Case Study—Part 1

11

Outcomes:

By the end of this chapter you should be able to:

- *describe each stage of the software development process*;
- ***design*** *a complete application using UML*;
- ***implement*** *a detailed UML design in Java*;
- ***document*** *their code using* ***Javadoc*** *comments*;
- *distinguish between* ***unit testing*** *and* ***integration testing***;
- ***test*** *individual program units by creating suitable drivers*;
- *document their test results professionally using a* ***test log***.

11.1 Introduction

The process of developing software requires us to carry out several tasks. They can be summarized as follows:

- **analysis and specification**: determining *what* the system is required to do (analysis) and writing it down in a clear and unambiguous manner (specification);
- **design**: making decisions about *how* the system is to be built in order to meet the specification;
- **implementation**: turning the design into an actual program;
- **testing**: ensuring that the system has been implemented correctly to meet the original specification;
- **installation**: delivering and setting up the completed system;
- **operation and maintenance**: running the final system and reviewing it over time—in light of changing requirements.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_11

Rather than completely finishing one task before beginning the next, object-oriented languages like Java encourage systems to be developed a little bit at a time. So, for example, we can build one class and test it (maybe in the presence of the client) before moving onto the next, rather than waiting for the whole system to be developed before testing and involving the client.

In this and the following chapter we will demonstrate this process by developing a case study that will enable you to get an idea of how a commercial system can be developed from scratch; we start with an informal description of the requirements, and then specify and design the system using UML notation and pseudocode where necessary. From there we go on to implement our system in Java. Java applications, such as this, typically consist of many classes working together. When testing for errors we will start with a process of **unit testing** (testing individual classes) followed by **integration testing** (testing classes that together make up an application).

The system that we are going to develop will keep records of the residents of a student hostel. In order not to cloud your understanding, we have simplified things, keeping details of individuals to a minimum, and keeping the functionality fairly basic; you will have the opportunity to improve on what we have done in the practical exercises at the end of the next chapter.

11.2 The Requirements Specification

The local university requires a program to manage one of its student hostels, which contains a number of rooms, each of which can be occupied by a single tenant who pays rent on a monthly basis. The program must keep a list of tenants and their monthly payments. The information held for each tenant will consist of a name, a room number and a list of all the payments a tenant has made (month and amount) for one year. The program must allow the user to add and delete tenants, to display a list of all tenants, to record a payment for a particular tenant, and to display the payment history of a tenant.

11.3 The Design

The two core classes required in this application are `Tenant` (to store the details of a tenant) and `Payment` (to store the details of a payment). We have made a number of design decisions about how the system will be implemented, and these are listed below:

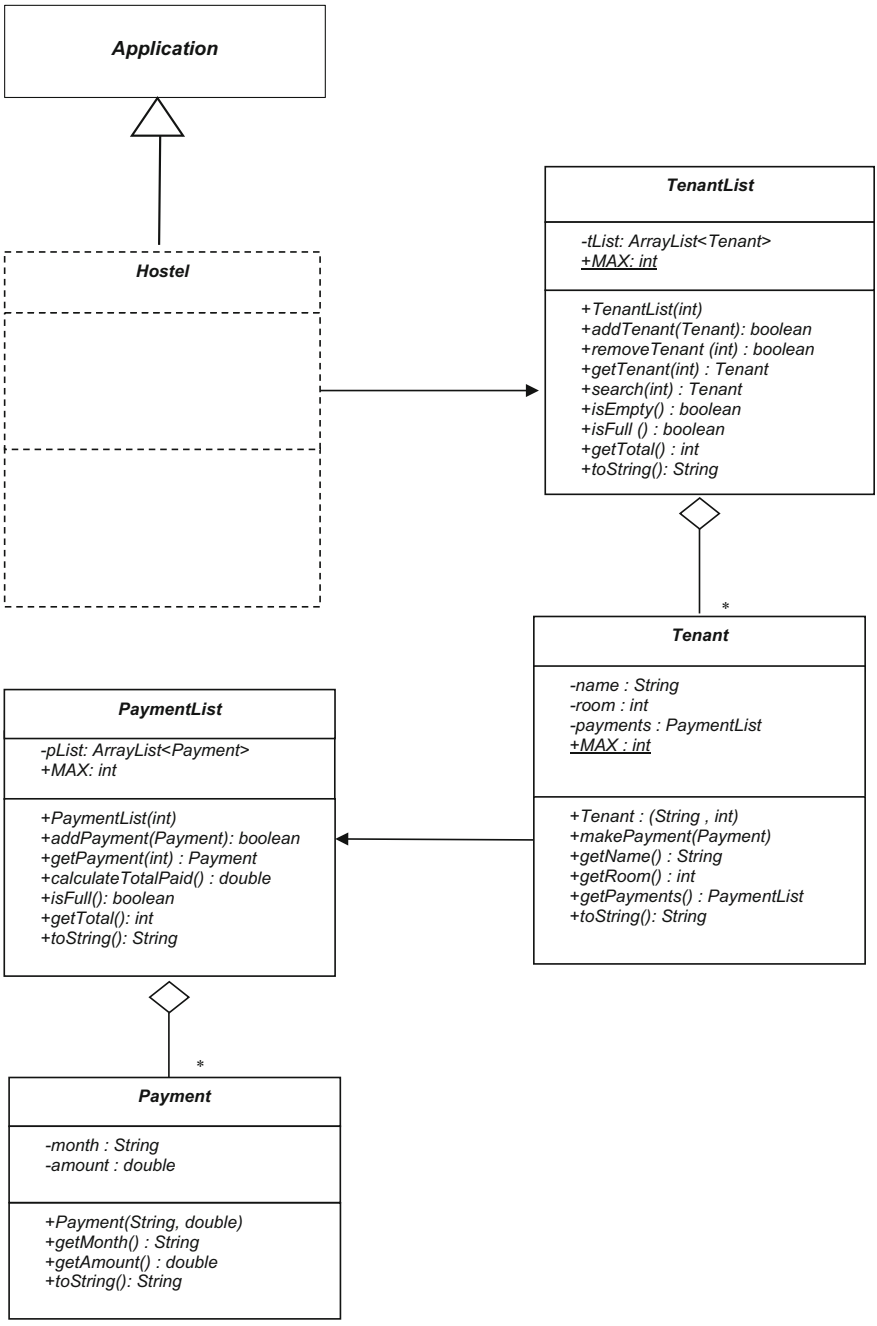


Fig. 11.1 The design of the student hostel system

- instances of the `Tenant` class and instances of the `Payment` class will each be held in a separate collection class, `PaymentList` and `TenantList` respectively;
- the collection classes `PaymentList` and `TenantList` both make use of an `ArrayList`;
- the `Hostel` class, which will hold the `TenantList`, will also act as the graphical interface for the system.

