement like

```
int[]  array_of_integers  =  new  int[ 50 ] ;
```

By putting a pair of empty brackets [] after the type name, we tell the compiler that we want to declare an array.

We can also create arrays that are based on the classes that we have declared. It is possible to declare and create arrays of type **Rectangle**, arrays of type **BankAccount**, arrays of type **Animal**, and so on. These arrays can store objects. An array that is based on a class type can be declared and created in the same way as the array above. An array reference named **array_of_objects** can be specified with a statement like

```
SomeClass[]  array_of_objects ;
```

and the actual array is created with a statement like

```
array_of_objects  =  new  SomeClass[ 50 ] ;
```

Also these two statements can be replaced with the single statement

```
SomeClass[]  array_of_objects  =  new  SomeClass[ 50 ] ;
```

The statement above creates an array whose type is **SomeClass[]**, and 50 objects of type **SomeClass** can be referenced by the array elements. What is important to understand is that the above array creation operation does not create any objects of type **SomeClass**. Right after its creation, the array above is a data structure that does not contain any references to objects. In the Java terminology, we say that such an array contain **null** references. **null** in a reserved keyword that means "no object referenced". At the machine level, when a program is executing, the array elements that contain a **null** are set to zero, but at the source program level, we speak about **null**.

To make the above hypothetical array reference objects, one possibility is to create an object for each array element in the following way

```
array_of_objects[ 0 ]  =  new  SomeClass() ;
array_of_objects[ 1 ]  =  new  SomeClass() ;
array_of_objects[ 2 ]  =  new  SomeClass() ;
array_of_objects[ 3 ]  =  new  SomeClass() ;
...
```

As you can see, arrays that contain references to objects can be indexed in the same way as arrays of the basic types. The index value for the first array element is zero, and the largest possible index value is the length of the array minus one.

Program **Olympics.java** is an example that uses an array that contains references to **Olympics** objects. The name of the array reference is **olympics_table**. This name was chosen because these kinds of arrays resemble tables that we can find in books and magazines. If **olympics_table** were a table in a book, it could begin in the following way

Olympic year	Olympic city	Olympic country
1896	Athens	Greece
1900	Paris	France
1904	St. Louis	United States
...	...	...

The names of the columns in the above book-style table are the same as the field names in class **Olympics**, and each row corresponds to an **Olympics** object in the array referenced by **olympics_table**. When you work with arrays like the one in **Olympics.java**, it may be helpful to imagine a book-style table in your mind.

Figure 10-3 shows how the array referenced by **olympics_table** looks like in the main memory of a computer. Because two fields of class **Olympics**, **olympic_city** and **olympic_country**, are string references, each **Olympics** object references two **String** objects. Each **Olympics** object, in turn, is referenced by an array element in the array that is referenced by **olympics_table**.

A method of a class can be called (invoked) for an object referenced by an array element with the call syntax

array_of_objects[*index expression* **].method_name(...) ;**

The dot operator . can thus be used also in the case of array references. The value of the index expression determines which array element is selected. An array that contains references to objects can be indexed in the same way as the arrays we have studied before. To clarify the indexing mechanism, let's study some examples supposing that we have the **olympics_table** of program **Olympics.java** available:

- The method call

  ```
  olympics_table[ 5 ].print_olympics_data() ;
  ```

  would print the data of year 1912 Olympics in Stockholm, Sweden.

- The method call

  ```
  olympics_table[ 2 ].get_year()
  ```

  would return value 1904.

- The **if** construct

  ```
  if ( olympics_table[ olympics_index + 1 ].get_year() == 9999)
  {
     ...
  ```

  would test if the **olympic_year** of the object referenced by the next array position is 9999.

- If the value of **olympics_index** is 9, the method call

  ```
  olympics_table[ olympics_index - 1 ].print_olympics_data() ;
  ```

  would print the data of year 1928 Olympics in Amsterdam, The Netherlands.

- The method call

  ```
  olympics_table[ 31 ].get_year()
  ```

  would generate a **NullPointerException** because there is a **null** in the 32th position in **olympics_table**. The **null** means that no object is referenced from that array position.

