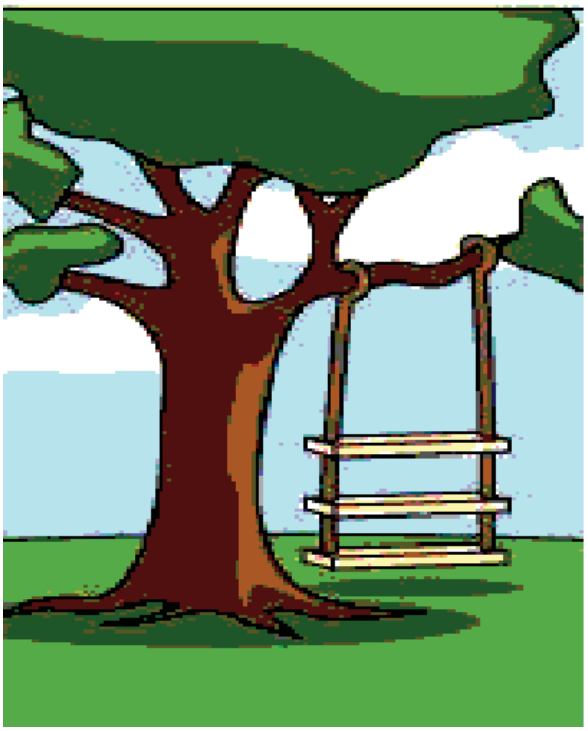
1. Using the Java Application Programming Interface

Chapter One takes examples from the Java Application Programming Interface (API) and the themed application in order to emphasise the critical importance of documentation. The examples are used to show how documentation is organised in the API and how it is inserted into developer's code.

1.1 Documentation in Developer-Written Java Classes

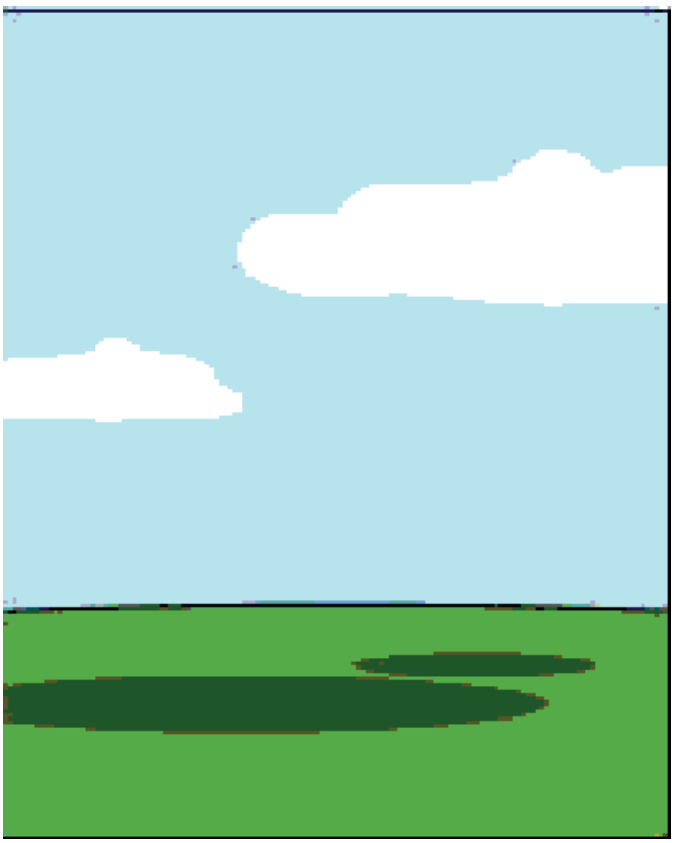
Previous chapters include a number of examples of classes (or partial classes) from the themed application, the main purpose of which is to illustrate programming concepts. It should not have escaped the notice of the reader that these examples often include material other than Java statements, usually in a non-bold font. The inclusion of such material raises a question: *what is the purpose of documentation is a class definition?*

Perhaps one way to address this question is to consider figures 1.1 and 1.2 below.



Source: http://www.cvr-it.com/PM_Jokes.htm

Figure 1.1 A view of an application as explained by the users



Source: http://www.cvr-it.com/PM_Jokes.htm

Figure 1.2 How the programme was documented

While it is obvious that the images are meant to be amusing, they make a serious point: *there is nothing worse than trying to maintain code written by someone else if there is little or no documentation.*

Documentation is an integral part of a Java class in that it is used, inter alia, to explain the purpose of members of the class (to the development team) and show consistency with the class diagram.