The design of the system is shown in Fig. 11.1. In this design there are two arrows from one class to another. In UML these represent **associations**. An association is a link from objects of one class to objects of another class. For example, a *customer* might have one or more *accounts*; a *student* might have one or more *tutors*. The simplest form of association is a one-to-one relationship whereby a single instance of one class is associated with a single instance of another class—for example a *purchase transaction* and an *invoice*. You have already come across inheritance and aggregation—these are special examples of association.

In our example, the associations represented by the arrows are one-to-one associations—a `Tenant` requires a single instance of `a PaymentList` and a `Hostel` requires a single instance of a `TenantList`.

The `Hostel` class itself has not yet been designed and this will be left until the next chapter where we consider the overall system design and testing; for this reason it has been drawn with a dotted line, but we can see it is a graphics class as it inherits from the JavaFX `Application` class.

In order to implement this application we should start with those classes that do not depend on any other, so that they can be unit tested in isolation. For example, we should not start by implementing the `Tenant` class as it requires the `PaymentList` class to be implemented first. You can see from the associations in Fig. 11.1 that the only class that does not require any other class for its implementation is the `Payment` class.

11.4 Implementing the *Payment* Class

Throughout this case study we will make use of the `Javadoc` style of comments (that we briefly mentioned back in Chap. 1) to document our classes. We will discuss how to read and write `Javadoc` comments in more detail in the next section.

The code for the `Payment` class is shown below.

Payment

```
/** Class used to store details of a single payment in a hostel
 *  @author Charatan and Kans
 *  @version 6th April 2018
 */
public class Payment
{
   private String month;
   private double amount;

   /** Constructor initialises the payment month and the amount paid
    *  @param  monthIn: month of payment
    *  @param  amountIn: amount of payment
    */
   public Payment(String monthIn, double amountIn)
   {
            month = monthIn;
            amount = amountIn;
   }

   /** Reads the month for which payment was made
    *  @return Returns the month for which payment was made
    */
   public String getMonth()
   {
            return month;
   }

   /** Reads the amount paid
    *  @return Returns the amount paid
    */
   public double getAmount()
   {
            return amount;
   }

   @Override
   public String toString()
   {
            return "("  + month + ", " + amount  + ")";
   }

}
```

As you can see, this class is fairly simple and does not require much explanation. Note that we have overridden the `toString` method (hence the `@Override` tag) to provide a convenient way of printing a `Payment` object (as discussed in Chap. 9).

```
@Override
public String toString()
{
       return "(" + month + " : " + amount + ")"; // a convenient way of displaying attributes
}
```

Before incorporating this class into a larger program you would to test if it was working reliably. Eventually, when this class is incorporated into the final program we will have a JavaFX `Hostel` class to run the application, but we need to test this class before an entire suite of classes has been developed. As we said before, testing an individual class in this way is often referred to as *unit testing*.

In order to unit test this class we will need to implement a separate class especially for this purpose. This new class will contain a `main` method and it will act as a **driver** for the original class. A driver is a special program designed to do

nothing except exercise a particular class. If you look back at all our previous examples, this is exactly how we tested individual classes. Initially you should generate an object from the given class. Once an object has been generated we can then test that object by calling its methods. When testing your class by generating objects and calling methods, you will want to display results on the screen, such as the data stored within your object. We could access this data by calling the appropriate `get` methods:

```
public class PaymentTester
{

    public static void main(String[ ] args)
    {
        Payment p1 = new Payment ("January", 175);

        // code to interrogate object data
        System.out.println("Month: " + p1.getMonth());
        System.out.println("Amount: " + p1.getAmount());
    }
}
```

This will display the expected output:

```
Month: January
Amount: 175.0
```

While having multiple output statements like this might be necessary in the final application, it is a rather cumbersome way of retrieving information from an object during the testing phase—when you will not be so concerned with the format of the output. This is where we can make use of the `toString` method we provided in the `Payment` class to display all attributes of the class in one print statement. Here is a modified tester:

PaymentTester

```
// a very simple driver program that makes use of the toString method

public class PaymentTester
{

    public static void main(String[ ] args)
    {
        Payment p1 = new Payment ("January", 175); // create object to test

        System.out.println(p1); // this will call the toString method in our Payment class
    }
}
```

Running this program will produce the following result:

```
(January : 175.0)
```

You can see the `Payment` object is displayed in the format given in our `toString` method. From this example, you can see how useful the `toString` method is. Now let's move on to the more interesting parts of this system. This

system requires us to develop two kinds of list, a `PaymentList` and a `TenantList`. Both make use of `ArrayList` to store the respective collections. A `TenantList` requires a `PaymentList` class to be developed first, so let's start by looking at this `PaymentList` class.

11.5 The *PaymentList* Class

The design of the `PaymentList` class is similar to a collection class that we showed you in Chap. 8—the `Bank` class. Both classes make use of an `ArrayList` to store a collection of objects. The main difference between the two classes is that in the `Bank` class the contained type was `BankAccount`, whereas in the `PaymentList` class the contained type is `Payment`. Also, we have included a constant `MAX`, to allow us to record the maximum number of payments we would like to record in our list. As `MAX` is a constant it has been given public visibility (+). Figure 11.2 provides a reminder of the design of the `PaymentList` class.

