Chapter 3 - Abstraction and Object-Orientation

The development of programming languages over the past three decades has been driven by the developing understanding of what makes a good program. Key to this understanding is the use of abstraction to isolate individual steps in problem solution and organizing those steps into a complete solution.

In this chapter, we examine the concepts that have led to this philosophy of language design to support good programming practice. We will illustrate these concepts through a sequence of programs, all written in Java. Each program in this sequence will demonstrate one important concept through the use of a Java language feature.

3.1 Characteristics of a Good Program

Three important properties of good programs that have had a significant influence on language design are implementation hiding, independence, and reusability. We describe these three properties in this section and then, in the following sections of this chapter, illustrate the ways they impact language design through progressive versions of an example program.

All three properties support the abstraction property, which was discussed in Chapter 1 and will be a recurring theme throughout this text. You will recall that abstraction was the representation of an object by only the relevant information, hiding information that is irrelevant to the task at hand. One very important type of information that is hidden in a good program is the implementation of the object. This is true whether the object is a procedure or data. This property is called implementation hiding.

Why is implementation hiding relevant to good programming? There are two key reasons. First, it enables the programmer to solve the problem by first concentrating on high-level tasks, then breaking those tasks into subtasks, and continuing this process until tasks at the lowest level are easily implemented to carry out a single action. This process, known as stepwise refinement, has proven to be a very effective way to organize the solution of complex problems, and it is made possible by procedural implementation hiding.

A second advantage of implementation hiding is that it enables the programmer to concentrate on one task or data object at a time, abstracting out other components that would complicate or obscure the task at hand.

Another key element for effective programming is independence. By this we mean that as program objects such as procedures and data are divided into smaller implementation units in support of implementation hiding, these units should be as independent as possible from each other.

If the units are independent of each other, each unit can be implemented without any information about the implementations of the other units. Another way of expressing this concept of independence is to say that each unit only needs to know its mission and the way other units will interface with it. Note that this is the other side of the implementation hiding coin. Not only is it important that a unit hide its own implementation details (implementation hiding), it must also have no reliance on the implementation of other units (independence).

The final important characteristic of a good program is that the units used to support abstraction should be reusable. This means that a procedural or data unit must have the flexibility
to be used in a variety of settings. It also means that when new units are required that duplicate a significant portion of units that already exist, the new units can be created without duplicating the components the two units have in common.

The support of these three characteristics of good programs has been the motivation for much of the development of programming languages. To illustrate this, the remainder of this chapter follows the progressive development of a program to solve a simple problem. Each program in this progression illustrates a single step toward the support of good programming characteristics by using a programming language feature.

The programming language Java is used for these programs since it contains all of the supporting language features needed for this illustration. To introduce support for the characteristics one at a time, the early programs will not use the full power of Java and the programs may be written in a manner that would be ill-advised when the full power of the language is brought to bear on the problem.

The Example Problem

The problem used as an illustration in this chapter is a very simple one. The program reads three pieces of data about a student and determines whether the data indicates a student who qualifies to graduate or not.

The three inputs for the student are the student’s name (a String), the student’s grade point average (a double), and the number of credit hours the student has completed (an int). Only valid input is accepted, where a valid grade point average is between 0.0 and 4.0 inclusive and a valid credit hours completed is positive. A student is eligible to graduate if the gpa is greater than or equal to 2.0 and the number of hours completed is greater than or equal to 126.

The program in Listing 3.1 is a straightforward Java solution to this problem using no language abstraction features at all. This is the base program from which the remaining programs will be derived.

