Chapter 2
Basics of Object-Oriented Programming

In the last chapter, we saw that the fundamental program structure in an object-oriented program is the object. We also outlined the concept of a class, which is similar to ADTs in that it can be used to create objects of types that are not directly supported by language.

In this chapter, we describe in detail how to construct a class. We will use the programming language Java (as we will do throughout the book). We will introduce the Unified Modelling Language (UML), which is a notation for describing the design of object-oriented systems. We also discuss interfaces, a concept that helps us specify program requirements and demonstrate its uses.

2.1 The Basics

To understand the notion of objects and classes, we start with an analogy. When a car manufacturer decides to build a new car, considerable effort is expended in a variety of activities before the first car is rolled out of the assembly lines. These include:

- Identification of the user community for the car and assessment of the user's needs. For this, the manufacturer may form a team.
- After assessing the requirements, the team may be expanded to include automobile engineers and other specialists who come up with a preliminary design.
- A variety of methods may be used to assess and refine the initial design (the team may have experience in building a similar vehicle): prototypes may be built, simulations and mathematical analysis may be performed.

Perhaps after months of such activity, the design process is completed. Another step that needs to be performed is the building of the plant where the car will be produced. The assembly line has to be set up and people hired.

After such steps, the company is ready to produce cars. The design is now reused many times in manufacture. Of course, the design may have to be fine-tuned during the process based on the company's observations and user feedback.

The development of software systems often follows a similar pattern. User needs have to be assessed, a design has to be made, and then the product has to be built.

From the standpoint of object-oriented systems, a different aspect of the car manufacturing process is important. The design of a certain type of car will call for specific types of engine, transmission, brake system, and so on, and each of these parts in itself has its own design (blue print), production plants, etc. In other words, the company follows the same philosophy in the manufacture of the individual parts as it does in the production of the car. Of course, some parts may be bought from manufacturers, but they in turn follow the same approach. Since the design activity is costly, a manufacturer reuses the design to manufacture the parts or the cars.

The above approach can be compared with the design of object-oriented systems which are composed of many objects that interact with each other. Often, these objects represent real-life players and their interactions represent real-life interactions. Just as design of a car is a collection of the individual designs of its parts and a design of the interaction of these parts, the design of an object-oriented system consists of designs of its constituent parts and their interactions.

For instance, a banking system could have a set of objects that represent customers, another set of objects that stand for accounts, and a third set of objects that correspond to loans. When a customer actually makes a deposit into her account in real life, the system acts on the corresponding account object to mimic the deposit in software. When a customer takes out a loan, a new loan object is created and connected to the customer object; when a payment is made on the loan, the system acts on the corresponding loan object.

Obviously, these objects have to be somehow created. When a new customer enters the system, we should be able to create a new customer object in software. This software entity, the customer object, should have all of the relevant features of the real-life customer. For example, it should be possible to associate the name and address of the customer with this object; however, customer's attributes that are not relevant to the bank will not be represented in software. As an example, it is difficult to imagine a bank being interested in whether a customer is right-handed; therefore, the software system will not have this attribute.

Definition 2.1.1 An attribute is a property that we associate with an object; it serves to describe the object and holds some value that is required for processing.

The class mechanism in object-oriented languages provides a way to create such objects. A class is a design that can be reused any number of times to create objects. For example, consider an object-oriented system for a university. There are student objects, instructor objects, staff member objects, and so on. Before such objects are created, we create classes that serve as blue-prints for students, instructors, staff members, and courses as follows:

```
public class Student {
  // code to implement a single student
}

public class Instructor {
  // code to implement a single instructor
}

public class StaffMember {
  // code to implement a single staff member
}

public class Course {
  // code to implement a single course
}
```

The above definitions show how to create four classes, without giving any details. (We should put in the details where we have given comments.) The token `class` is a keyword that says that we are creating a class and that the following token is the name of the class. We have thus created four classes `Student`, `Instructor`, `StaffMember`, and `Course`. The left-curly bracket (`{`) signifies the beginning of the definition of the class and the corresponding right-curly bracket (`}`) ends the definition. The token `public` is another keyword that makes the corresponding class available throughout the file system.

Before we see how to put in the details of the class, let us see how to create objects using these classes. The process of creating an object is also called **instantiation.** Each class introduces a new type name. Thus `Student`, `Instructor`, `StaffMember` and `Course` are types that we have introduced.

The following code instantiates a new object of type `Student`.

```
new Student();
```

The `new` operator causes the system to allocate an object of type `Student` with enough storage for storing information about one student. The operator returns the address of the location that contains this object. This address is termed a **reference.**

The above statement may be executed when we have a new student admitted to the university. Once we instantiate a new object, we must store its reference somewhere, so that we can use it later in some appropriate way. For this, we create a variable of type `Student`.

```
Student harry;
```

Notice that the above definition simply says that `harry` is a variable that can store references to objects of type `Student`. Thus, we can write

```
harry = new Student();
```

We cannot write

```
harry = new Instructor();
```

because `harry` is of type `Student`, which has no relationship (as far as the class declarations are concerned) to `Instructor`, which is the type of the object created on the right-hand side of the assignment.

Whenever we instantiate a new object, we must remember the reference to that object somewhere. However, it is not necessary that for every object that we instantiate, we declare a different variable to store its reference. If that were the case, programming would be tedious.

Let us illustrate by giving an analogy. When a student drives to school to take a class, she deals with only a relatively small number of objects: the controls of the car, the road, the nearby vehicles (and sometimes their occupants, although not always politely), and traffic signals and signs. (Some may also deal with a cell phone, which is not a good idea!) There are many other objects that the driver (student) knows about, but is not dealing with them at this time.

Similarly, we keep references to a relatively small number of objects in our programs. When a need arises to access other objects, we use the references we already have to discover them. For instance, suppose we have a reference to a `Student` object. That object may have an attribute that remembers the student's adviser, an `Instructor` object. If it is necessary to find out the adviser of a given student, we can query the corresponding `Student` object to get the `Instructor` object. A single `Instructor` object may have attributes that remember all the advisees of the corresponding instructor.

2.2 Implementing Classes

In this section we give some of the basics of creating classes. Let us focus on the `Student` class that we initially coded as

```
public class Student {
  // code to implement a single student
}
```

We certainly would like the ability to give a student a name: given a student object, we should be able to specify that the student's name is `"Tom"` or `"Jane"`, or, in general, some string. This is sometimes referred to as a **behaviour** of the object. We can think of student objects having the behaviour that they respond to assigning a name.

For this purpose, we modify the code as below.

```
public class Student {
  // code for doing other things
  public void setName(String studentName) {
    // code to remember the name
  }
}
```

The code that we added is called a method. The method's name is `setName`. A method is like a procedure or function in imperative programming in that it is a unit of code that is not activated until it is invoked. Again, as in the case of procedures and functions, methods accept parameters (separated by commas in Java). Each parameter states the type of the parameter expected. A method may return nothing (as is the case here) or return an object or a value of a primitive type. Here we have put `void` in front of the method name meaning that the method returns nothing. The left and right curly brackets begin and end the code that defines the method.

Unlike functions and procedures, methods are usually invoked through objects. The `setName` method is defined within the class `Student` and is invoked on objects of type `Student`.

```
Student aStudent = new Student();
aStudent.setName("Ron");
```

The method `setName()` is invoked on that object referred to by `aStudent`. Intuitively, the code within that method must store the name somewhere. Remember that every object is allocated its own storage. This piece of storage must include space for remembering the name of the student.

We embellish the code as below.

```
public class Student {
  private String name;
  public void setName(String studentName) {
    name = studentName;
  }
  public String getName() {
    return name;
  }
}
```

Inside the class we have defined the variable `name` of type `String`. It is called a **field**.

Definition 2.2.1 A field is a variable defined directly within a class and corresponds to an attribute. Every instance of the object will have storage for the field.

Let us examine the code within the method `setName`. It takes in one parameter, `studentName`, and assigns the value in that String object to the field `name`.