When an array is used in a program, the array is usually filled starting from the beginning of the array. Then, while processing the data in an array, it is usually necessary to test if the end of meaningful data of an array has been encountered. In program **Olympics.java**, the end of meaningful data is marked with a special `Olympics` object whose `olympic_year` is 9999. The array referenced by `olympics_table` has thus the following structure

- The array positions with indexes from 0 to 27 contain references to "real" `Olympics` objects.
- The array position with index value 28 references a "surreal" `Olympics` object whose purpose is to mark the end of the data.
- The array positions with indexes from 29 to 39 contain `null` values (zeroes) which were automatically written to these positions when the array was created. Because no objects were created for these positions, the `null` values remained. Also the other array positions were originally set to `null`, but these `null` values were overwritten when the `Olympics` objects were created.

Marking the end of meaningful data with special values is one possible way to make an array store data. Programs **Convert.java** and **Planets.java**, which you can find after **Olympics.java**, are examples that demonstrate two other ways for marking the end of meaningful data. In program **Convert.java**, the array referenced by `conversion_table` is such that it does not contain any `null` references. The array in **Convert.java** is thus full of meaningful data, and the meaningful data ends when the array ends. The end of the array in **Convert.java** is detected by using the data field `length` that belongs to every array in Java. The array in **Planets.java** is like the array of **Olympics.java** in that both arrays contain `null` references in those positions that come after the meaningful data. In **Planets.java**, the end of meaningful data is detected when the first `null` value is encountered in the array.

Arrays in Java are such that their length, the value stored by the `length` field, cannot be altered. The length of an array is the number of array elements. The array length is fixed when an array is created. For example, the statement

```
SomeClass[] array_name  =  new  SomeClass[ some_integer ] ;
```

creates an array whose length is the same as the value of variable `some_integer` at the moment when the array is created. If the value of `some_integer` is increased later, the length of the array referenced by `array_name` does not change. If the length of an array must be increased in a program, one possible way to solve the problem is to create a new longer array, copy all elements from the old array to the new array, and finally make the original array name reference the new array.

Exercises related arrays containing object references

Exercise 10-7. Modify program **Olympics.java** so that you remove the "surreal" `Olympics` object whose `olympic_year` is 9999 from the array referenced by `olympics_table`. The end of olympics data in the modified program should be detected in the same way as in program **Planets.java**, i.e., the first `null` reference in the array marks the end of meaningful data.

Exercise 10-8. By using program **Olympics.java** as an example, write a program that gives information about your favorite sports. For example, if you are interested in football, soccer, basketball, or ice-hockey, you can write a program that can inform which team was the champion in a given year. If your interest is car racing, you can write a program that knows which driver and which team were the champions in a given year.

We have learned that the double slash // is a mechanism for writing comments in Java programs. The pairs of characters /* and */ provide another possibility to write comments. The character pair /* marks the beginning of a comment. When the Java compiler sees the character pair /*, it discards all subsequent characters until it encounters the character pair */ which marks the end of the comment. The character pairs /* and */ are useful when we want to write long comments which occupy several lines of text.

```
/*  Olympics.java  Copyright (c) Kari Laitinen

    This program demonstrates the use of an array of
    objects, or, more precisely, an array that contains
    references to objects. The program first introduces
    a class named Olympics. An Olympics object can contain
    the data of olympic games. By using the class Olympics,
    an array named olympics_table is defined inside
    the main() method of class named OlympicsDataFinder.
    olympics_table is used to search data of olympic games.
*/

import java.util.* ;

class  Olympics
{
   int     olympic_year  ;
   String  olympic_city ;
   String  olympic_country ;

   public Olympics( int    given_olympic_year,
                    String given_olympic_city,
                    String given_olympic_country )
   {
      olympic_year    =  given_olympic_year ;
      olympic_city    =  given_olympic_city ;
      olympic_country =  given_olympic_country ;
   }

   public int get_year()
   {
      return  olympic_year ;
   }

   public void print_olympics_data()
   {
      System.out.print( "\n    In "  +  olympic_year  +
              ", Olympic Games were held in " +  olympic_city  +
              ", "  +  olympic_country  +  ".\n" ) ;
   }
}
```

The constructor of class **Olympics** copies its parameters to the corresponding fields of the class.

get_year() is a so-called accessor method with which it is possible to read one field of an object.

Olympics.java - 1: The declaration of class Olympics.