As you can see, as well as a constructor, there are methods to add a new payment to the list, to check if the list is full and to count the total number of payments made so far. There are also methods to get a payment based on a position number and to calculate the total payments made so far. Finally, once again we have included a `toString` method for ease of testing. Take a look at the code for the `PaymentList` class below before we discuss it—once again we are using the `Javadoc` style of comments which will be fully explained in the next section.

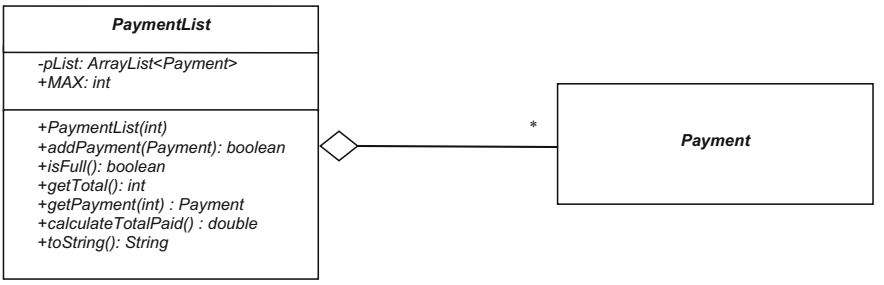


Fig. 11.2 The design of the `PaymentList` collection class

PaymentList

```
import java.util.ArrayList;

/** Collection class to hold a list of Payment objects
 *  @author Charatan and Kans
 *  @version 4th April 2018
 */
public class PaymentList
{
   // attributes
   private ArrayList<Payment> pList;
   public final int MAX;

   /** Constructor initialises the empty payment list and sets the maximum list size
    *  @param   maxIn: The maximum number of payments in the list
    */
   public PaymentList(int maxIn)
   {
           pList = new ArrayList<>();
           MAX = maxIn;
   }

   /** Checks if the payment list is full
    *  @return Returns true if the list is full and false otherwise
    */
   public boolean isFull()
   {
           return pList.size()== MAX;
   }

   /** Gets the total number of payments
    *  @return Returns the total number of payments currently in the list
    */
   public int getTotal()
   {
           return pList.size();
   }

   /** Adds a new payment to the end of the list
    *  @param  pIn: The payment to add
    *  @return Returns true if the object was added successfully and false otherwise
    */
   public boolean addPayment(Payment pIn)
   {
       if(!isFull())
       {
              pList.add(pIn);
              return true;
       }
       else
       {
              return false;
       }
   }

   /** Reads the payment at the given position in the list
    *  @param     positionIn: The logical position of the payment in the list
    *  @return    Returns the payment at the given logical position in the list
    *             or null if no payment at that logical position
    */
   public Payment getPayment(int positionIn)
   {
       //check for valid logical position
       if (positionIn <1 || positionIn > getTotal())
       {
              // no object found at given position
              return null;
       }
       else
       {
              // take one off logical position to get ArrayList position
             return pList.get(positionIn - 1);
       }
   }

   /** Calculates the total payments made by the tenant
    *  @return  Returns the total value of payments recorded
    */
   public double calculateTotalPaid()
   {
       double totalPaid = 0; // initialize totalPaid
       // loop through all payments
       for (Payment p: pList)
       {
              // add current payment to running total
              totalPaid = totalPaid + p.getAmount();
       }
       return totalPaid;
   }

   @Override
   public String toString()
   {
           return pList.toString();
   }
}
```

Firstly you can see that we have imported the `ArrayList` class from the `java.util` package:

```
import java.util.ArrayList;
```

We make use of the `ArrayList` class to store a collection of payments in the `pList` attribute. In addition to this attribute we have a **`public`** constant value `MAX` to record the maximum number of payments that we can record:

```
// attributes
private ArrayList<Payment> pList;
public final int MAX;
```

The constructor, as well as initialising the `ArrayList` attribute `pList`, sets the value for `MAX` via a parameter (`maxIn`) sent to the constructor:

```
/** Constructor initialises the empty payment list and sets the maximum list size
 *  @param   maxIn: The maximum number of payments in the list
 */
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       MAX = maxIn;
}
```

Now let's take a look at the remaining methods of this class. Firstly, we have an `isFull` method. The `pList` will be full when its size is equal to the value of our constant `MAX`:

```
public boolean isFull()
{
     return pList.size()== MAX; // use the size method of ArrayList
}
```

You can see that the `size` method of `ArrayList` has been used to check the number of items currently in the `pList`.

The `addPayment` method makes use of `isFull` to check if there is space in our `pList` before adding a new `Payment` object. A **`boolean`** value is returned to indicate success or failure:

```
public boolean addPayment(Payment pIn)
{
    if(!isFull()) // ok to add Payment
    {
        pList.add(pIn);
        return true;
    }
    else
    {
        return false; // Payment not added to a full list
    }
}
```

The `size` method of `ArrayList` that we met earlier is used again to implement the `getTotal` method, which returns the total number of payments made so far:

```
public int getTotal()
{
    return pList.size();
}
```

The `getPayment` method makes use of `getTotal` to check that the validity of the parameter `positionIn`. The `positionIn` parameter is the logical position of a payment in the list, which should be a number between 1 and the total number of items in the list. If an invalid position is sent the **`null`** value is returned, otherwise the payment at the associated `ArrayList` position is returned:

```
public Payment getPayment(int positionIn)
{
   //check for invalid logical position
   if (positionIn <1 || positionIn > getTotal())
   {
       // no object to return at the position
       return null;
   }
   else
   {
       // take one off logical position to get ArrayList position
      return pList.get(positionIn - 1);
   }
}
```

Next, the `calculateTotalPaid` method computes the sum of all payments in the list. This `calculateTotalPaid` method uses a standard algorithm for computing sums from a list of items. We met such an algorithm in Sect. 6.8.2 of this book. This algorithm can be expressed in pseudocode as follows:

```
SET totalPaid TO 0
LOOP FROM first item in list TO last item in list
BEGIN
   SET totalPaid TO totalPaid + amount of current payment
END
return totalPaid
```