It is important to understand how Java uses the `name` field. Every object of type `Student` has a field called `name`. We invoked the method `setName()` on the object referred to by `aStudent`. Since `aStudent` has the field `name` and we invoked the method on `aStudent`, the reference to `name` within the method will act on the `name` field of `aStudent`.

The `getName()` method retrieves the contents of the `name` field and returns it. To illustrate this further, consider two objects of type `Student`.

```
Student student1 = new Student();
Student student2 = new Student();
student1.setName("John");
student2.setName("Mary");
System.out.println(student1.getName());
System.out.println(student2.getName());
```

Members (fields and methods for now) of a class can be accessed by writing

```
<object-reference>.<member-name>
```

The object referred to by `student1` has its `name` field set to "John," whereas the object referred to by `student2` has its `name` field set to "Mary." The field `name` in the code

```
name = studentName;
```

refers to different objects in different instantiations and thus different instances of fields.

Let us write a complete program using the above code.

```
public class Student {
  // code
  private String name;
  public void setName(String studentName) {
    name = studentName;
  }
  public String getName() {
    return name;
  }
  public static void main(String[] s) {
    Student student1 = new Student();
    Student student2 = new Student();
    student1.setName("John");
    student2.setName("Mary");
    System.out.println(student1.getName());
    System.out.println(student2.getName());
  }
}
```

The keyword `public` in front of the method `setName()` makes the method available wherever the object is available. But what about the keyword `private` in front of the field `name`? It signifies that this variable can be accessed only from code within the class `Student`. Since the line

```
name = studentName;
```

is within the class, the compiler allows it. However, if we write

```
Student someStudent = new Student();
someStudent.name = "Mary";
```

outside the class, the compiler will generate a syntax error.

As a general rule, fields are often defined with the `private` access specifier and methods are usually made public. The general idea is that fields denote the state of the object and that the state can be changed only by interacting through pre-defined methods which denote the behaviour of the object. Usually, this helps preserve data integrity.

In the current example though, it is hard to argue that data integrity consideration plays a role in making `name` private because all that the method `setName` does is change the name field. However, if we wanted to do some checks before actually changing a student's name (which should not happen that often), this gives us a way to do it. If we had kept `name` public and others coded to directly access the field, making the field private later would break their code.

For a more justified use of private, consider the grade point average (GPA) of a student. Clearly, we need to keep track of the GPA and need a field for it. GPA is not something that is changed arbitrarily: it changes when a student gets a grade for a course. So making it public could lead to integrity problems because the field can be inadvertently changed by bad code written outside. Thus, we code as follows.

```
public class Student {
  // fields to store the classes the student has registered for.
  private String name;
  private double gpa;
  public void setName(String studentName) {
    name = studentName;
  }
  public void addCourse(Course newCourse) {
    // code to store a ref to newCourse in the Student object.
  }
  private void computeGPA() {
    // code to access the stored courses, compute and set the gpa
  }
  public double getGPA() {
    return gpa;
  }
  public void assignGrade(Course aCourse, char newGrade) {
    // code to assign newGrade to aCourse
    computeGPA();
  }
}
```

We now write code to utilise the above idea.

```
Student aStudent = new Student();
Course aCourse = new Course();
aStudent.addCourse(aCourse);
aStudent.assignGrade(aCourse, 'B');
System.out.println(aStudent.getGPA());
```

The above code creates a `Student` object and a `Course` object. It calls the `addCourse` method on the student, to add the course to the collection of courses taken by the student, and then calls `assignGrade`. Note the two parameters: `aCourse` and `'B'`. The implied meaning is that the student has completed the

course (aCourse) with a grade of 'B'. The code in the method should then compute the new GPA for the student using the information presumably in the course (such as number of credits) and the number of points for a grade of 'B'.

2.2.1 Constructors

The Student class has a method for setting the name of a student. Here we set the name of the student after creating the object. This is somewhat unnatural. Since every student has a name, when we create a student object, we probably know the student's name as well. It would be convenient to store the student's name in the object as we create the student object.

To see where we are headed, consider the following declarations of variables of primitive data types.

```
int counter = 0;
double final PI = 3.14;
```

Both declarations store values into the variables as the variables are created. On the other hand, the Student object, when created, has a zero in every bit of every field.

Java and other object-oriented languages allow the initialisation of fields by using what are called **constructors.**

Definition 2.2.2 A constructor is like a method in that it can have an access specifier (like public or private), a name, parameters, and executable code. However, constructors have the following differences or special features.

1. Constructors cannot have a return type: not even void.
2. Constructors have the same name as the class in which they are defined.
3. Constructors are called when the object is created.

For the class Student we can write the following constructor.

```
public Student(String studentName) {
  name = studentName;
}
```

The syntax is similar to that of methods, but there is no return type. However, it has a parameter, an access specifier of public, and a body with executable code. If needed, one could put local variables as well inside constructors.

Let us rewrite the Student class with this constructor and a few other modifications.

```
public class Student {
  private String name;
  private String address;
  private double gpa;
```

```
   public Student(String studentName) {
      name = studentName;
   }
   public void setName(String studentName) {
     name = studentName;
   }
   public void setAddress(String studentAddress) {
     address = studentAddress;
   }
   public String getName() {
     return name;
   }
   public String getAddress() {
     return address;
   }
   public double getGpa() {
     return gpa;
   }
   public void computeGPA(Course newCourse, char grade) {
     // use the grade and course to update gpa
   }
}
```

We now maintain the address of the student and provide methods to set and get the name and the address.

With the above constructor, an object is created as below.

```
Student aStudent = new Student("John");
```

When the above statement is executed, the constructor is called with the given parameter, "John." This gets stored in the name field of the object.

In previous versions of the Student class, we did not have a constructor. In such cases where we do not have an explicit constructor, the system inserts a constructor with no arguments. Once we insert our own constructor, the system removes this default, no-argument constructor.

As a result, it is important to note that the following is no longer legal because there is no constructor with no arguments.

```
Student aStudent = new Student();
```

A class can have any number of constructors. They should all have different signatures: that is, they should differ in the way they expect parameters. The following adds two more constructors to the Student class.

```
public class Student {
  private String name;
  private String address;
  private double gpa;
  public Student(String studentName) {
     name = studentName;
  }
  public Student(String studentName, String studentAddress) {
```

```
      name = studentName;
      address = studentAddress;
  }
  public Student() {
  }
  public void setName(String studentName) {
    name = studentName;
  }
  public void setAddress(String studentAddress) {
    address = studentAddress;
  }
  public String getName() {
    return name;
  }
  public String getAddress() {
    return address;
  }
  public double getGpa() {
    return gpa;
  }
  public void computeGPA(Course newCourse, char grade) {
      // use the grade and course to update gpa
  }
}
```

Notice that all constructors have the same name, which is the name of the class. One of the new constructors accepts the name and address of the student and stores it in the appropriate fields of the object. The other constructor accepts no arguments and does nothing: as a result, the name and address fields of the object are `null`.

2.2.2 Printing an Object

Suppose we want to print an object. We might try

```
System.out.println(student);
```

where `student` is a reference of type `Student`.

The statement, however, will not produce anything very useful for someone expecting to see the name and address of the student. For objects, unless the programmer has provided specific code, Java always prints the name of the class of which the object is an instance, followed by the @ symbol and a value, which is the unsigned hexadecimal representation of the hash code of the object. It does not make any assumptions on the fields to be printed; it prints none of them!

This problem is solved by putting a method called `toString()` in the class. This method contains code that tells Java how to convert the object to a String.

```
public String toString() {
  // return a string
}
```

Whenever an object is to be converted to a String, Java calls the `toString` method on the object just as any other method. The method call `System.out.println()` attempts to convert its arguments to the string form. So it calls the `toString()` method.

We can complete the `toString` method for the Student class as below.

```
public String toString() {
  return "Name " + name + " Address " + address + " GPA " + gpa;
}
```

It is good practice to put the `toString` method in every class and return an appropriate string. Sometimes, the method may get slightly more involved than the simple method we have above; for instance, we may wish to print the elements of an array that the object maintains, in which case a loop to concatenate the elements is in order.