This statement both declares and creates an array that contains 40 references to **Olympics** objects. Immediately after the execution of this statement, the 40 references contain a **null** which means that they do not yet reference an object. This statement means the same as the two separate statements

```
Olympics[]  olympics_table ;
olympics_table  =  new  Olympics[ 40 ] ;
```

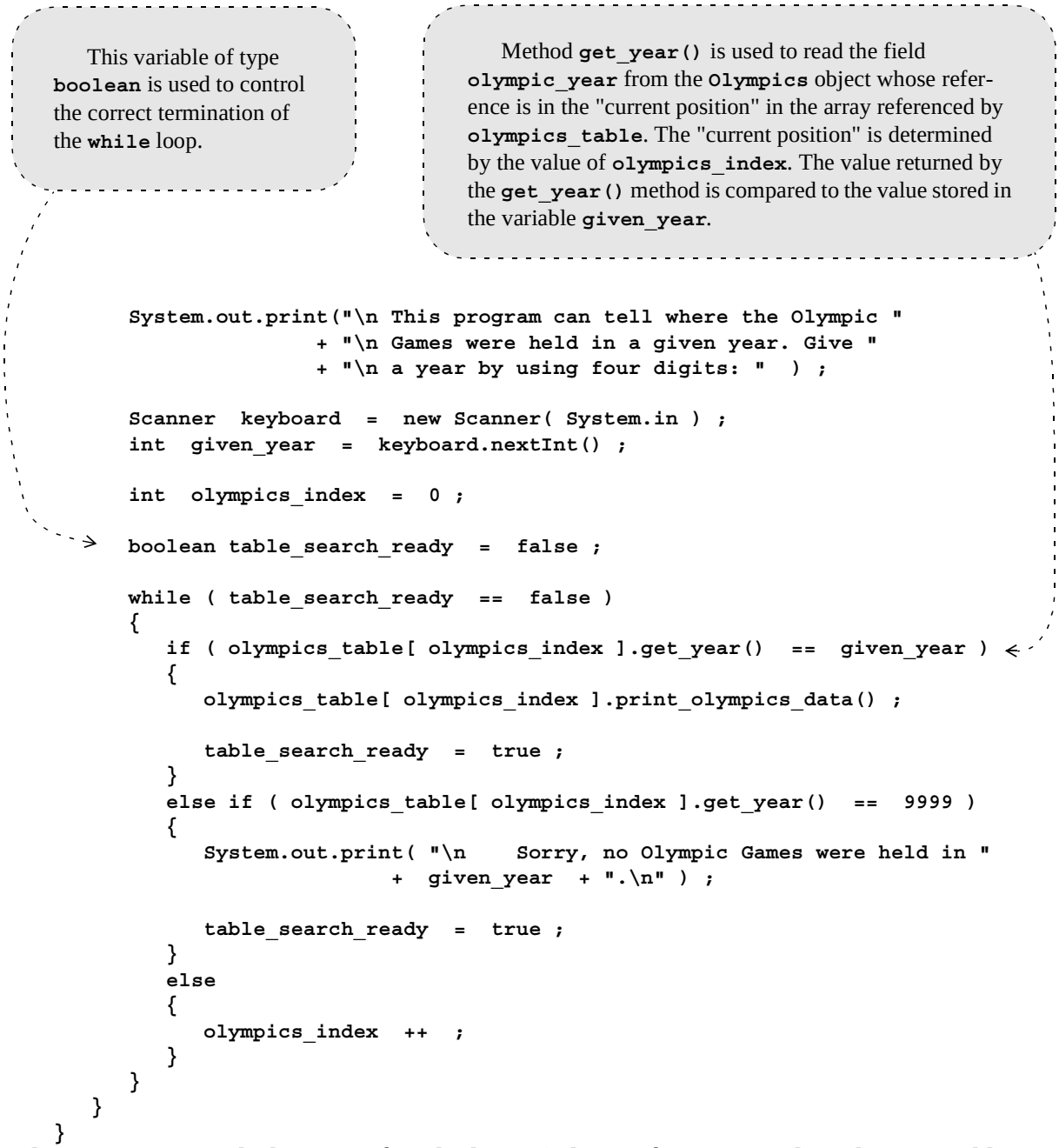
This and the rest of the statements on this page create 29 **Olympics** objects, and the references to the created objects are stored into the **olympics_table** positions with indexes from 0 to 28.

```
class  OlympicsDataFinder
{
   public static void main( String[] not_in_use )
   {
      Olympics[]  olympics_table  =  new Olympics[ 40 ] ;

      olympics_table[ 0 ] = new Olympics( 1896, "Athens",   "Greece" ) ;
      olympics_table[ 1 ] = new Olympics( 1900, "Paris",    "France" ) ;
      olympics_table[ 2 ] = new Olympics( 1904, "St. Louis", "U.S.A." );
      olympics_table[ 3 ] = new Olympics( 1906, "Athens",   "Greece"  ) ;
      olympics_table[ 4 ] = new Olympics( 1908, "London",   "Great Britain");
      olympics_table[ 5 ] = new Olympics( 1912, "Stockholm","Sweden" ) ;
      olympics_table[ 6 ] = new Olympics( 1920, "Antwerp",  "Belgium"   ) ;
      olympics_table[ 7 ] = new Olympics( 1924, "Paris",    "France"    ) ;
      olympics_table[ 8 ] = new Olympics( 1928, "Amsterdam","Netherlands");
      olympics_table[ 9 ] = new Olympics( 1932, "Los Angeles", "U.S.A.");
      olympics_table[ 10 ] = new Olympics( 1936, "Berlin",  "Germany"   ) ;
      olympics_table[ 11 ] = new Olympics( 1948, "London",  "Great Britain");
      olympics_table[ 12 ] = new Olympics( 1952, "Helsinki","Finland"  ) ;
      olympics_table[ 13 ] = new Olympics( 1956, "Melbourne","Australia" ) ;
      olympics_table[ 14 ] = new Olympics( 1960, "Rome",     "Italy"   ) ;
      olympics_table[ 15 ] = new Olympics( 1964, "Tokyo",    "Japan"   ) ;
      olympics_table[ 16 ] = new Olympics( 1968, "Mexico City","Mexico" ) ;
      olympics_table[ 17 ] = new Olympics( 1972, "Munich",   "West Germany");
      olympics_table[ 18 ] = new Olympics( 1976, "Montreal", "Canada"  ) ;
      olympics_table[ 19 ] = new Olympics( 1980, "Moscow",   "Soviet Union");
      olympics_table[ 20 ] = new Olympics( 1984, "Los Angeles","U.S.A.");
      olympics_table[ 21 ] = new Olympics( 1988, "Seoul",    "South Korea");
      olympics_table[ 22 ] = new Olympics( 1992, "Barcelona","Spain"   ) ;
      olympics_table[ 23 ] = new Olympics( 1996, "Atlanta",  "U.S.A." );
      olympics_table[ 24 ] = new Olympics( 2000, "Sydney",   "Australia" ) ;
      olympics_table[ 25 ] = new Olympics( 2004, "Athens",   "Greece"  ) ;
      olympics_table[ 26 ] = new Olympics( 2008, "Beijing",  "China"   ) ;
      olympics_table[ 27 ] = new Olympics( 2012, "London",   "Great Britain");
      olympics_table[ 28 ] = new Olympics( 9999, "end of",   "data"  ) ;
```