Since the loop in this algorithm is just reading the items in the payment list, it can be implemented in Java with the use of an enhanced **`for`** loop:

```
public double calculateTotalPaid()
{
   double totalPaid = 0; // initialize totalPaid
   // loop through all payments
   for (Payment p: pList)
   {
       // add current payment to running total
       totalPaid = totalPaid + p.getAmount();
   }
   return totalPaid; // return sum
}
```

Finally, we have overridden the `toString` method again to allow the payment list to be displayed as a single `String`. This may seem like quite a challenging task when dealing with a collection of objects. Maybe we need to loop through the

list and join all the payments together to form a single `String`? Luckily, we do not have to go to such lengths as the `ArrayList` class has a `toString` method built in! So all we need to do is call the `toString` method of the `ArrayList` attribute here:

```
@Override
public String toString()
{
    return pList.toString(); // call toString of ArrayList

}
```

If we assume we have three payment objects in the payment list, say ("Jan" : 310), ("Feb" : 280) and ("March" : 310), the `toString` method of `ArrayList` would return a `String` that looks as follows:

[("Jan" : 310), ("Feb" : 280), ("March" : 310)]

As you can see, the `toString` method of `ArrayList` calls the `toString` method of the contained items, separates them by commas and encloses the whole list in a pair of square brackets.

We will make use of this `toString` method during the testing of the `PaymentList` class, but first let's have a closer look at the `Javadoc` comments that we have included in the `PaymentList` class and the `Javadoc` tool itself.

11.5.1 Javadoc

Oracle's Java Development Kit contains a tool, `Javadoc`, which allows you to generate documentation for classes in the form of HTML files. In order to use this tool you must comment your classes in the `Javadoc` style. As we mentioned in Chap. 1, `Javadoc` comments must begin with /** and end with */. `Javadoc` comments can also contain 'tags'. Tags are special formatting markers that allow you to record information such as the author of a piece of code. Table 11.1 gives some commonly used tags in `Javadoc` comments.

The `@author` and `@version` tags are used in the `Javadoc` comments for the class as a whole. You can see examples of these tags at the top of the `PaymentList` class:

```
/** Collection class to hold a list of Payment objects
 *  @author Charatan and Kans
 *  @version 4th April 2018
 */
public class PaymentList
{
    // attributes and methods go here
}
```

When `Javadoc` comments run over several lines, as in the example above, it is common (though not necessary) to begin each line with a leading asterisk.

Table 11.1 Some Javadoc tags

Tag	Information
`@author`	The name(s) of the code author(s)
`@version`	A version number for the code (often a date is used here)
`@param`	The name of a parameter and its description
`@return`	A description of the return value of a method

The `@param` and `@return` tags can be used in the `Javadoc` comments preceding each method. The `@param` tag is used to name and describe the purpose of a given parameter. The `@return` tag is used to describe the value returned by a method. Here for example are the `Javadoc` comments for the `addPayment` method, which makes use of both the `@param` and `@return` tags.

```
/** Adds a new payment to the end of the list
 *  @param  pIn: The payment to add
 *  @return Returns true if the object was added successfully and false otherwise
 */
public boolean addPayment(Payment pIn)
{
    if(!isFull())
    {
        pList.add(pIn);
        return true;
    }
    else
    {
        return false;
    }
}
```

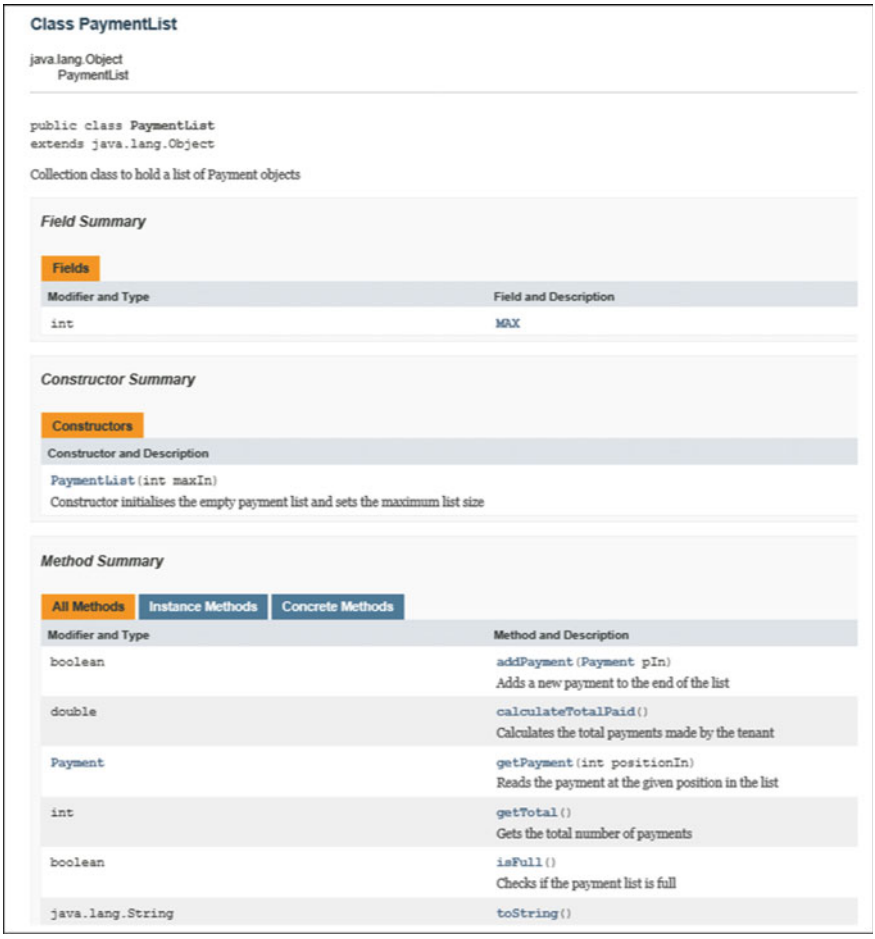
The `@param` tag has been used to provide a comment on the parameter (`pIn`) and the `@return` tag has been used to comment the role of the **`boolean`** value returned by this method. Using `Javadoc` comments in this way provides a comprehensive explanation of the functionality of a class. Take a look at the remaining `Javadoc` comments used in the `PaymentList` class for more examples of these tags.