2.2.3 Static Members

So far, all members of a class were accessed using the syntax

```
<object_reference>.<member_name>
```

This is quite logical because we wanted to act on specific objects. Every `Student` object, for example, has its own `name`, `gpa`, and `address` fields. If we did not specify the object and merely specified the field/method, the specification would be incomplete.

Sometimes, we need fields that are common to all instances of an object. In other words, such fields have exactly one instance and this instance is shared by all instances of the class. Such fields are called **static** fields. In contrast, fields maintained separately for each object are called **instance** fields.

Let us turn to an example. Most universities usually have the rule that students not maintaining a certain minimum GPA will be put on academic probation. Let us assume that this minimum standard is the same for all students. Once in a while, a university may decide that this minimum standard be raised or lowered. (Grade inflation can be a problem!)

We would like to introduce a field for keeping track of this minimum GPA. Since the value has to be the same for all students, it is unnecessary to maintain a separate field for each student object. In fact, it is risky to keep a separate field for each object: since every instance of the field has to be given the same value, special effort will have to be made to update all copies of the field whenever we decide to change its value. This can give rise to integrity problems. It is also quite inefficient.

Suppose we decide to call this new field, `minimumGPA`, and make its type `double`. We define the variable as below.

```
private static double minimumGPA;
```

The specifier `static` means that there will be just one instance of the field `minimumGPA`; The field will be created as soon as the class is loaded by the system. Note that there does not have to be any objects for this field to exist. This instance will be shared by all instances of the class.

Suppose we need to modify this field occasionally and that we also want a method that tells us what its value is. We typically write what are called **static methods** for doing the job.

```
public static void setMinimumGPA(double newMinimum) {
  minimumGPA = newMinimum;
}
public static double getMinimumGPA() {
  return minimumGPA;
}
```

The keyword `static` specifies that the method can be executed without using an object. The method is called as below.

```
<class_Name>.<method_name>
```

For example,

```
Student.setMinimumGPA(2.0);
System.out.println("Minimum GPA requirement is "
+ Student.getMinimumGPA());
```

Methods and fields with the keyword `static` in front of them are usually called **static methods** and **static fields** respectively.

It is instructive to see, in the above case, why we want the two methods to be static. Suppose they were instance methods. Then they have to be called using an object as in the following example.

```
Student student1 = new Student("John");
student1.setMinimumGPA(2.0);
```

While this is technically correct, it has the following disadvantages:

1. It requires that we create an object and use that object to modify a static field. This goes against the spirit of static members; they should be accessible even if there are no objects.
2. Someone reading the above fragment may be lead to believe that `setMinimum GPA()` is used to modify an instance field.

On the other hand, a static method cannot access any instance fields or methods. It is easy to see why. A static method may be accessed without using any objects at all. Therefore, what object should the method use to access the member? In fact, there may not be any objects created yet when the static method is in use.

2.3 Programming with Multiple Classes

Even the simplest object-oriented application system will have multiple classes that are related. For the university system we discussed earlier in this chapter, we identified and wrote the skeletons of four classes: `Student`, `Instructor`, `StaffMember`, and `Course`. In this section, we look at how to structure the classes for such cases.

Let us consider the `Course` class. A course exists in the school catalog, with a name, course id, brief description and number of credits. Here is a possible definition.

```
public class Course {
  private String id;
  private String name;
  private int numberofCredits;
  private String description;
  public Course(String courseId, courseName) {
    id = courseId;
    name = courseName;
  }
  public void setNumberOfCredits(int credits) {
    numberOfCredits = credits;
  }
  public void setDescription(String courseDescription) {
    description = courseDescription;
  }
  public String getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public int getNumberOfCredits() {
    return numberOfCredits;
  }
  public String getDescription() {
    return description;
  }
}
```

A department selects from the catalog a number of courses to offer every semester. A section is a course offered in a certain semester, held in a certain place on certain days at certain times. (We will not worry about the instructor for the class, capacity, etc.) Let us create a class for this.

We will use `String` objects for storing the place, days, time, and semester. Thus, we have three fields named `place`, `daysAndTimes`, and `semester` with the obvious semantics.

Clearly, this is inadequate: the class does not hold the name and other details of the course. But it is redundant to have fields for these because the information is available in the corresponding `Course` object. What is required is a field that remembers the corresponding course. We can do this by having the following field declaration.

```
private Course course;
```

When the `Section` instance is created, this field can be initialised.

```
public class Section {
  private String semester;
  private String place;
  private String daysAndTimes;
  private Course course;
  public Section(Course theCourse, String theSemester,
                 String thePlace, String theDaysAndTimes) {
    course = theCourse;
    place = thePlace;
    daysAndTimes = theDaysAndTimes;
    semester = theSemester;
  }
  public String getPlace() {
    return place;
  }
  public String getDaysAndTimes() {
    return daysAndTimes;
  }
  public String getSemester() {
    return semester;
  }
  public Course getCourse() {
    return course;
  }
  public void setPlace(String newPlace) {
    place = newPlace;
  }
  public void setDaysAndTimes(String newDaysAndTimes) {
    daysAndTimes = newDaysAndTimes;
  }
}
```

Where do we create an instance of `Section`? One possibility is to do this in `Course`. Let us assume that we add a new method named `createSection` in `Course`, which accepts the semester, the place, days, and time as parameters and returns an instance of a new `Section` object for the course. We will then use it as follows.

```
Course cs350 = new Course("CS 350", "Data Structures");
Section cs350Section1 = cs350.createSection("Fall 2004",
                                "Lecture Hall 12", "T H 1-2:15");
Section cs350Section2 = cs350.createSection("Fall 2004",
                                "Lecture Hall 25", "'M W F 10-10:50");
```

Let us get to the task of coding the `createSection` method. It looks like the following:

```
public Section createSection(String semester, String place, String time) {
  return new Section(/* parameters */);
}
```

How do we invoke the constructor of `Section` from the `createSection` method? The problem is that although we do have references to the semester, place,

and days and times available in the parameters of this method, we need a reference
to the `Course` object itself. This is not an explicit parameter to the method, but the
`Course` object on which the `createSection` method is invoked is indeed the
reference we need! Here the language comes to our aid. In the `createSection`
method, the reference to the object that was used in its invocation is available via a
special keyword called `this`.

In general, assume that we have a class `C` with a method `m` in it as shown below.
Also shown is another class `C2`, which has a method named `m2` that requires an
object of type `C` as its only parameter.

```
public class C {
  public void m() {
    // this refers to the object on whom m is being invoked
  }
}

public class C2 {
  public void m2(C aC) {
    // code
  }
}
```

Suppose that we create an instance of `C` from the outside and invoke `m` as below.

```
C c1 = new C();
c1.m();
```

This is depicted in Fig. 2.1. The reference `c1` points to an instance of `C`. Suppose the
method `m` contained the following code:

```
public void m(){
    C2 c2 = new C2();
    c2.m2(this);
}
```

In the above, `this` is a reference that points to the same object as `c1`. In summary,
an object can refer to itself by using the keyword `this`.

Fig. 2.1 The notion of
`this`

c1

this

An instance of C
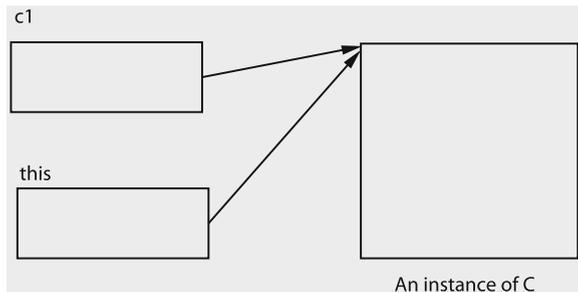
Continuing with the example of courses and their sections, we can code the `createSection` method as below.

```
public Section createSection(String semester, String place, String time) {
  return new Section(this, semester, place, time);
}
```

The keyword `this` obtains the reference to the course object and is passed to the constructor of `Section`.

In addition to passing a reference to itself to methods, we can use `this` to obtain the fields of the object, which come in handy for resolving conflicts. For example,

```
class Section {
    private String place;
    public void setPlace(String place) {
        this.place = place;
    }
}
```

The identifier `place` on right hand side of the assignment refers to the formal parameter; on the left hand side it is prefixed by `this` and is therefore a reference to the private field.