This **Olympics** object is used to mark the end of real olympics data.

Olympics.java - 2: olympics_table at the beginning of method main().




```
        System.out.print("\n This program can tell where the Olympic "
                       + "\n Games were held in a given year. Give "
                       + "\n a year by using four digits: "  ) ;

        Scanner  keyboard  =  new Scanner( System.in ) ;
        int  given_year  =  keyboard.nextInt() ;

        int  olympics_index  =  0 ;

        boolean table_search_ready  =  false ;

        while ( table_search_ready  ==  false )
        {
           if ( olympics_table[ olympics_index ].get_year()  ==  given_year )
           {
              olympics_table[ olympics_index ].print_olympics_data() ;

              table_search_ready  =  true ;
           }
           else if ( olympics_table[ olympics_index ].get_year()  ==  9999 )
           {
              System.out.print( "\n    Sorry, no Olympic Games were held in "
                             +  given_year  + ".\n" ) ;

              table_search_ready  =  true ;
           }
           else
           {
              olympics_index  ++  ;
           }
        }
     }
}
```

Olympics.java - 3. The last part of method main() that performs a search in olympics_table.

```
D:\javafiles3>java OlympicsDataFinder

 This program can tell where the Olympic
 Games were held in a given year. Give
 a year by using four digits: 1976

    In 1976, Olympic Games were held in Montreal, Canada.
```

Olympics.java - X. Here the search for olympics data was successful.