The `Javadoc` HTML documentation files themselves can then be generated either from the command line using the **`javadoc`** command:

```
javadoc PaymentList.java
```

or invoked directly by your IDE. Figure 11.3 gives part of the documentation generated as a result of the `PaymentList` class `Javadoc` comments.

Comments, such as the `Javadoc` comments we added into the `PaymentList` class, provide one technique for documenting the code that you write. Documenting your code is important as it assists in the maintenance of that code, should it need to be modified in the future. Well documented code is also easier to fix if errors arise during development. As well as commenting your code, you should ensure that the code is well laid out so that it becomes easier to read and follow; more about this in a moment.



Class PaymentList

java.lang.Object
PaymentList

```
public class PaymentList
extends java.lang.Object
```

Collection class to hold a list of Payment objects

Field Summary

Fields

Modifier and Type	Field and Description
int	MAX

Constructor Summary

Constructors

Constructor and Description
PaymentList(int maxIn) Constructor initialises the empty payment list and sets the maximum list size

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	addPayment(Payment pIn) Adds a new payment to the end of the list
double	calculateTotalPaid() Calculates the total payments made by the tenant
Payment	getPayment(int positionIn) Reads the payment at the given position in the list
int	getTotal() Gets the total number of payments
boolean	isFull() Checks if the payment list is full
java.lang.String	toString()

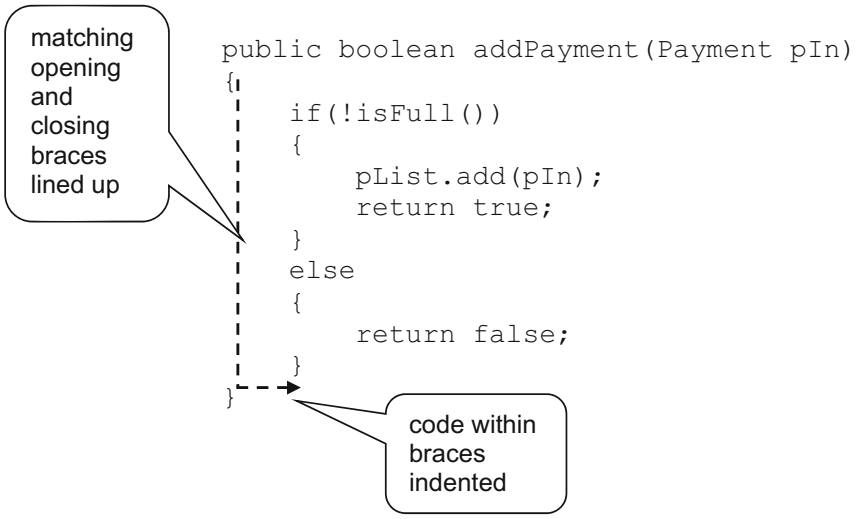
Fig. 11.3 `Javadoc` documentation generated for the `PaymentList` class

11.5.2 Code Layout

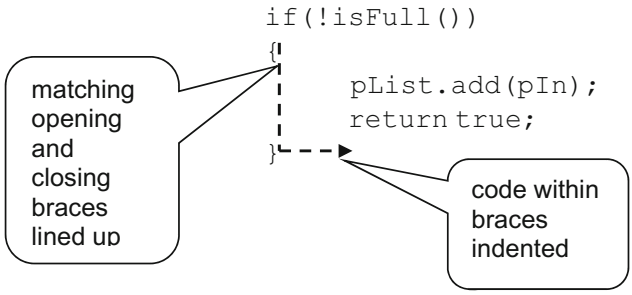
Consistent and clear indentation is important to improve the readability of your programs. Look at the example programs that we have presented to you and notice the care we have taken with our indentation. We are following two simple rules all the time:

- keep braces lined up under the structure to which they belong;
- indent, by one level, all code that belongs within those braces.

For example, look again at the `addPayment` method of the `PaymentList` class:



Notice how these rules are applied again with the braces of the inner **`if...else`** statements:



11.6 Testing the *PaymentList* Class

As we have said before, it is always important to test classes in order to ensure that they are functioning correctly before moving on to the rest of the development. Whereas the testing of the `Payment` class was an example of *unit testing*, testing this `PaymentList` class is an example of *integration testing* as it requires the `PaymentList` class working in conjunction with the `Payment` class.

To test the `PaymentList` class we need a driver that not only creates a `PaymentList` object, but also creates payments to add to this list.

We have quite a few methods to test in our `PaymentList` class, so we need to spend some time considering how we will go about testing these methods. For example, it would make sense to limit the size of our `PaymentList` so that we

can quickly fill up our list to check the `isFull` method. Here is one possible test strategy:

1. limit the size of the `PaymentList` to a relatively small number (say 4) using the `PaymentList` constructor;
2. add two payments to this list, say ("Jan", 310) and ("Feb", 280) using the `addPayment` method;
3. display the list (using the `toString` method) to check the items have been added successfully;
4. check to see if the `isFull` method returns **`false`**;
5. add two more payments to this list, say ("March", 310) and ("April", 300) using the `addPayment` method;
6. display the list (using the `toString` method) to check the items have been added successfully;
7. check to see if the `isFull` method returns **`true`**;
8. get details of one of the payments made (say the second payment) using the `getPayment` method.
9. attempt to retrieve a payment at an invalid position (say 5);
10. display the total number of payments made so far (using the `getTotal` method);
11. display the total of the payments made so far (using the `calculateTotalPaid` method);
12. attempt to add another payment to this full list using the `addPayment` method.