2.4 Interfaces

We design classes based on specifications. These specifications could be written in English and augmented with diagrams, but a compiler cannot read such documents and ensure that the class meets the specifications.

An interface is one way of partially specifying our requirements. Suppose we need to create a list of all students in our university. Let us say that we should be able to add a student, remove a student, and print all students in the list. We can specify the syntax for the methods by creating an interface as given below.

```
public interface StudentList {
  public void add(Student student);
  public void delete(String name);
  public void print();
}
```

Notice that the syntax of the first line resembles the syntax for a class with the keyword `class` replaced by the keyword `interface`. We have *specified* three methods: `add` with a single parameter of type `Student`; `delete` with the name of the student as a parameter, and `print` with no parameters. Notice that we haven't given a body for the methods; there is a semicolon immediately after the right parenthesis that ends the parameters.

Let us see how to utilise the above entity. We can now create a class that implements the above three operations as below.

```
public class StudentLinkedList implements StudentList {
  // fields for maintaining a linked list
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
  // other methods
}
```

The first line states that we are creating a new class named `StudentLinkedList`. The words `implements  StudentList` mean that this class will have all of the methods of the interface `StudentList`. It is a syntax error if the class did not implement the three methods because it has claimed that it implements them.

Just as a class introduces a new type, an interface also creates a new type. In the above example, `StudentList` and `StudentLinkedList` are both types. All instances of the `StudentLinkedList` class are also of type `StudentList`.

We can thus write

```
StudentList students;
students = new StudentLinkedList();
// example of code that uses StudentList;
Student s1 = new Student(/* parameters */);
students.add(s1);
s1 = new Student(/* parameters */);
students.add(s1);
students.print();
```

We created an instance of the `StudentLinkedList` class and stored a reference to it in `students`, which is of type `StudentList`. We can invoke the three methods of the interface (and of the class) via this variable.

Part of these probably seems like wasted effort. Although at this time we cannot discuss all the benefits of using interfaces, let us discuss one: In the above, pay special attention to the following facts:

1. The class `StudentLinkedList` implements the interface `StudentList`. So variables of type `StudentLinkedList` are also of type `StudentList`.
2. We declared `students` as of type `StudentList` and *not* `StudentLinked List`.
3. We restricted ourselves to using the methods of the interface `StudentList`.

Next, assume that we find that the class `StudentLinkedList` is not satisfacatory: perhaps it is not efficient enough. We would like to try and create a new class `StudentArrayList` which uses arrays rather than a linked implementation.

```
public class StudentArrayList implements StudentList {
  // fields for maintaining an array-based list
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
}
```

Now, we can rewrite the code that manipulates StudentList as below.

```
StudentList students;
students = new StudentArrayList();
// code that uses StudentList;
```

The only change that we need to make in our code for using the list is the one
that creates the StudentList object. Since we restricted ourselves to using the
methods of StudentList in the rest of the code (as opposed to using methods
or fields unique to the class StudentLinkedList), we do not need to change
anything else. This makes maintenance easier.

It is instructive to complete the code for StudentLinkedList and Student
ArrayList.

2.4.1 Implementation of StudentLinkedList

A linked list consists of nodes each of which stores the address of the next. We thus
write the following class.

```
public class StudentNode {
  private Student data;
  private StudentNode next;
  public StudentNode(Student student, StudentNode initialLink) {
    this.data = student;
    next = initialLink;
  }
  public Student getData() {
    return data;
  }
  public void setData(Student student) {
    this.data = student;
  }
  public StudentNode getNext() {
    return next;
  }
  public void setNext(StudentNode node) {
    next = node;
  }
}
```

This class will be needed in `StudentLinkedList` only. Therefore, we can use what are called **inner classes** in Java. An inner class is a class enclosed within another class. Thus, we write

```java
public class StudentLinkedList implements StudentList {
  private StudentNode head;
  private class StudentNode {
    private Student data;
    private StudentNode next;
    public StudentNode(Student student, StudentNode initialLink) {
      this.data = student;
      next = initialLink;
    }
    public Student getData() {
      return data;
    }
    public void setData(Student student) {
      this.data = student;
    }
    public StudentNode getNext() {
      return next;
    }
    public void setNext(StudentNode node) {
      next = node;
    }
  }
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
}
```

The inner class `StudentNode` is now declared as private, so that it cannot be used from code outside of the class.

Let us code the `add` method.

```java
public void add(Student student) {
  head = new StudentNode(student, head);
}
```

The code creates a new `StudentNode` and puts it at the front of the list.

Next, we code the `print` method.

```java
public void print() {
  System.out.print("List: ");
  for (StudentNode temp = head; temp != null; temp = temp.getNext()) {
    System.out.print(temp.getData() + " ");
  }
  System.out.println();
}
```

The code starts at the front of the list, extracts the data in the corresponding node and prints that data. Printing ends when the node it points to is `null`; that is, it doesn't exist. Assuming that the `Student` class has a proper `toString()` method, we will get the name, address and GPA of each student printed.

Finally, we code the method to delete a student. We will need to look at each `Student` object and see if the `name` field matches the given name. How do we do this comparison? Suppose `temp` is a variable that refers to a `Student` object. The call `temp.getData()` retrieves the `Student` object, and `temp.getData().getName()` gets the name of the student. Consider the following comparison:

```
temp.getData().getName() ==  studentName
```

Both sides of the equality comparison generate a reference. The system simply compares these references and the expression is true if and only if the two are the same. In general, this is not a correct comparison.

When we need to compare two objects, say, `object1` and `object2`, we should write

```
object1.equals(object2)
```

which returns a logical value which is true if the two objects are equal and false otherwise.

The code for the delete method is given below.

```
public void delete(String studentName) {
  if (head == null) {
     return;
  }
  if (head.getData().getName().equals(studentName)) {
     head = head.getNext();
  } else {
     for (StudentNode temp = head.getNext(), previous = head;
                       temp != null; temp = temp.getNext()) {
       if (temp.getData().getName().equals(studentName)) {
          previous.setNext(temp.getNext());
          return;
       }
     }
  }
}
```

The code first checks if the list is empty; if so, there is nothing to do. With an non-empty list, it checks if the name of the student at the front of the list is the same as the name supplied in the parameter. If they match, the `Student` object at the front of the list is deleted from the list by moving the head to the next object (which may not exist, in which case we have a null). If the element at the front of the list is not what we want, execution proceeds to a loop that examines all elements starting at the

second position until the end of the list is reached or the student with the given name is located. The variable `previous` always refers to the object preceding the object referred to by `temp`. Once it is located, the object can be deleted using `previous`.

2.4.2 Array Implementation of Lists

We need to set up an array of Student objects. This is done as follows.

1. Declare a field in the class `StudentArrayList`, which is an array of type `Student`.
2. Allocate an array of the required size. We will allocate storage for as many students as the user wishes; if the user does not specify a number, we will allocate space for a small number, say, 10, of objects. In any case, when this array fills up, we will allocate more.

Therefore, we need two constructors: one that accepts the initial capacity and the other that accepts nothing. The code for the array field and the constructor is given below.

```
public class StudentArrayList implements StudentList {
  private Student[] students;
  private int initialCapacity;
  public StudentArrayList() {
    students = new Student[10];
    initialCapacity = 10;
  }
  public StudentArrayList(int capacity) {
    students = new Student[capacity];
    initialCapacity = capacity;
  }
  // other methods
}
```

Note that the code for the first constructor is a special case of the second constructor. This is undesirable. We should try to reuse the code in the second constructor because it is general enough. Thus, when the user does not supply an initial capacity, we should somehow invoke the second constructor with a value of 10. This reuse can be achieved by rewriting the first constructor as follows:

```
public StudentArrayList() {
  this(10);
}
```

In this case, `this` refers to another constructor of the class. We are specifying a constructor that has a single `int` parameter and invoking it with a parameter value of 10. The net effect would be the same as that of the user writing `new StudentArrayList(10)`.