Listing 3.1

// Solution of the student graduation problem
// using no language abstraction features

import java.io.*;

public class student_1
{
    // Reader for input data.
    static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

    static String student_name;  // Student’s name
    static double student_gpa;   // Student’s grade point average
    static int    student_hours; // Student’s number of credit hours completed

    public static void main(String args[]) {
        // Get a valid student name.
System.out.print("Enter student name: ");
try {
    student_name = inStream.readLine();
} catch (Exception iox)
    { iox.printStackTrace(); }
// Get a valid student grade point average
while (true) {
    System.out.print("Enter student gpa: ");
    try {
        student_gpa = Double.valueOf(inStream.readLine()).doubleValue();
    } catch (Exception iox)
        { student_gpa = -1.0; }
    if (((student_gpa >= 0.0) && (student_gpa <= 4.0))
        break;
    else
        System.out.println("Invalid gpa--must be between 0.0 and 4.0");
}
// Get a valid student hours completed.
while (true) {
    System.out.print("Enter student hours completed: ");
    try {
        student_hours = Integer.valueOf(inStream.readLine()).intValue();
    } catch (Exception iox)
        { student_hours = -1; }
    if (student_hours > 0)
        break;
    else
        System.out.println("Invalid hours--must be positive integer");
}
// Test for eligibility.
if (((student_gpa>=2.0) && (student_hours >= 126))
    System.out.println(student_name + " is eligible to graduate.");
else
    System.out.println(student_name + " is not eligible to graduate.");
}

3.3 Procedural Abstraction
In this section we will look at a succession of programs to demonstrate the three phases of
procedural abstraction.

3.3.1 Implementation Hiding
Procedural implementation hiding is supported by the ability to group statements together
and reference them by a single name. For example, the first five lines in the main program of List-
ing 3.1 are grouped together in Listing 3.2 into a single module called get_valid_name. The
Java mechanism used for this grouping of statements is the function. In Listing 3.2 you can see
that the statements from the main program of Listing 3.1 have been grouped into four function.
Because we are not yet concerned with independence or reuse, these four functions are all void
functions (meaning they return no value), parameterless functions, and functions with no inter-
ally declared identifiers.
The primary benefit of implementation hiding is that it breaks the problem into a few fundamental tasks whose implementation is hidden in the program unit that calls these tasks. The tasks are then relegated to functions, each of which performs a single, cohesive action.

Listing 3.2
// Solution of the student graduation problem
// using procedural implementation hiding

import java.io.*;

public class Listing_2
{
    // Reader for input data
    static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

    static String student_name; // Student’s name
    static double student_gpa; // Student’s grade point average
    static int    student_hours; // Student’s number of credit hours completed

    // Reads a name from inStream
    public static void get_valid_name()
    {
        System.out.print(“Enter student name: “);
        try {
            student_name = inStream.readLine();
        } catch (Exception iox) {
            iox.printStackTrace();
        }
    }

    // Reads and validates the grade point average from inStream
    public static void get_valid_gpa()
    {
        while (true)
        {
            System.out.print(“Enter student gpa: “);
            try {
                student_gpa = Double.valueOf(inStream.readLine()).doubleValue();
            } catch (Exception iox) {
                student_gpa = -1.0;
            }
            if ((student_gpa >= 0.0) && (student_gpa <= 4.0))
                break;
            else
                System.out.println(“Invalid gpa--must be between 0.0 and 4.0“);
        }
    }

    // Reads and validates the credit hours completed from inStream
    public static void get_valid_hours()
while (true)
{
    System.out.print("Enter student hours completed: ");
    try {
        student_hours = Integer.valueOf(inStream.readLine()).intValue();
    } catch (Exception iox) {
        student_hours = -1;
    }
    if (student_hours > 0)
        break;
    else
        System.out.println("Invalid hours--must be positive integer");
}

// Checks eligibility for graduation and prints result
public static void check_for_graduation()
{
    if ((student_gpa>=2.0) && (student_hours >= 126))
        System.out.println(student_name + " is eligible to graduate.");
    else
        System.out.println(student_name + " is not eligible to graduate.");
}

public static void main(String args[])
{
    get_valid_name();
    get_valid_gpa();
    get_valid_hours();
    check_for_graduation();
}

3.3.2 Independence

The next step in procedural abstraction is independence. The dependence that we are addressing in Listing 3.2 is the fact that the function modules depend upon knowing the variable names used by the main program and vice versa. We eliminate this dependence by introducing parameters and return values for the functions.