Notice that the test strategy should ensure that all methods of the class are tested and all possible routes through a method are tested. So, for example, as well as ensuring that the `isFull` method is called ensure we provide a scenario where the method should return **`true`** and provide a scenario where the method should return **`false`**.

Once a strategy is chosen, the test results should be logged in a **test log**. A test log is a document that records the testing that took place during system development. Each row of the test log associates an *input* with an *expected output*. If the output is not as expected, reasons for this error have to be identified and recorded in the log.

Figure 11.4 illustrates a suitable test log to document the testing strategy we developed above.

Test logs such as this should be devised *before* the driver itself (and may even be developed before the class we are testing has been developed). The test log can then be used to prompt the development of the driver. As you can see by looking at the test in Fig. 11.4, we assume that the driver is a menu driven program.

TEST LOG			
Purpose: To test the PaymentList class			
Run Number:	**Date:**		
Action	**Expected Output**	**Pass/ Fail**	**Reason for failure**
-	Prompt for size of list		
Enter 4	Display menu of options		
Select ADD option	Prompt for Payment details		
Enter "Jan", 310	Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "Feb", 280	Display menu of options		
Select DISPLAY option	Message [(Jan : 310.0), (Feb : 280.0)] Display menu of options		
Select IS FULL option	Message "list is NOT full" Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "March", 310	Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "April", 300	Display menu of options		
Select DISPLAY option	Message [(Jan : 310.0), (Feb : 280.0), (March : 310.0), (April : 300.0)] Display menu of options		
Select IS FULL option	Message "list is full" Display menu of options		
Select GET PAYMENT option	Prompt for position to retrieve		
Enter 2	Message (Feb : 280.0) Display menu of options		
Select GET PAYMENT option	Prompt for position to retrieve		
Enter 5	Message INVALID PAYMENT NUMBER Display menu of options		
Select GET TOTAL option	Message 4 Display menu of options		
Select CALCULATE TOTAL PAID option	Message 1200.0 Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "May", 310	Message Error – list full message Display menu of options		
Select EXIT option	Program terminates		

Fig. 11.4 Test log for the `PaymentList` class

The `PaymentListTester` program is one possible driver we could develop in order to process the actions given in this test log:

PaymentListTester

```
public class PaymentListTester
{

    public static void main(String[] args)
    {
       char choice;
       int size;
       PaymentList list; // declare PaymentList object to test

       // get size of list
       System.out.print("Size of list? ");
       size = EasyScanner.nextInt();
       list = new PaymentList(size); // create object to test
       // menu
       do
       {
            // display options
            System.out.println();
            System.out.println("[1] ADD");
            System.out.println("[2] DISPLAY");
            System.out.println("[3] IS FULL");
            System.out.println("[4] GET PAYMENT");
            System.out.println("[5] GET TOTAL");
            System.out.println("[6] CALCULATE TOTAL PAID");
            System.out.println("[7] Quit");
            System.out.println();
            System.out.print("Enter a choice [1-7]: ");
            // get choice
            choice = EasyScanner.nextChar();
            System.out.println();
            // process choice
            switch(choice)
            {
                case '1': option1(list); break;
                case '2': option2(list); break;
                case '3': option3(list); break;
                case '4': option4(list); break;
                case '5': option5(list); break;
                case '6': option6(list); break;
                case '7': System.out.println("TESTING COMPLETE"); break;
                default: System.out.print("1-7 only");
            }
       } while (choice != '7');
    }

    // ADD
    static void option1(PaymentList listIn)
    {
        // prompt for payment details
        System.out.print("Enter Month: ");
        String month = EasyScanner.nextString();
        System.out.print("Enter Amount: ");
        double amount = EasyScanner.nextDouble();
        // create new Payment object from input
        Payment p = new Payment(month, amount);
        // attempt to add payment to list
        boolean ok = listIn.addPayment(p); // value of false sent back if unable to add
        if (!ok)// check if item was not added
        {
            System.out.println("ERROR: list full");
        }
    }

    // DISPLAY
    static void option2(PaymentList listIn)
    {
        System.out.println("ITEMS ENTERED");
        System.out.println(listIn); // calls toString method of PaymentList
    }

    // IS FULL
    static void option3(PaymentList listIn)
    {
        if (listIn.isFull())
        {
            System.out.println("list is full");
        }
        else
        {
            System.out.println("list is NOT full");
        }
    }

    // GET PAYMENT
    static void option4(PaymentList listIn)
    {
        // prompt for and receive payment number
        System.out.print("Enter payment number to retrieve: ");
```

```
        int num = EasyScanner.nextInt();
        // retrieve Payment object form list
        Payment p = listIn.getPayment(num); // returns null if invalid position
        if (p != null)// check if Payment retrieved
        {
            System.out.println(p); // calls toString method of Payment
        }
        else
        {
            System.out.println("INVALID PAYMENT NUMBER"); // invalid position error
        }
    }

    // GET TOTAL
    static void option5(PaymentList listIn)
    {
        System.out.print("TOTAL NUMBER OF PAYMENTS ENTERED: ");
        System.out.println(listIn.getTotal());
    }

    // GET TOTAL PAID
    static void option6(PaymentList listIn)
    {
        System.out.print("TOTAL OF PAYMENTS MADE SO FAR: ");
        System.out.println(listIn.calculateTotalPaid());
    }
}
```

We are now in a position to run the driver and check the actions documented in the test log:

```
Size of list? 4
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: Jan
Enter Amount: 310
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: Feb
Enter Amount: 280
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
```

```
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 2
ITEMS ENTERED
[(Jan : 310.0), (Feb : 280.0)]
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 3
list is NOT full
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: March
Enter Amount: 310
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: April
Enter Amount: 300
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
```

```
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 2
ITEMS ENTERED
[(Jan : 310.0), (Feb : 280.0), (March : 310.0), (April :
  300.0)]
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 3
list is full
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 4
Enter payment number to retrieve: 2
(Feb : 280.0)
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 4
Enter payment number to retrieve: 5
INVALID PAYMENT NUMBER
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
```

```
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 5
TOTAL NUMBER OF PAYMENTS ENTERED: 4
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 6
TOTAL OF PAYMENTS MADE SO FAR: 1200.0
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
  [7] Quit
Enter a choice [1-7]: 1
Enter Month: May
Enter Amount: 310
ERROR: list full
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 7
TESTING COMPLETE
```