1.1.1 Documentation in the Themed Application

The class diagrams of two of the classes of the themed application are shown in the figure on the next page.

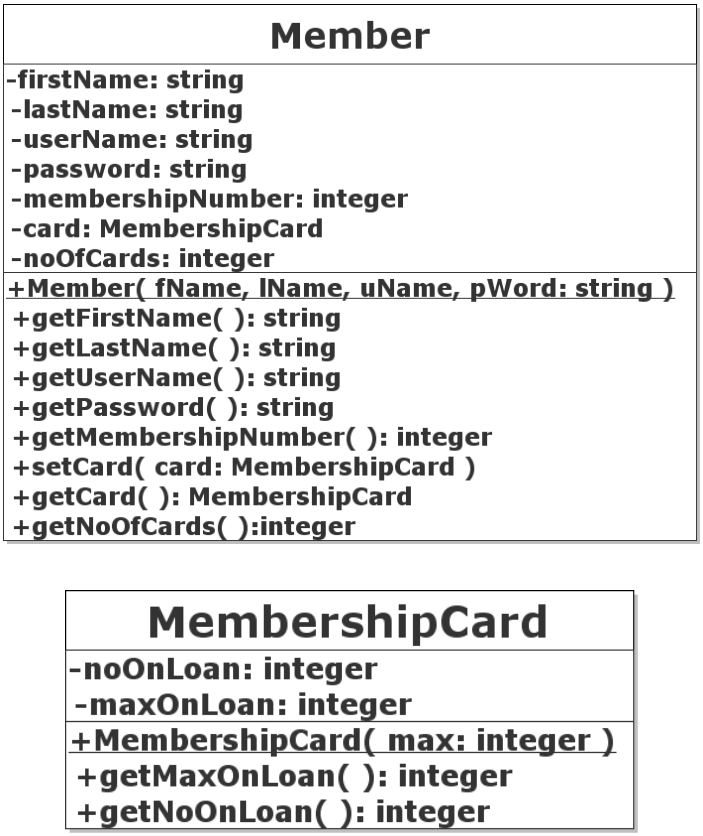


Figure 1.3 The Member and MembershipCard classes of the themed application

The classes are related by a 'has a' relationship, as shown in the next figure.

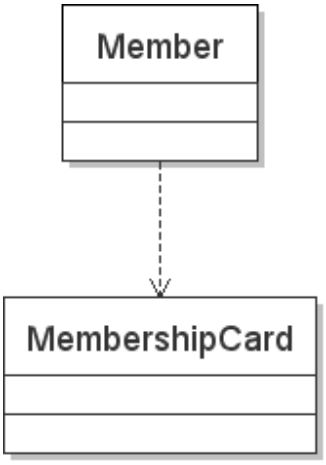


Figure 1.4 Each Member 'has a' MembershipCard

The source code for the **Member** class follows next. Some of the Java statements, white space and single-line comments are omitted for the sake of brevity so that the learner can concentrate on Java documentation, rather than on the logic of the code. Documentation blocks are displayed in a **bold** font.

```
/**
* The purpose of the Member class is … < to be completed by the developer >.
* @author  David M. Etheridge.
* @version 1.0, dated 29 October 2008.
*/
public class Member {
        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private int membershipNumber;
        private String noOfCards;
        private MembershipCard card;

        /**
        * This constructor is used to initalise the first four attributes.
        * @param  fName  The member's first name.
        * @param  lName  The member's last name.
        * @param  uName  The member's user name.
        * @param  pWord  The member's password.
        */
```

```
public Member( String fName, String lName, String uName, String pWord ) {
        firstName = fName;
        lastName = lName;
        userName = uName;
        password = pWord;

} // End of constructor.

/**
* Accessor for the attribute firstName.
* @return  firstName  The value of the attribute firstName.
*/
public String getFirstName( ) {
        return firstName;
}  // End of definition of getFirstName.

/**
* Accessor for the attribute lstName.
* @return  lastName  The value of the attribute lastName.
*/
public String getLastName( ) {
        return lastName;
}  // End of definition of getLastName.

/**
* Accessor for the attribute userName.
* @return  userName  The value of the attribute userName.
*/
public String getUserName( ) {
        return username;
}  // End of definition of getUserName.

/**
* Accessor for the attribute password.
* @return  password  The value of the attribute password.
*/
public String getPassword( ) {
        return password;
}  // End of definition of getPassword.

/**
* Accrssor for the attribute membershipNumber.
* @return  membershipNumber  The value of the attribute
* membershipNumber.
*/
```

```
        public int getMembershipNumber( ) {
                return membershipNumber;
        }  // End of definition of getMembershipNumber.

        /**
        * Accessor for the attribute noOfCards.
        * @return  noOfCards  The value of the attribute noOfCards.
        */
        public String getNoOfCards( ) {
                return noOfCards;
        }  // End of definition of getNoOfCards.

        /**
        * Mutator for the attribute card.
        * @param  card  The member's membership card.
        */
        public void setCard( MembershipCard card ) {
                this.card = card;
        } // End of setCard.

        /**
        * Accessor for the attribute card.
        * @return  card  The member's membership card.
        */
        public MembershipCard getCard( ) {
                return card;
        } // End of setCard.

}  // End of class definition of Member.
```