For example, consider the function get_valid_name in Listing 3.3. Whereas in Listing 3.2 this function needed to know the name of the variable student_name used in the main program, in Listing 3.3 a variable named name is declared inside the function and that declaration makes get_valid_name independent from main. The function then returns the value obtained to main, which can then put that value into a variable of any name that main chooses, independently of the name used in the function.

Just as local declarations and return values are used to permit independence of names for objects sent out of a module, parameters allow independence of names for objects sent into modules. Look at the new version of check_for_graduation given in Listing 3.3. Since student_name, student_gpa, and student_hours are sent into this function as parameters, the function is independent of main program in the sense that it does not need to know the
names of any of main's objects.

Independence further advances procedural abstraction because it makes it possible to write a function module without any knowledge of the implementation of other modules. The only information modules need about each other are their interface protocols.

Listing 3.3

// Solution of the student graduation problem
// using procedural implementation hiding and independence

import java.io.*;

public class Listing_3
{
    // Reader for input data
    static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

    // Reads a name from inStream, returns valid name
    public static String get_valid_name()
    {
        String name = "";
        System.out.print("Enter student name: ");
        try {
            name = inStream.readLine();
        } catch (Exception iox) {
            iox.printStackTrace();
        }
        return name;
    }

    // Reads and validates the grade point average from inStream
    // Returns valid gpa
    public static double get_valid_gpa()
    {
        double gpa;
        while (true)
        {
            System.out.print("Enter student gpa: ");
            try {
                gpa = Double.valueOf(inStream.readLine()).doubleValue();
            } catch (Exception iox) {
                gpa = -1.0;
            }
            if ((gpa >= 0.0) && (gpa <= 4.0))
                break;
            else
                System.out.println("Invalid gpa--must be between 0.0 and 4.0");
        }
        return gpa;
    }

    // Reads and validates the credit hours completed from inStream
// Returns valid hours
public static int get_valid_hours()
{
    int hours;
    while (true)
    {
        System.out.print("Enter student hours completed: ");
        try {
            hours = Integer.valueOf(inStream.readLine()).intValue();
        } catch (Exception iox)
        {
            hours = -1;
        }
        if (hours > 0)
            break;
        else
            System.out.println("Invalid hours--must be positive integer");
    }
    return hours;
}

// Checks eligibility for graduation and prints result.
// Student name, gpa and hours completed are sent in as parameters.
public static void check_for_graduation(String name, double gpa, int hours)
{
    if ((gpa>=2.0) && (hours >= 126))
        System.out.println(name + " is eligible to graduate.");
    else
        System.out.println(name + " is not eligible to graduate.");
}

public static void main(String args[])
{
    String student_name; // Student’s name
    double student_gpa;// Student’s grade point average
    int    student_hours;// Student’s number of credit hours completed
    student_name  = get_valid_name();
    student_gpa   = get_valid_gpa();
    student_hours = get_valid_hours();
    check_for_graduation(student_name,student_gpa,student_hours);
}

3.3.3 Reuse

In addition to their value in specifying independence, function parameters can also be valuable for introducing procedural reuse. Reuse refers to the ability to make use of the same function in other, slightly different settings.

To illustrate this, Listing 3.4 shows how three of the functions can be parameterized to enable them to be reused in different circumstances. The functions get_valid_gpa and get_valid_hours are given parameters to specify the smallest and largest allowable values. Also, the function check_for_graduation is made reusable by including parameters for the minimum gpa and the minimum number of credit hours needed to qualify for graduation.
While reuse gives no advantage in a single application of a function, considerable savings can be realized in later reapplications of the function under slightly different conditions. This is illustrated in Listing 3.4 by the introduction of a second student with different bounds in `main`.