You have seen menu driven tester programmes such as this before so we will not discuss it in any detail. Note that our `EasyScanner` class has been used in this tester for ease of keyboard input. For example:

```
// get size of list
System.out.print("Size of list? ");
size = EasyScanner.nextInt(); // EasyScanner used to simplify keyboard input
```

Also, notice how the display method of our PaymentListTester (option 2 on the menu) displays the PaymentList object using the toString method we discussed earlier:

```
// DISPLAY
static void option2(PaymentList listIn)
{
    System.out.println("ITEMS ENTERED");
    System.out.println(listIn); // calls toString method of PaymentList
}
```

The output from this method gives us the ArrayList values in the format we discussed earlier, for example:

```
ITEMS ENTERED
[(Jan : 310.0), (Feb : 280.0), (March : 310.0), (April :
  300.0)]
```

If unexpected results are produced during testing, you should stop and identify the cause of the error in the class that you are testing. Both the cause of the error and how the error was fixed should be documented in the test log. The driver can then be run again with a fresh test log and this process should continue until *all* results are delivered as predicted. In this case, however, the results were as expected, so we can now move on to developing the rest of our system. We have two more classes to look at Tenant and TenantList. Before we look at the TenantList class we need to implement the Tenant class.

11.7 Implementing the *Tenant* Class

As you can see from the UML diagram of Fig. 11.1, the Tenant class contains four attributes:

- name;
- room;
- payments;
- MAX.

The first two of these represent the name and the room of the tenant respectively. The third attribute, payments, is to be implemented as a PaymentList object and the last attribute, MAX, is to be implemented as a **static** class attribute. The MAX attribute will also be implemented as a *constant* as we are assuming that tenants make a *fixed* number of payments in a year (twelve—one for each month). Since class constants cannot be modified, it makes sense to allow them to be declared as **public**. Below is the code for the Tenant class.

Tenant

```
/** Class used to record the details of a tenant
 *  @author Charatan and Kans
 *  @version 6th April 2018
 */
public class Tenant
{
   private String name;
   private int room;
   private PaymentList payments;
   public static final int MAX = 12;

  /** Constructor initialises the name and room number of the tenant
   *  and sets the payments made to the empty list
   *  @param  nameIn: name of tenant
   *  @param  roomIn: room number of tenant
   */
   public Tenant(String nameIn, int roomIn)
   {
            name = nameIn;
            room = roomIn;
            payments = new PaymentList(MAX);
   }

  /** Records a payment for the tenant
   *  @param paymentIn: payment made by tenant
   */
   public void makePayment(Payment paymentIn)
   {
            payments.addPayment(paymentIn); // call PaymentList method
   }

  /** Reads the name of the tenant
   *  @return Returns the name of the tenant
   */
   public String getName()
   {
            return name;
   }

  /** Reads the room of the tenant
   *  @return Returns the room of the tenant
   */
   public int getRoom()
   {
            return room;
   }

  /** Reads the payments of the tenant
   *  @return Returns the payments made by the tenant
   */
   public PaymentList getPayments()
   {
            return payments;
   }

   @Override
   public String toString()
   {
            return name+", "+room +", "+payments;
   }
}
```

The Javadoc comments should be sufficient documentation for you to follow the code in this class. Note how the Javadoc comments for the constructor includes two @param tags, as the constructor has two parameters. Also, it is worth noting that the payments attribute, being of type PaymentList, can respond to any of the PaymentList methods we discussed in Sect. 11.5. The makePayment method illustrates this by calling the addPayment method of PaymentList:

```
public void makePayment(Payment paymentIn)
{
    payments.addPayment(paymentIn); // call PaymentList method
}
```

We will leave the testing of this class and the next TenantList class as exercises for you as end of chapter programming exercises.

11.8 Implementing the *TenantList* Class

The TenantList class is a collection class to hold our Tenant objects. Once again we use an ArrayList to store this collection and have a MAX constant to fix an upper limit on the number of tenants our hostel can accommodate. A reminder of the design of the TenantList class is given in Fig. 11.5.

Most of the methods of the TenantList class should be familiar to you from the PaymentList collection classed that we discussed earlier. We will just take a closer look at two methods that were not mirrored in the PaymentList class, namely the remove and search methods. Let's start with the search method. Here is a reminder of its UML interface:

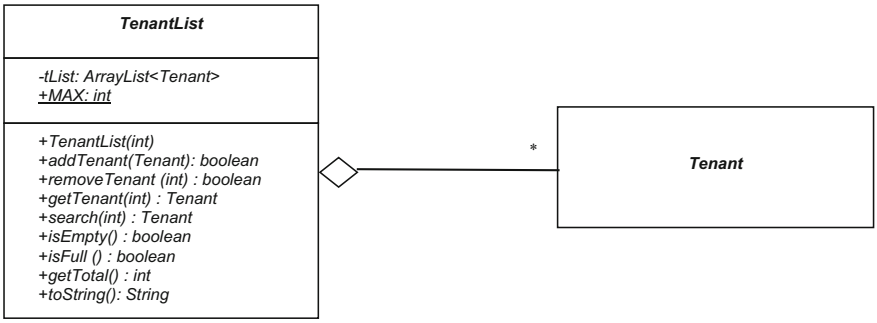


Fig. 11.5 The design of the TenantList collection class

search (int): Tenant

The integer parameter represents the room number of the tenant that this method is searching for. The tenant returned is the tenant living in that particular room; if no tenant is found in that room then **`null`** is returned. Here is a suitable algorithm for finding a tenant, expressed in pseudocode:

```
LOOP FROM first tenant in list TO last tenant in list
BEGIN
    IF current tenant's room number = room to locate
    BEGIN
        return current tenant
    END
END
return null
```