The use of `this` in the above context should not be confused with the one that is used to refer to the object used in instance methods. Also, note the following aspects.

1. There can be no code before the statement `this()`. In other words, this call should be the very first statement in the constructor.
2. You can have code in the constructor after the call to another constructor.
3. You can call at most one other constructor from a constructor.

We will use the following approach to manage the list. We will have two variables, `first` that gives the index of the first occupied cell, and `count`, the number of objects in the list. When the list is empty, both are 0. When we add an object to the list, we will insert it at `(first + count) %` array size and increment `count`.

```java
public class StudentArrayList implements StudentList {
  private Student[] students;
  private int first;
  private int count;
  private int initialCapacity;
  public StudentArrayList() {
    students = new Student[10];
    initialCapacity = 10;
  }
  public StudentArrayList(int capacity) {
    students = new Student[capacity];
    initialCapacity = capacity;
  }
  public void add(Student student) {
    if (count == students.length) {
      reallocate(count * 2);
    }
    int last = (first + count) % students.length;
    students[last] = student;
    count++;
  }
  public void delete(String name) {
    for (int index = first, counter = 0; counter < count;
              counter++, index = (index + 1) % students.length) {
      if (students[index].getName().equals(name)) {
        students[index] = students[(first + count - 1) % students.length];
        students[(first + count - 1) % students.length] = null;
        count--;
        return;
      }
    }
  }
  public Student get(int index) {
    if (index >= 0 && index < count) {
      return students[index];
    }
    return null;
  }
  public int size() {
    return count;
  }
  public void print() {
    for (int index = first, counter = 0; counter < count;
                    counter++, index = (index + 1)
```

```
                                        % students.length) {
        System.out.println(students[index]);
      }

    }
    public void reallocate(int size) {
      Student[] temp = new Student[size];
      if (first + count >= students.length) {
        int count1 = students.length - first;
        int count2 = count - count1;
        System.arraycopy(students, first, temp, 0, count1);
        System.arraycopy(students, first + count1, temp, count1, count2);
      } else {
        System.arraycopy(students, first, temp, 0, count);
      }
      students = temp;
      first = 0;
    }
}
```

2.5 Abstract Classes

In a way, classes and interfaces represent the extreme ends of a spectrum of possible
implementations. When we write a class, we code every field and method; in other
words, the code is complete in a sense. Interfaces are merely specifications.

Sometimes, we might know the specifications for a class, but might not have the
information needed to implement the class completely. For example, consider the set
of possible shapes that can be drawn on a computer screen. While the set is infinite,
let us consider only three possibilities: triangles, rectangles, and circles. We know
that the set of fields needed to represent each object is different, but there are some
commonalities as well. For example, all shapes have an area.

In such cases, we can implement a class partially using what are called **abstract**
classes. In the case of a shape, we may code

```
public abstract class Shape {
  private double area;
  public abstract void computeArea();
  public double getArea() {
    return area;
  }
  // more fields and methods
}
```

The class is declared as abstract (using the keyword `abstract` prior to the keyword
`class`), which means that the class is incomplete. Since we know that every shape
has an area, we have defined the `double` field `area` and the method `getArea()`
to return the area of the shape. We require that there be a method to compute the
area of a shape, so we have written the method `getArea()`. But since the formula
to compute the area is different for the three possible shapes, we have left out the
implementation and declared the method itself as abstract.

Any class that contains an abstract method must be declared abstract. We cannot create an instance of an abstract class. The utility of an abstract class comes from the fact that it provides a basic implementation that other classes can "extend". This is done using the technique of inheritance, covered in Chap. 3.

2.6 Comparing Objects for Equality

We have seen the need to use the `equals` method to compare two objects. In this section we explore this issue a little more.

Given any two variables of the same primitive type, it is easy for Java to decide whether they are equal: the variables are equal if they have the same value. However, consider a class such as `Student`. It is a user defined class. When do you say that two `Student` objects are equal? Here are some possibilities.

1. The language specifies that two objects are equal if they occupy the same physical storage.
2. The language provides a facility to check whether the corresponding fields of the objects are equal. This is a recursive definition. For example, in the `Student` class, the fields are `name`, `address` and `gpa`. For the `name` field of two objects to be equal, we have to know when two `String` objects are equal. Since `gpa` is a `double`, that field presents no problems.
3. The language leaves the responsibility to the class itself; that is, it lets the class specify when two of its objects are equal.

Java supports both (1) and (3) above. Since a class can specify when another object is equal to an object of its type, we can implement (2) as a special case.

To specify how objects should be compared for equality, we need to write a special method called `equals` which has the following format:

```
public boolean equals(Object someObject) {
   // implement the policy for comparison in this method.
   // return true if and only if this object is equal to someObject
}
```

We are given two objects: `this`, the one on which we invoke `equals()`, and `someObject`, an arbitrary object, which can be of any type. It is enough at this stage to know that `Object` is a special class in Java and every object can be thought of as an instance of this class. The method is free to decide whether `someObject` is equal to `this` in any way it pleases.

For example, let us say that a `Student` object is equal to another object only if that object is a `Student` object, the names are equal and they have the same address. One could definitely argue that the policy is flawed, but that is not our focus. Here is how to implement the equals method.

```
public boolean equals(Object anObject) {
  Student student = (Student) anObject;
  return student.name.equals(name) && student.address.equals(address);
}
```

As explained earlier, the method is placed inside the `Student` class and is invoked as below.

```
Student student1 = new Student("Tom");
student1.setAddress("1 Main Street");
// some other code
Student student2 = new Student("Tom");
student2.setAddress("1 Main Street");
// more code
if (student1.equals(student2)) {
  System.out.println("student1 is the same as student2");
} else {
  System.out.println("student1 is not the same as student2");
}
```

After creating the two `Student` objects with the same name and address, we invoked the `equals` method on `student1` with `student2` as the actual parameter. The first thing that the `equals` method does is cast the incoming object as a `Student` object. The resulting reference can be used to access all of the members of the corresponding `Student` object and, in particular, the `name` and `address` fields.

After the cast, we check if the `name` field of the cast object is equal to the name field of `this`, which in our example is `student1`. Note that we are doing this by invoking the `equals` method on the object `student.name`, which is a `String`; thus, we are invoking the `equals` method of the `String` class. It turns out that the `equals` method of the `String` class returns `true` if and only if every character in one string is equal to the corresponding character of the other string.

The address fields are compared in a similar way. The method returns true if and only if the two fields match.

What happens when you pass an object other than a `Student`, for instance, a `Course` object? This is valid because a `Course` object can also be viewed as of type `Object`. The cast in the `equals` method will fail and the program may crash if this problem is not addressed.

2.7 A Notation for Describing Object-Oriented Systems

We all know that it is important to document systems and programs. In this section, we introduce a notation called **Unified Modeling Language (UML)**, which is the standard for documenting object-oriented systems. Many different ideas had been suggested to document object-oriented systems in the past and the term "Unified" reflects the fact that UML was an attempt to unify these different approaches. Among the ones who contributed to the development of this notation, the efforts of Grady Booch, James Rumbaugh, and Ivor Jacobson deserve special mention. After the

initial notation was developed around 1995, the Object Management Group (OMG) took over the task of developing the notation further in 1997. As the years went by, the language became richer and, naturally, more complex. The current version is UML 2.0.

UML provides a pictorial or graphical notation for documenting the artefacts such as classes, objects and packages that make up an object-oriented system. UML diagrams can be divided into three categories.

1. **Structure diagrams** that show the static architecture of the system irrespective of time. For example, structure diagrams for a university system may include diagrams that depict the design of classes such as Student, Faculty, etc.
2. **Behaviour diagrams** that depict the behaviour of a system or business process.
3. **Interaction diagrams** that show the methods, interactions and activities of the objects. For a university system, a possible behaviour diagram would show how a student registers for a course.

Structure diagrams could be one of the following.