Listing 3.4
// Solution of the student graduation problem
// using procedural information hiding, independence, and reuse

import java.io.*;

class Listing_4 {

  // Reader for input data
  static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

  // Reads a name from inStream, returns valid name
  public static String get_valid_name() {
    String name = "";
    System.out.print("Enter student name: ");
    try {
      name = inStream.readLine();
    } catch (Exception iox) {
      iox.printStackTrace();
    }
    return name;
  }

  // Reads and validates the grade point average from inStream.
  // Returns valid gpa. gpa_lo and gpa_hi are the bounds on allowable gpas.
  public static double get_valid_gpa(double gpa_lo, double gpa_hi) {
    double gpa;
    while (true) {
      System.out.print("Enter student gpa: ");
      try {
        gpa = Double.valueOf(inStream.readLine()).doubleValue();
      } catch (Exception iox) {
        gpa = gpa_lo - 1;
      }
      if ((gpa >= gpa_lo) && (gpa <= gpa_hi))
        break;
    } else
      System.out.println("Invalid gpa--must be between "+gpa_lo+" and "+gpa_hi);
    return gpa;
  }
}
// Reads and validates the credit hours completed from inStream
// Returns valid hours. hours_lo and hours_hi are bounds on hours.
public static int get_valid_hours(int hours_lo, int hours_hi)
{
    int hours;
    while (true)
    {
        System.out.print("Enter student hours completed: ");
        try {
            hours = Integer.valueOf(inStream.readLine()).intValue();
        } catch (Exception iox)
        {
            hours = hours_lo - 1;
        }
        if ((hours >= hours_lo) && (hours <= hours_hi))
            break;
        else
            System.out.println("Invalid hours--must be between " + hours_lo + " and " + hours_hi);
    }
    return hours;
}

// Checks eligibility for graduation and prints result.
// Student name, gpa and hours completed are sent in as parameters.
// Student eligible if gpa >= min_gpa and hours >= min_hours.
public static void check_for_graduation(String name, double gpa, int hours, double min_gpa, int min_hours)
{
    if ((gpa>=min_gpa) && (hours >= min_hours))
        System.out.println(name + " is eligible to graduate.");
    else
        System.out.println(name + " is not eligible to graduate.");
}

public static void main(String args[])
{
    String student_name; // Student’s name
    double student_gpa; // Student’s grade point average
    int student_hours; // Student’s number of credit hours completed

    student_name = get_valid_name();
    student_gpa = get_valid_gpa(0.0, 4.0);
    student_hours = get_valid_hours(1, Integer.MAX_VALUE);
    check_for_graduation(student_name, student_gpa, student_hours, 2.0, 126);

    // Second student with different allowable ranges and different requirements
    student_name = get_valid_name();
    student_gpa = get_valid_gpa(0.0, 5.0);
    student_hours = get_valid_hours(0, 200);
    check_for_graduation(student_name, student_gpa, student_hours, 2.5, 150);
}
3.4 Data Abstraction

3.4.1 Implementation Hiding

Just as procedural implementation hiding involves grouping a number of procedural objects together and referring to them by a single name, data implementation hiding is the result of grouping data objects together so they can be referenced as a unit. This simplifies the manipulation of such composite data when it is not necessary to reference its individual components, while retaining the ability to reference the components when required.

In Listing 3.5 we see such implementation hiding with the three data components comprising student data. Here the three components are collected together to form a class. This simplifies the declaration of objects to represent students into a single declaration. In Java this declaration is

```java
Student student = new Student();
```

The fact that all three components are declared to be public within the class definition ensures that they are accessible to all procedural units. In particular, in `main` the components are set by referencing their names preceded by `student.` in the three assignment statements.

When `check_for_graduation` is called from `main`, all three components are passed as a single data object, `student`.