Initializing data fields with initializers

Constructors are the usual means to build objects and initialize their data fields. Another possibility to initialize fields is to assign initial values when the fields are introduced in a class declaration. It is thus possible to declare a class in the following way

```
class  SomeClass
{
   int     some_integer_field  =  9 ;
   double  some_number         =  33.44 ;
   String  some_string_field   =  "initial text" ;
   int[]   some_integer_array  =  { 66, 77, 88 } ;
   ...
```

The initial values that are given to the fields of a class this way are called initializers. The values that the fields receive through initializers take effect before the constructors are executed. The fields are initialized in the order in which they are written in the class declaration.

Initializers are useful when a class has several constructors, and certain fields must be given certain initial values in every constructor. In such a situation the constructors are simplified if initial values are given by using initializers.

The order of class members in a class declaration

Class members include data fields, constants, constructors, and methods. Constants are immutable data fields declared with the `final` keyword. The classes in the example programs of this book are written so that different kinds of class members are introduced in a certain order. The order is such that data fields are introduced before constructors and methods. The classes of this book thus have the structure

```
class  ClassName
{
   declarations of data fields and constants

   constructors

   accessor methods

   other methods
}
```

The Java compiler does not, however, require that class members are introduced in the above order. It is possible, for example, to declare a class so that data fields are introduced at the end of the class declaration. The class `Animal` of program **Animals.java** could thus alternatively be written in the following way

```
class  Animal
{
   public void feed( String  food_for_this_animal )
   { ...

   public void make_speak()
   { ...

   public Animal( String  given_species_name )
   { ...

   public Animal( Animal  another_animal )
   { ...

   String  species_name ;
   String  stomach_contents ;
}
```

Although the Java compiler does not set any strict rules for the order of class members in a class declaration, it is a good programming practice to always use a certain order of class members. The order of class members that is used in this book can be considered a logical order because the same order is used in UML class diagrams. (UML diagrams will be discussed in the following chapter.)