1. **Class diagrams**: They show the classes, their methods and fields.
2. **Composite structure diagrams**: They provide a means for presenting the details of a structural element such as a class. As an example, consider a class that represents a microcomputer system. Each object contains other objects such as CPU, memory, motherboard, etc, which would be shown as parts that make up the microcomputer system itself. The composite structure diagram for such a system would show these parts and exhibit the relationships between them helping the reader understand the details.
3. **Component diagrams**: Components are software entities that satisfy certain functional requirements specified by interfaces. These diagrams show the details of components.
4. **Deployment diagrams**: An object-oriented system consists of a number of executable files sometimes distributed across multiple computing elements. These diagrams show the assignment of executable files on the computing elements and the communication that involves between these entities.
5. **Object diagrams**: They are used to show how objects are related and used at run-time. For instance, in a university system we may show the object corresponding to a specific course and show other objects that represent students who have registered for the course. Since this shows an actual scenario that involves students and a course, it is far less abstract than class diagrams and contributes to a better understanding of the system.
6. **Package diagrams**: Classes may be grouped into packages and packages may reside in other packages. These diagrams show packages and dependencies among them: whether a change in one package may affect other packages.

Each of the six diagrams is a structure diagram. This hierarchy is illustrated in Fig. 2.2 as a tree with nodes representing these six diagrams as children of the Structure diagram node. It turns out that this method of showing a hierarchy is used in UML; so we are using UML notation itself to describe UML!
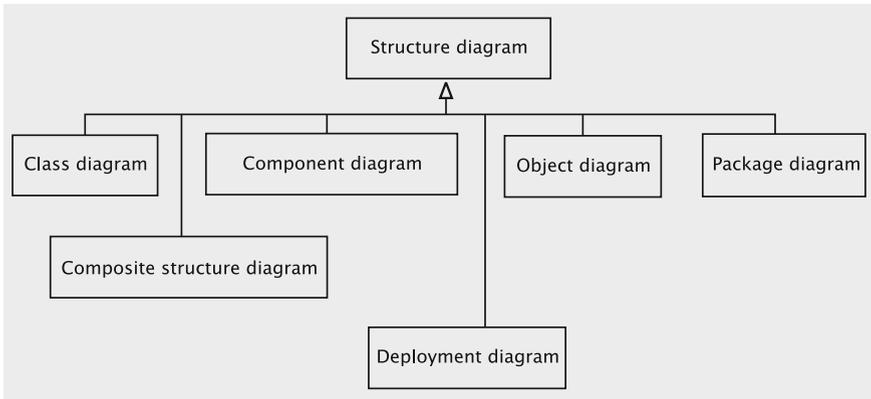
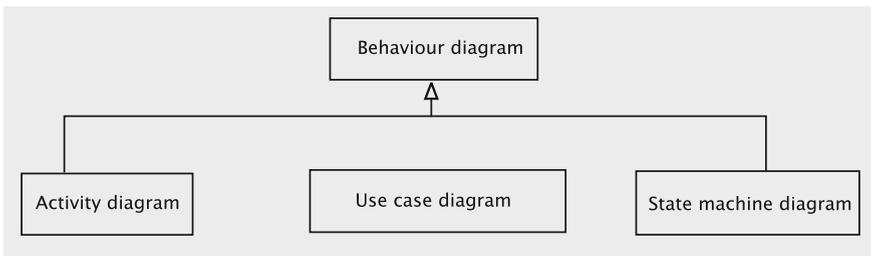**Fig. 2.2** Types of UML structure diagrams

Fig. 2.3 Types of UML behaviour diagrams

Behaviour diagrams can be any of the following (see Fig. 2.3).

1. **Activity diagrams**: This is somewhat like a flowchart in that it shows the sequence of events in an activity. Just as a flowchart, it uses several types of nodes such as actions, decisions, merge points, etc. It accommodates objects with suitable types that depict objects, *object flows*, etc.
2. **Use case diagrams**: A use case is a single unit of some useful work. It involves a user (called an actor) and the system. An example of a use case in a university environment is a student registering for a course. A use case diagram shows the interaction involved in a use case.
3. **State machine diagrams**: It shows the sequence of states that an object goes through during its lifetime, e.g., the software that controls a washer for clothes. Initially, the washer is in the off state. After the soap is put in, the clothes are loaded and the on button pressed, the system goes to a state where it takes in water. In this state the system waits for a signal from the water sensor to indicate that the water has reached the required level. Then the system goes into the wash state where washing takes place. After this the system may go through further states such as rinse and spin and eventually reaches the washed state.
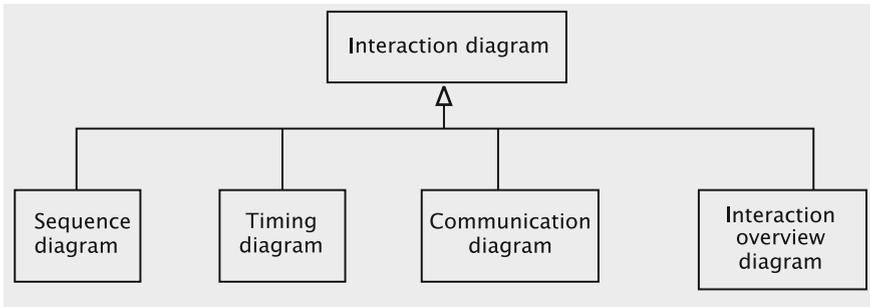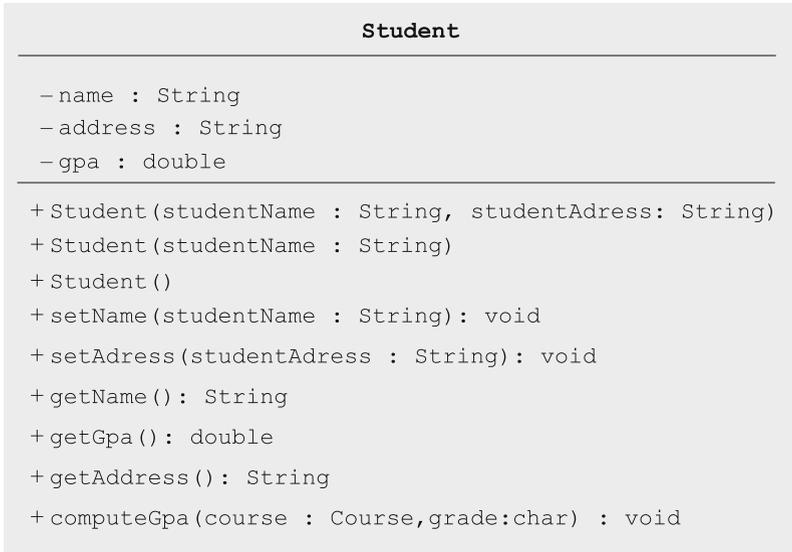
Fig. 2.4 Types of UML interaction diagrams

There are four types of interaction diagrams as shown in Fig. 2.4.

1. **Sequence diagrams**: A sequence diagram is an interaction diagram that details how operations are carried out—what messages are sent and when. Sequence diagrams are organised according to time. Time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.
2. **Timing diagrams**: It shows the change in state of an object over time as the object reacts to events. The horizontal axis shows time and the state changes are noted on the vertical axis. Contrast this with sequence diagrams in which time is in the vertical axis.
3. **Communication diagrams**: A communication diagram essentially serves the same purpose as a sequence diagram. Just as in a sequence diagram, this diagram also has nodes for objects and uses directed lines between objects to indicate message flow and direction. However, unlike sequence diagrams, vertical direction has no relationship with time and message order is shown by numbering the directed lines that represent messages.
   Interactions that involve a large number of objects can be somewhat inconvenient to show using sequence diagrams because they must be arranged horizontally. Since no such restrictions are placed on communication diagrams, they are easier to draw. However, the order of messages can be harder to see in communication diagrams.
4. **Interaction overview diagrams**: An interaction overview diagram shows the high-level control flow in a system. It shows the interactions between interaction diagrams such as sequence diagrams and communication diagrams. Each node in the diagram can be an interaction diagram.