This is similar to the algorithm we looked at for searching an array in Sect. 8.8.1 except that we return the given item in the list rather than its index. This algorithm requires a loop to search through the tenants in the list and an enhanced **`for`** loop is a good way to do this:

```
public Tenant search(int roomIn)
{
    for(Tenant currentTenant: tList)// enhanced for loop to serach through the list of tenants
    {
        // find tenant with given room number
        if(currentTenant.getRoom() == roomIn)
        {
            return currentTenant;
        }
    }
     return null; // no tenant found with given room number
 }
```

Now let's look at the `removeTenant` method. The UML interface for this method is as follows:

removeTenant(int): boolean

Here the integer parameter represents the room number of the tenant that is to be removed from the list and the **`boolean`** return value indicates whether or not such a tenant has been removed successfully.

The previous `search` method can be used here to determine if a tenant exists in that particular room (a value of **`null`** will be returned if no such tenant exists). If such a tenant does exist it can be removed from the list using the `remove` method

of `ArrayList` and a **`boolean`** value of **`true`** can be returned, otherwise a **`boolean`** value of **`false`** can be returned. Here is the code:

```
public boolean removeTenant(int roomIn)
{
    Tenant findT = search(roomIn); // call search method
    if (findT != null) // check tenant is found at given room
    {
        tList.remove(findT); // remove given tenant
        return true;
    }
    else
    {
        return false; // no tenant in given room
    }
}
```

The complete code for the `TenantList` class is now presented below. The `Javadoc` comments provided should now provide sufficient explanation of each part of this class.

TenantList

```
import java.util.ArrayList;

/** Collection class to hold a list of tenants
 *  @author Charatan and Kans
 *  @version 6th April 2018
 */
public class TenantList
{
    private ArrayList<Tenant> tList;
    public final int MAX;

    /** Constructor initialises the empty tenant list and sets the maximum list size
     *  @param   maxIn The maximum number of tenants in the list
     */
    public TenantList(int maxIn)
    {
        tList = new ArrayList<>();
        MAX = maxIn;
    }

    /** Adds a new Tenant to the list
     *  @param  tIn The Tenant to add
     *  @return Returns true if the tenant was added successfully and false otherwise
     */
    public boolean addTenant(Tenant tIn)
    {
        if(!isFull())
        {
            tList.add(tIn);
            return true;
```

```
        }
        else
        {
            return false;
        }
    }

    /** Removes the tenant in the given room number
     *  @param roomIn The room number to of the tenant to remove
     *  @return Returns true if the tenant is removed successfully or false otherwise
     */
    public boolean removeTenant(int roomIn)
    {
        Tenant findT = search(roomIn); // call search method
        if (findT != null) // check tenant is found at given room
        {
            tList.remove(findT); // remove given tenant
            return true;
        }
        else
        {
            return false; // no tenant in given room
        }
    }

    /** Searches for the tenant in the given room number
     *  @param roomIn The room number to search for
     *  @return Returns the tenant in the given room or null if no tenant in the given room
     */
    public Tenant search(int roomIn)
    {
        for(Tenant currentTenant: tList)
        {
            // find tenant with given room number
            if(currentTenant.getRoom() == roomIn)
            {
                return currentTenant;
            }
        }
        return null; // no tenant found with given room number
    }

    /** Reads the tenant at the given position in the list
     *  @param      positionIn The logical position of the tenant in the list
     *  @return     Returns the tenant at the given logical position in the list
     *              or null if no tenant at that logical position
     */
    public Tenant getTenant(int positionIn)
    {
        if (positionIn<1 || positionIn>getTotal()) // check for valid position
        {
            return null; // no object found at given position
        }
        else
        {
            // remove one frm logical poition to get ArrayList position
            return tList.get(positionIn -1);
        }
    }


    /** Reports on whether or not the list is empty
     *  @return Returns true if the list is empty and false otherwise
     */
    public boolean isEmpty()
    {
        return tList.isEmpty();
    }

    /** Reports on whether or not the list is full
     *  @return Returns true if the list is full and false otherwise
     */
    public boolean isFull()
    {
        return tList.size()== MAX;
    }

    /** Gets the total number of tenants
     *  @return Returns the total number of tenants currently in the list
     */
    public int getTotal()
    {
        return tList.size();
    }

    @Override
    public String toString()
    {
        return tList.toString();
    }
}
```

All that remains for us to do to complete our case study in the next chapter is to design, implement and test the `Hostel` class which will not only keep track of the tenants but will also act as the graphical user interface for the system.

11.9 Self-test Questions

1. Describe the class associations given in the UML design of Fig. 11.1.
2. Produce suitable `Javadoc` comments for the `Oblong` class from Chap. 8.
3. The test log of Sect. 11.5.3 did not include checks for the `getItem` and `getTotal` methods of the `PaymentList` class. It also did not include a check that attempts to add to a full list and remove from an empty list would fail. Modify the test log to include these checks.
4. Develop test logs for testing the `Tenant` and `TenantList` classes.
5. Identify the benefits of adding a `toString` method into your classes and then write a suitable `toString` method for the `Bank` class from Chap. 8.

11.10 Programming Exercises

You will need to copy the entire suite of classes that make up the student hostel system and the Bank and BankApplication classes from the website.

1. Modify and then run the driver given in Program 11.2 in light of the changes made to the test log in self-test question 3 above.
2. Develop suitable drivers to test the `Tenant` and `TenantList` classes.
3. Use the test logs you developed in self-test question 5 and the drivers you developed in Exercise 2 above to test the `Tenant` and `TenantList` classes.
4. Incorporate the `toString` method into the `Bank` class from Chap. 8 you developed in self-test question 4 above then modify and run the `BankApplication` tester program from the same chapter to test make use of this `toString` method.