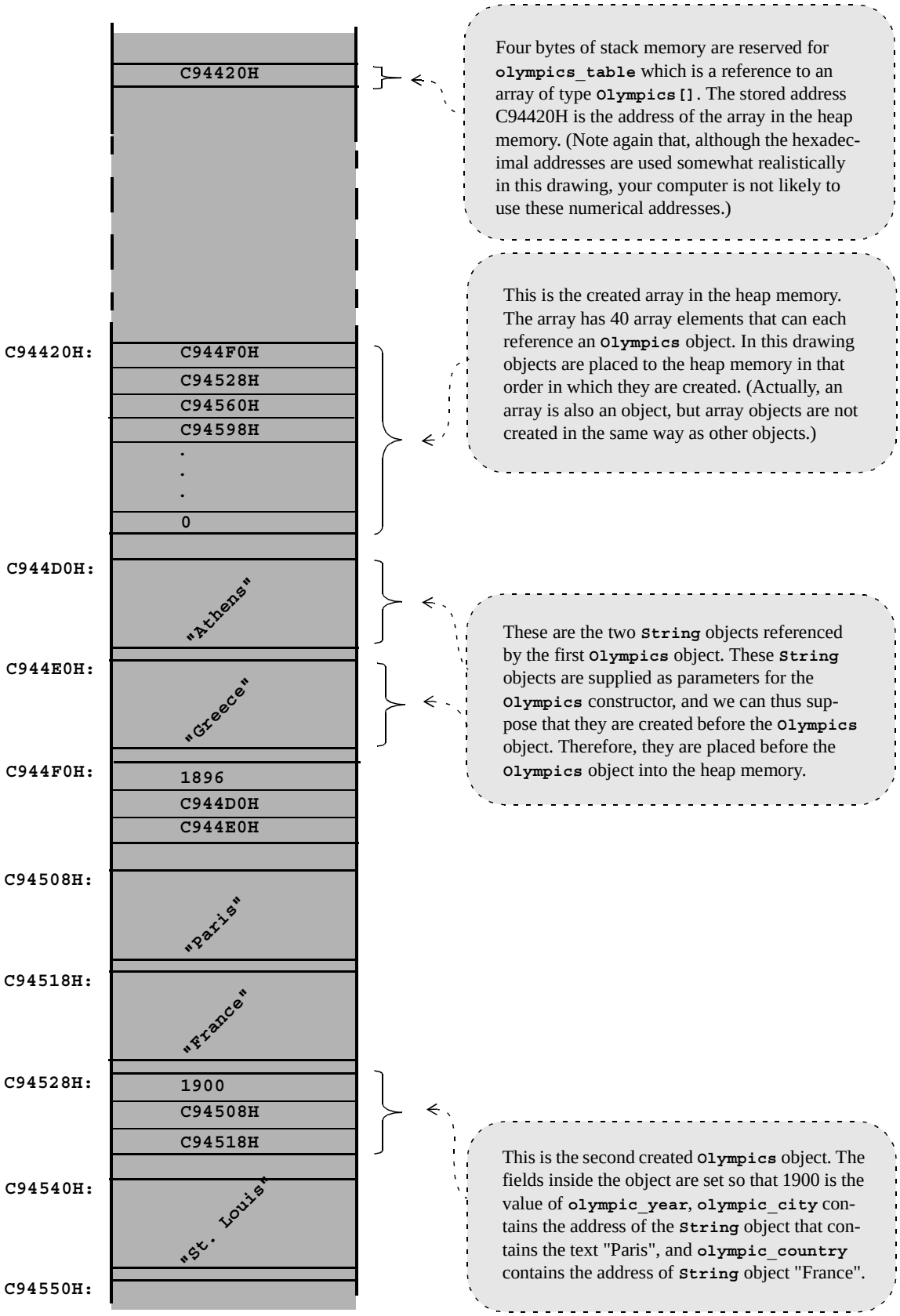


Figure 10-3. The objects of program Olympics.java in the main memory.

```
D:\javafiles3>java Convert liters 20

  20.0 liters is 5.284015852047556 gallons (U.S.)
  20.0 liters is 4.399472063352397 gallons (Br.)
  20.0 liters is 42.28329809725159 pints (U.S.)
  20.0 liters is 35.21126760563381 pints (Br.)
```

Convert.java - X. Here, the program is executed by giving input data on the command line.

Objects of class **Conversion** will be stored in an array in this program, which is a small intelligent system that can be asked to make conversions between various units of measure. For example, if the user of this program wants to know how much is 20 miles in kilometers, he or she can simply type the following on the command line

java Convert miles 20

```
//  Convert.java

import java.util.* ;

class  Conversion
{
   String  first_unit ;
   String  second_unit ;
   double  conversion_constant ;

   public Conversion( String  given_first_unit,
                      String  given_second_unit,
                      double  given_conversion_constant )
   {
      first_unit   =  given_first_unit ;
      second_unit  =  given_second_unit ;
      conversion_constant  =  given_conversion_constant ;
   }

   public void convert( String  given_unit,
                        double  amount_to_convert )
   {
      if ( first_unit.contains( given_unit ) )
      {
         System.out.print(  "\n  "  +  amount_to_convert  +  " "
                         +  first_unit  + " is "
                         +  amount_to_convert * conversion_constant
                         +  " "  +  second_unit ) ;
      }

      if ( second_unit.contains( given_unit ) )
      {
         System.out.print( "\n  "  +  amount_to_convert  +  " "
                         +  second_unit  + " is "
                         +  amount_to_convert / conversion_constant
                         +  " "  +  first_unit ) ;
      }
   }
}
```

The plus sign means that part of the shown program is explained in more detail later. In this case, method **convert()** is explained in a more explicit program description.