We will see examples of many of these diagrams as we develop concepts in this book. At this time, we show an example of a class diagram.

```
                            Student
   ───────────────────────────────────────────────────────────
   − name : String
   − address : String
   − gpa : double
   ───────────────────────────────────────────────────────────
   + Student(studentName : String, studentAdress: String)
   + Student(studentName : String)
   + Student()
   + setName(studentName : String): void
   + setAdress(studentAdress : String): void
   + getName(): String
   + getGpa(): double
   + getAddress(): String
   + computeGpa(course : Course,grade:char) : void
```

**Fig. 2.5** Example of a class diagram

2.7.1 Class Diagrams

Figure 2.5 is an example of a class diagram. Each class is represented by a box, which is divided into three rectangles. The name of the class is given in the top rectangle. The attributes are shown with their names and their types in the second box. The third box shows the methods with their return types and parameters (names and types). The access specifier for each field and method is given just in front of the field name or method name. A − sign indicates private access, + stands for public access and # (not shown in this example) is used for protected access which we will discuss in Chap. 3.

2.7.2 Use Cases and Use Case Diagrams

A use case describes a certain piece of desired functionality of an application system. It is constructed during the analysis stage. It shows the interaction between an **actor**, which could be a human or a piece of software or hardware and the system. It does *not* specify *how* the system carries out the task.

As an example of a simple use case, let us describe what a simple ATM machine will do. A user may withdraw or deposit money into his bank account using this machine. This functionality is shown in the use case diagram in Fig. 2.6.

Fig. 2.6 Example of a use
case diagram
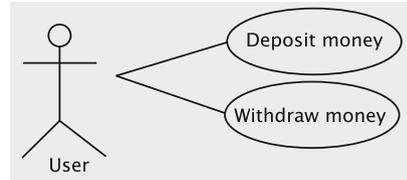
Use cases may be verbally described in a table with two columns: The first column
shows what the actor does and the second column depicts the system's behaviour.

We give below the use case for withdrawing money.

	Action performed by the actor		Responses from the system
1.	Inserts debit card into the 'Insert card' slot		
		2.	Asks for the PIN number
3.	Enters the PIN number		
		4.	Verifies the PIN. If the PIN is invalid, displays an error and goes to Step 8. Otherwise, asks for the amount
5.	Enters the amount		
		6.	Verifies that the amount can be withdrawn. If not, display an error and goes to Step 8. Otherwise, dispenses the amount and updates the balance
7.	Takes the cash		
		8.	Ejects the card
9.	Takes the card		

Notice that the use case specifies the responsibilities of the two entities but does not
show *how* the system processes the request. Throughout the book, we express use
cases in a two-column format as above.

The use case as specified above does not say what the system is supposed to do
in all situations. For example, what should the system do if something other than
a valid ATM card is inserted? Such considerations may result in a more involved
specification. What is specified above is sometimes called the **main flow.**

2.7.3 Sequence Diagrams

One of the major goals of design is to determine the classes and their responsibilities
and one way of progressing toward the above goal is to create sequence diagrams
for each use case we identify in the analysis stage. In such a diagram we break down
the system into a number of objects and decide what each object should accomplish
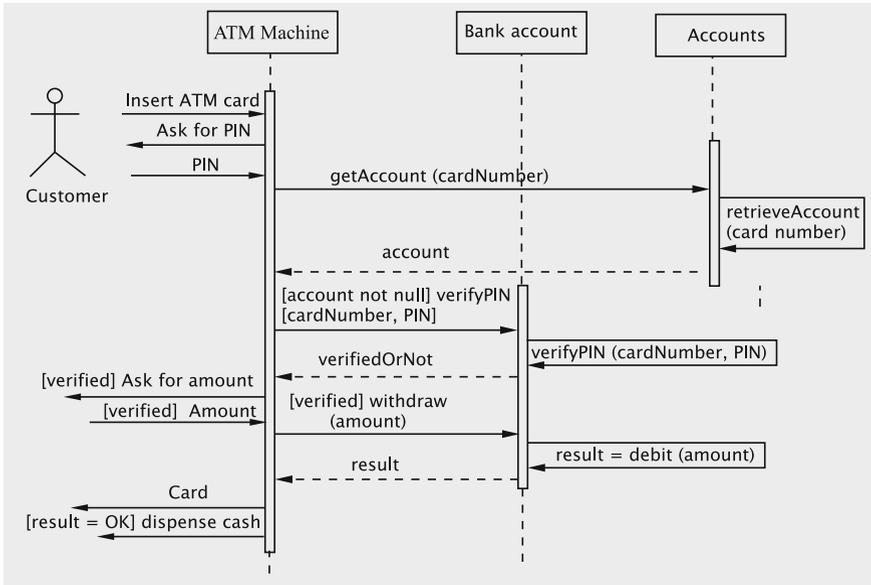in the corresponding use case. That is, we delegate responsibilities.

Fig. 2.7 Example of a simple sequence diagram

We have one column for each entity that plays a role in the use case. The vertical direction represents the flow of time. Horizontal arrows represent functionalities being invoked; the entity at the tail of the arrow invokes the named method on the entity at the head of the arrow.

For example, Fig. 2.7 shows the sequence diagram corresponding to the use case we gave above for withdrawing from an ATM. The rectangles at the top of the diagram represent the customer, the ATM, and two objects that reside in the bank database: `Accounts`, which stores all the account objects and `BankAccount`, which stores account-related information for a single account. For each object, we draw a dashed vertical line, called a **lifeline**, for showing the actions of the object. The long and thin rectangular boxes within these lifelines show when that object is active.

In many use cases, the actor interacts only with the left most entity, which usu-ally represents some kind of interface. These interactions mirror the functionality described in the use case. The first arrow denotes the customer (actor) inserting the debit card into the ATM, which, in turn, asks for the PIN, as shown by the arrow from the ATM to the customer. Notice that the latter line is lower than the line that stands for the card insertion. This is because time increases as we go down in the diagram. The events in the sequence diagram that happen after the customer enters the PIN depend on how the system has been implemented. In our hypothetical exam-ple, we assume that the ATM has to access a central repository (viz., `Accounts`)

Fig. 2.8 An example of association

```
                        ┌──────────┐  1   contains   0...*  ┌─────────────┐
                        │ Accounts │──────────────────────── │ BankAccount │
                        └──────────┘                         └─────────────┘
```

and attempt to retrieve the user's information.[1] If successful, the repository returns a reference to an object (`BankAccount`) representing the user's account, and the ATM then interacts with this object to complete the transaction.

The sequence diagram gives us the specifics of the implementation: the ATM calls the method `getAccount` on the `Accounts` object with the card number as parameter. The `Accounts` object either returns the reference to the appropriate `BankAccount` object corresponding to the card number, or `null` if such an account does not exist. When the `getAccount` method is invoked, the `Accounts` object calls the method `retrieveAccount` to get the `BankAccount` object to which the card number corresponds. Note the self-directed arc on the lifeline of the `Accounts` object, indicating that this computation is carried out locally within `Accounts`. The `getAccount` method invocation and its return are on separate lines, with the return shown by a dotted line.

The ATM then invokes the `verifyPIN` method on the `BankAccount` object to ensure that the PIN is valid. If for some reason the card is not valid, `Accounts` would have returned a `null` reference, in which case further processing is impossible. Therefore, the call to verify the PIN is conditional on reference being non-null. This is indicated in the sequence diagram by writing `[account not null]` along with the method call `verifyPIN`. Such a conditional is called a **guard**.
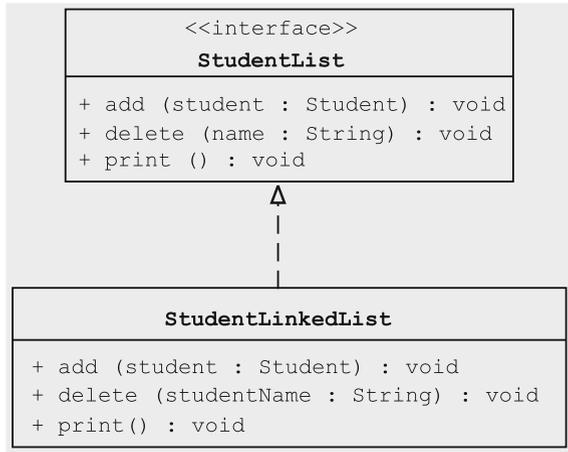
Just as `Accounts` called a method on itself, `BankAccount` calls the method `verifyPIN` to see if the PIN entered by the user is valid. The result, a boolean, is returned and shown on a separate dotted line in the diagram. If the PIN is valid, the ATM asks the user for the amount to be withdrawn. Once again, note the guard associated with this action. After receiving the value (the amount to be withdrawn), the machine sends the message `withdraw` with the amount as parameter to the `BankAccount` object, which verifies whether the amount can be withdrawn by calling the method `debit` on itself. The result is then returned to the ATM, which dispenses cash provided the result is acceptable.