Comments in the source code that are placed between /** and */ are known as *documentation comments* and can be used to generate documentation about classes automatically. Comments that are tagged with '@' have a specific meaning in that they are used to refer to elements such as the programme's author, parameters and return values.

The (similarly simplified) class definition of the **MembershipCard** class follows on the next page.

```
/**
* The purpose of the class MembershipCard is …
* @author David M. Etheridge.
* @version 1.0, dated 29 October 2008.
*/
public class MembershipCard {

        private int noOnLoan;
        private int maxOnLoan;

        /**
        * This constructor is used to initalis the maxOnLoan attribute.          *
        * @param  max  The maximum number of items permitted to be on loan
        * against this card.
        */
        public MembershipCard( int max ) {
                maxOnLoan = max;
        } // End of constructor.
```

```
    /**
    * Accessor for the attribute noOnLoan.
    * @return  noOnLoan  The value of the attribute noOnLoan.
    */
    public int getNoOnLoan( ) {
            return noOnLoan;
    }  // End of definition of getNoOnLoan.

    /**
    * Accessor for the attribute maxOnLoan.
    * @return  maxOnLoan  The value of the attribute maxOnLoan.
    */
    public int getMaxOnLoan( ) {
            return maxOnLoan;
    }  // End of definition of getNoOnLoan.

} // End of class definition.
```

Most development environments for Java include a feature that runs the **javadoc** tool that is provided with the **javac** compiler and other java tools in Java's bin directory. When it is executed, the **javadoc** tool scans the tags and generates a set of linked HTML files. A snapshot of part of the documentation for the **MembershipCard** class is shown next.

Class MembershipCard

```
java.lang.Object
  └MembershipCard
```

`public class MembershipCard extends java.lang.Object`
The purpose of the class definition for the class Member is …

Version:

1.0, dated 29 October 2008.

Author:

David M. Etheridge.

Constructor Summary

`MembershipCard(int max)` This constructor is used to initalize the maxOnLoan attribute.	

Method Summary

`int`	`getMaxOnLoan()` Accessor for the attribute maxOnLoan.
`int`	`getNoOnLoan()` Accessor for the attribute noOnLoan.

Constructor Detail

MembershipCard

`public` **`MembershipCard`**`(int max)`

This constructor is used to initalize the maxOnLoan attribute.

Parameters: `max` - The maximum number of items permitted to be on loan against this card.

Method Detail

getMaxOnLoan

`public int` **`getMaxOnLoan`**`()`

Accessor for the attribute maxOnLoan.

Returns: maxOnLoan The value of the attribute maxOnLoan.

---

getNoOnLoan

`public int` **`getNoOnLoan`**`()`

Accessor for the attribute noOnLoan.

Returns: noOnLoan The value of the attribute noOnLoan.

The **javadoc** tool detects tags such as **@aurhor**, **@version**, **@param** and **@return** and generates the relevant HTML file, as seen by comparing the source code and documentation for the **MembershipCard** class shown above. Single-line comments are not detected by the **javadoc** tool; they are, however, an important component of documentation, as is evident by their extensive use in examples presented in previous chapters.

For further details about **javadoc** tags, the reader is referred to the section titled *Tag Conventions* in http://java.sun.com/j2se/javadoc/writingdoccomments/#sourcefiles

Using the **javadoc** tool is straightforward and should always to be sued to produce documentation for all classes written by Java developers as an integral part of the development process.

1.2 Documentation in the Java Application Programming Interface

The opening page of the version of the API (stored on the author's computer at the time of writing) is shown in the next screen shot.