Listing 3.5
```
// Solution of the student graduation problem
// using procedural implementation hiding, independence, and reuse
// and data implementation hiding
import java.io.*;

public class Listing_5
{
    // Reader for input data
    static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

    // Data class Student for implementation hiding
    public static class Student
    {
        public String name;
        public double gpa;
        public int hours;
    }

    // Reads a name from inStream, returns valid name
    public static String get_valid_name()
    {
        String name = "";
        System.out.println("Enter student name: ");
        try {
            name = inStream.readLine();
        } catch (Exception iox)
        {
            iox.printStackTrace();
        }
        return name;
    }
}
public static double get_valid_gpa(double gpa_lo, double gpa_hi) {
    double gpa;
    while (true) {
        System.out.print("Enter student gpa: ");
        try {
            gpa = Double.valueOf(inStream.readLine()).doubleValue();
        } catch (Exception iox) {
            gpa = gpa_lo - 1;
        }
        if ((gpa >= gpa_lo) && (gpa <= gpa_hi))
            break;
        else
            System.out.println("Invalid gpa--must be between " + gpa_lo + " and " + gpa_hi);
    }
    return gpa;
}

public static int get_valid_hours(int hours_lo, int hours_hi) {
    int hours;
    while (true) {
        System.out.print("Enter student hours completed: ");
        try {
            hours = Integer.valueOf(inStream.readLine()).intValue();
        } catch (Exception iox) {
            hours = hours_lo - 1;
        }
        if ((hours >= hours_lo) && (hours <= hours_hi))
            break;
        else
            System.out.println("Invalid hours--must be between " + hours_lo + " and " + hours_hi);
    }
    return hours;
}

public static void check_for_graduation(Student s, double min_gpa, int min_hours) {
    if ((s.gpa>=min_gpa) && (s.hours >= min_hours))
System.out.println(s.name + " is eligible to graduate.");
else
    System.out.println(s.name + " is not eligible to graduate.");
}

public static void main(String args[])
{
    Student student = new Student();
    student.name = get_valid_name();
    student.gpa = get_valid_gpa(0.0, 4.0);
    student.hours = get_valid_hours(1, Integer.MAX_VALUE);
    check_for_graduation(student, 2.0, 126);
}

3.4.2 Independence

The data composition in Listing 3.5 lacks independence since any procedure that uses the student data must know the names of the components in order to set or retrieve their values. We next proceed to move the actual data into a protected mode by declaring each data component to be protected rather than public. Since this does not eliminate the need to set and retrieve the data components, it is now necessary to provide an interface for the Student class. This interface takes the form of a constructor for setting the component values and an accessor method for each component for retrieving their values. In general, it is also necessary to provide assignor methods to assign values to the individual components, but this is not required to solve the student graduation problem since the component values are never changed after they are set by the constructor.

Note how main has now bee reduced to two statements, one to declare the student and initialize its components, and the second to check for graduation eligibility. The Student class has been made more complex, but this gives it an extra measure of independence while providing a basic set of method interfaces.

Listing 3.6
// Solution of the student graduation problem
// using procedural implementation hiding, independence, and reuse
// and data implementation hiding and independence
import java.io.*;

public class Listing_6
{
    // Reader for input data
    static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

    // Class Student with constructor and accessors
    public static class Student
    {
        protected String name;
        protected double gpa;
        protected int hours;
    }
public Student(String name, double gpa, int hours) {
    this.name = name;
    this.gpa = gpa;
    this.hours = hours;
}

public String get_name() {
    return name;
}

public double get_gpa() {
    return gpa;
}

public int get_hours() {
    return hours;
}

public static String get_valid_name() {
    String name = "";
    System.out.print("Enter student name: ");
    try {
        name = inStream.readLine();
    } catch (Exception iox) {
        iox.printStackTrace();
    }
    return name;
}

public static double get_valid_gpa(double gpa_lo, double gpa_hi) {
    double gpa;
    while (true) {
        System.out.print("Enter student gpa: ");
        try {
            gpa = Double.valueOf(inStream.readLine()).doubleValue();
        } catch (Exception iox) {
            gpa = gpa_lo - 1;
        }
        if ((gpa >= gpa_lo) && (gpa <= gpa_hi))