Association

In our example that involved the ATM, `Accounts` and `BankAccount`, the `Accounts` instance contained all of the `BankAcount` objects, each of which could be retrieved by supplying a card number. This relationship can be shown using an association as in Fig. 2.8. Notice the number 1 above the line near the rectangle that represents `Accounts` and 0...* at the right end of the line near BankAccount. They mean that one `Accounts` object may hold references to zero or more BankAccount objects.

---

[1]This may not reflect a real ATM's behaviour, but bear in mind that this is a pedagogical exercise in UML, not banking.

Fig. 2.9 Depicting
interfaces and their
implementation

```
            <<interface>>
            StudentList

+ add (student : Student) : void
+ delete (name : String) : void
+ print () : void
```

```
            StudentLinkedList

+ add (student : Student) : void
+ delete (studentName : String) : void
+ print() : void
```

Interfaces and Their Implementation

Interfaces and their implementation can be depicted in UML as in Fig. 2.9. With the
`StudentList` interface and the class `StudentLinkedList` class that imple-
ments it, we draw one box to represent the interface and another to stand for the
class. The methods are shown in both. The dotted line from the class to the interface
shows that the class implements the interface.

2.8 Discussion and Further Reading

The concept of a class is fundamental to the object-oriented paradigm. As we have
discussed, it is based on the notion of an abstract data type and one can trace its origins
to the Simula programming language. This chapter also discussed some of the UML
notation used for describing classes. In the next chapter we look at how classes
interconnect to form a system, and the use of UML to denote these relationships.

The Java syntax and concepts that we have described in this chapter are quite
similar to the ones in C++; so the reader should have little difficulty getting introduced
to that language. A fundamental difference between Java and C++ is in the availability
of pointers in C++, which can be manipulated using pointer arithmetic in ways that
add considerable flexibility and power to the language. However, pointer arithmetic
and other features in the language also make C++ more challenging to someone new
to this concept.

Since our intention is to cover just enough language features to complete the
implementations, some readers may wish to explore other features of the language.
For those who want an exposure to the numerous features of Java, we suggest *Core
Java* by Cornell and Horstmann [1]. A more gentle and slow exposure to program-

ming in Java can be found in Liang [2]. If syntax and semantics of Java come fairly easy to you but you wish to get more insights into Java usage, you could take a look at Eckel [3].

It is important to realise that the concepts of object-oriented programming we have discussed are based on the Java language. The ideas are somewhat different in languages such as Ruby, which abandons static type checking and allows much more dynamic changes to class structure during execution time. For an introduction to Ruby, see [4].

Projects

1. A consumer group tests products. Create a class named Product with the following fields:

   (a) Model name,
   (b) Manufacturer's name,
   (c) Retail price,
   (d) An overall rating ('A', 'B', 'C', 'D', 'F'),
   (e) A reliability rating (based on consumer survey) that is a double number between 0 and 5,
   (f) The number of customers who contributed to the survey on reliability rating.

   Remember that names must hold a sequence of characters and the retail price may have a fractional part.

   The class must have two constructors:

   (a) The first constructor accepts the model name, the manufacturer name, and the retail price in that order.
   (b) The second constructor accepts the model name and the manufacturer name in that order, and this constructor must effectively use the first constructor.

   Have methods to get every field. Have methods to set the retail price and the overall rating.
   Reliability rating is the average of the reliability ratings by all customers who rated this product. A method called `rateReliability` should be written to input the reliability rating of a customer. This method has a single parameter that takes in the reliability of the product as viewed by a customer. The method must then increment the number of customers who rated the product and update the reliability rating using the following formula.
   New value of reliability rating = (Old value of reliability rating * Old value of number of customers + Reliability rating by this customer) / New value of number of customers.
   For example, suppose that the old value of reliability was 4.5 based on the input from 100 customers. If a new customer gives a reliability rating of 1.0, then the new value of reliability would be

```
    (4.5 * 100 + 1.0) / 101
```

which is 4.465347.

Override the `toString` method appropriately.

2. Write a Java class called `LongInteger` as per the following specifications. Objects of this class store integers that can be as long as 50 digits. The class must have the following constructors and methods.

   (a) `public LongInteger()`: Sets the integer to 0.
   (b) `public LongInteger(int[] otherDigits)`: Sets the integer to the given integer represented by the parameter. A copy of `otherDigits` must be made to prevent accidental changes.
   (c) `public LongInteger(int number)` Sets the integer to the value given in the parameter.
   (d) `public void readIn()`: reads in the integer from the keyboard. You can assume that only digits will be entered.
   (e) `public LongInteger add(int number)` Adds number to the integer represented by this object and returns the result.
   (f) `public LongInteger add(LongInteger number)` Adds `number` to the integer represented by this object and returns the result.
   (g) `public String toString()` returns a `String` representation of the integer.

   Use an array of 50 `ints` to store the digits of the number.

3. Study the interface `Extendable` given below.

```
public interface Extendable {
  public boolean append(char c);
  public boolean append(char[] sequence);
}
```

The method `append(char c)` appends a character to the object (or, more precisely the object's class) that implements this interface. The second version of the method appends all characters in the array to this object. If there is no space in the object to append, the methods return `false`; otherwise they return `true`. Write code for the class `SimpleBuffer` that implements the above interface which has a constructor of the following signature.

```
public SimpleBuffer(int size)
```

The initial size of the array is passed as a parameter.

The class must have two fields: one which stores the `char` array and the other which stores the number of elements actually filled in the array.

This class must also implement the `toString` method to bring back correctly a `String` representation of the `char` array. It should also implement the `equals` method such that two buffers are equal if and only if they contain the same set of characters.

2.9 Exercises

1. Given the following class, write a constructor that has no parameters but uses the given constructor so that x and y are initialised at construction time to 1 and 2 respectively.

```
public class SomeClass {
  private int x;
  private int y;
  public SomeClass(int a, int b) {
    x = a;
    y = b;
  }
  // write a no-argument (no parameters)
  // constructor here, so that x and y are
  // initialised to 1 and 2 respectively.
  // You MUST Utilise the given constructor.
}
```

2. In Sect. 2.3, we had a class called `Course`, which had a method that creates `Section` objects. Modify the two classes so that

   (a) `Course` class maintains the list of all sections.
   (b) `Section` stores the capacity and the number of students enrolled in the class.
   (c) `Course` has a search facility that returns a list of sections that are not full.

3. In Sect. 2.7, we had a discussion on two possible use cases for using an ATM. Develop the use case for depositing money using an ATM machine.
4. Draw the sequence diagram for the use case you developed for Exercise 3.
5. Take a look at the use case and sequence diagram we developed for withdrawing money through an ATM. Design the method `getAccount()` in the class `Accounts`. Does this need interaction between the two classes, `Accounts` and `BankAccount`? If so, what additional methods do you need in `BankAccount`?

References

1. C.S. Horstmann, G. Cornell, *Core Java(TM)*, vol. 1, Fundamentals 8th edn. (Sun Microsystems, California, 2007)
2. Y.D. Liang, *Introduction to Java Programming Comprehensive Version* (Pearson Prentice Hall, New Jersey, 2007)
3. B. Eckel, *Thinking in Java*, 4th edn. (Prentice Hall, New Jersey, 2006)
4. P. Cooper, *Beginning Ruby: From Novice to Professional (Beginning from Novice to Professional)*. (Apress, New York, 2007)