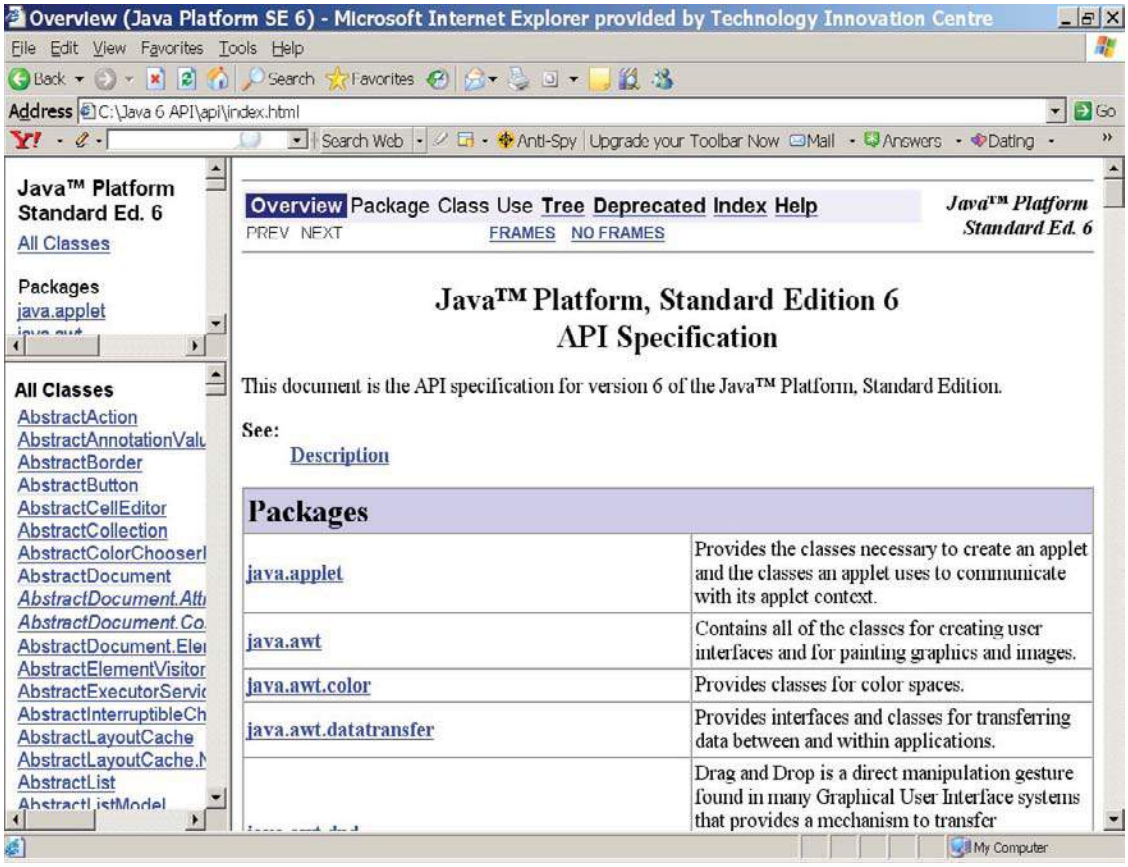


Figure 1.5 The Java API

Part of the section for the **String** class is shown below.

java.lang

Class String

java.lang.Object
  └ **java.lang.String**

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String extends Object
implements Serializable, Comparable<String>, CharSequence
```

Constructor Summary

String() Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
… other constructors follow but are not shown

Method Summary

char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
	... other methods follow but are not shown

Clicking on any of the constructors or methods reveals the details about that member. For example, clicking on the compareTo method of the String class displays the following page.

`compareTo`

`public int` **`compareTo`**`(String anotherString)`

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns `0` exactly when the `equals(Object)` method would return `true`.

Specified by:
`compareTo` in interface `Comparable`<`String`>
Parameters:
`anotherString` - the `String` to be compared.
Returns:
the value `0` if the argument string is equal to this string; a value less than `0` if this string is lexicographically less than the string argument; and a value greater than `0` if this string is lexicographically greater than the string argument.

The principal purpose of the illustrations in this section is to show that the documentation provided by the API is organised in an identical fashion to that produced by the **`javadoc`** tool for developer-written classes, as exemplified in Section 1.1.1. This gives rise to consistently-structured documentation for the infinitude of Java classes used and written by the worldwide Java development community.

While the examples in Section 1.1.1 are relatively straightforward in terms of the logic of Java source code, they are included to illustrate the essential principles and purpose of providing single-line and block documentation as an integral part of writing class definitions.

The next chapter returns to the Java language itself and explores how flow of control is managed in blocks of Java source code.