Convert.java - 1+: A program to make conversions between units of measure.

conversion_table references an array that contains references to **Conversion** objects. Each **Conversion** object contains the names of two units of measure, and a constant that tells how these two units relate to each other. The data with which the objects are initialized can be found in Physics books and almanacs. This line, for example, says that one mile is 1.609344 kilometers.

```
class  Convert
{
   public static void main( String[]  command_line_parameters )
   {
      Scanner  keyboard  =  new Scanner( System.in ) ;

      Conversion[]  conversion_table  =  new  Conversion[ 13 ] ;

      conversion_table[ 0 ] = new Conversion("meters", "yards", 1.093613 );
      conversion_table[ 1 ] = new Conversion("meters", "feet",  3.280840 );
      conversion_table[ 2 ] = new Conversion("miles",  "kilometers",1.609344);
      conversion_table[ 3 ] = new Conversion("inches", "centimeters", 2.54 );
      conversion_table[ 4 ] = new Conversion("acres",  "hectares", 0.4046873);
      conversion_table[ 5 ] = new Conversion("pounds", "kilograms",0.4535924);
      conversion_table[ 6 ] = new Conversion("ounces", "grams",    28.35 );
      conversion_table[ 7 ] = new Conversion("gallons (U.S.)","liters", 3.785);
      conversion_table[ 8 ] = new Conversion("gallons (Br.)", "liters", 4.546);
      conversion_table[ 9 ] = new Conversion("pints (U.S.)",  "liters", 0.473);
      conversion_table[ 10 ]= new Conversion("pints (Br.)",   "liters", 0.568);
      conversion_table[ 11 ]= new Conversion("joules",       "calories",4.187);
      conversion_table[ 12 ]= new Conversion("lightyears",   "kilometers",
                                                  9.461e12 ) ;
      String  unit_from_user ;
      int     amount_to_convert ;

      if ( command_line_parameters.length  ==  2 )
      {
         unit_from_user     =  command_line_parameters[ 0 ]  ;
         amount_to_convert  =  Integer.parseInt(
                                    command_line_parameters[ 1 ] ) ;
      }
      else
      {
         System.out.print( "\n Give the unit to convert from: " ) ;
         unit_from_user  =  keyboard.nextLine() ;
         System.out.print( " Give the amount to convert:   " ) ;
         amount_to_convert  =  Integer.parseInt( keyboard.nextLine() ) ;
      }

      for ( int conversion_index  =  0 ;
                conversion_index  <  conversion_table.length ;
                conversion_index  ++ )
      {
         conversion_table[ conversion_index ].convert( unit_from_user,
                                                       amount_to_convert ) ;
      }
   }
}
```

Convert.java - 2. The second part of the program.

Method **convert()** is called from method **main()** for every created **Conversion** object in the array referenced by **conversion_table**. The method is called without caring whether or not the given unit is represented by the **Conversion** object in question. If the **convert()** method cannot convert the given unit, it does not print anything to the screen.

```
public void convert( String  given_unit,
                     double  amount_to_convert )
   {
      if ( first_unit.contains( given_unit ) )
      {
         System.out.print(  "\n  "  +  amount_to_convert  +  " "
                         +  first_unit  + " is "
                         +  amount_to_convert * conversion_constant
                         +  " "  +  second_unit ) ;
      }

      if ( second_unit.contains( given_unit ) )
      {
         System.out.print( "\n  "  +  amount_to_convert  +  " "
                         +  second_unit  + " is "
                         +  amount_to_convert / conversion_constant
                         +  " "  +  first_unit ) ;
      }
   }
}
```

Method **contains()** returns the value **true** when the unit name stored by this object includes the unit string given as a parameter. This method attempts conversions from **first_unit** to **second_unit** and vice versa. Depending on which conversion is possible, either multiplication or division operation is used in conversion.

By using the method **contains()** instead of a more accurate string comparison method like **compareTo()**, it was possible to make this program more flexible. Although all the unit names inside the **Conversion** objects are in plural form (e.g. "pints"), the program also works when the user types in the units in singular form (e.g. "pint").

Instead of the **contains()** method it would be possible to use the **indexOf()** method. An alternative way to write the latter **if** construct would be:

```
if ( second_unit.indexOf( given_unit )  !=  -1 )
{
   ...
```

indexOf() is a string method that returns the index of the string that is given as a parameter. For example, if **second_unit** references the string "kilometers" and **given_unit** references the string "meter", the above call to method **indexOf()** returns 4 because the string "meter" starts in position with index 4 in the string "kilometers". **indexOf()** returns -1 when it cannot find the given substring.

Convert.java - 1 - 1. Method convert() of class Conversion.