            break;
        else
            System.out.println("Invalid gpa--must be between " + gpa_lo + " and " + gpa_hi);
    }
    return gpa;
}
// Reads and validates the credit hours completed from inStream
// Returns valid hours. hours_lo and hours_hi are bounds on hours.
public static int get_valid_hours(int hours_lo, int hours_hi)
{
    int hours;
    while (true)
    {
        System.out.print("Enter student hours completed: ");
        try {
            hours = Integer.valueOf(inStream.readLine()).intValue();
        } catch (Exception iox)
        {
            hours = hours_lo - 1;
        }
        if ((hours >= hours_lo) && (hours <= hours_hi))
            break;
        else
            System.out.println("Invalid hours--must be between "+hours_lo+
            " and "+hours_hi);
    }
    return hours;
}

// Checks eligibility for graduation and prints result.
// Student data aggregate s is sent in as a parameter.
// Student eligible if gpa >= min_gpa and hours >= min_hours.
public static void check_for_graduation(Student s,double min_gpa,int min_hours)
{
    if ((s.get_gpa()>=min_gpa) && (s.get_hours() >= min_hours))
        System.out.println(s.get_name() + " is eligible to graduate.");
    else
        System.out.println(s.get_name() + " is not eligible to graduate.");
}

public static void main(String args[])
{
    Student student = new Student(get_valid_name(),get_valid_gpa(0.0,4.0),
        get_valid_hours(1,Integer.MAX_VALUE));
    check_for_graduation(student,2.0,126);
}

3.5 Abstract Data Types and Object Orientation

In the preceding two sections we have seen how both procedures and data can be
abstracted through implementation hiding, independence, and reuse. The next step is to encapsu-
late procedures and data together into a single abstraction. In this section we will do this in two
phases. The first will incorporate implementation hiding and independence while the second will
add reuse. These two steps are commonly referred to as Abstract Data Types and Object-Orienta-
tion, respectively.
3.5.1 Abstract Data Types

An abstract data type is an abstraction of a data type that incorporates both its data components and the procedures that operate on that data type. This construct and the language issues related to it are discussed in detail in Chapter 8. Here we will show the Java implementation of our student graduation problem to illustrate how an abstract data type provides implementation hiding and independence jointly for the data and the procedures associated with a student. In our previous listings, the class has been used to provide an abstraction for data only, but in those cases, we were not using the full capabilities of classes.

The grouping of procedures and data is called a class in Java. The implementation hiding of the procedures are in the form of methods such as `get_valid_name`, `get_valid_gpa`, `get_valid_hours`, and `check_for_graduation`, as before. But now, these methods are encapsulated within the class and they are called by attaching the method name to an object belonging to the class.

This joint abstraction is found in Listing 3.6, where the abstraction of the data is the same as in Listing 3.5 with the methods included in the class definition. In this new listing, the accessor methods from Listing 3.5 are not included as the values for the data components are read and assigned in the methods local to the class. The data components of the class, as protected elements, are accessible only to units inside the class. This ensures independence of the class from the procedural units that use it.

In Listing 3.6 a constructor is given that accepts as parameters the lower and upper bounds of `gpa` and `hours` and then proceeds to read all of the data components by calling on the local methods provided for this purpose. The main method is now reduced to a declaration of a `Student` and a call on that `Student`’s `check_for_graduation`.

Listing 3.6

```java
import java.io.*;

public class Listing_3_6
{
    static BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));

    public static class Student
    {
        protected String name;
        protected double gpa;
        protected int hours;

        public Student(double gpa_lo, double gpa_hi, int hours_lo, int hours_hi)
        {
            this.name = get_valid_name();
            this.gpa  = get_valid_gpa(gpa_lo, gpa_hi);
            this.hours = get_valid_hours(hours_lo, hours_hi);
        }

        public Student(String name, double gpa, int hours)
        {
```
this.name = name;
this.gpa = gpa;
this.hours= hours;
}

public String get_valid_name()
{
    String name = "";
    System.out.print("Enter student name: ");
    try {
        name = inStream.readLine();
    } catch (Exception iox)
    {
        iox.printStackTrace();
    }
    return name;
}

public double get_valid_gpa(double gpa_lo, double gpa_hi)
{
    double gpa;
    while (true)
    {
        System.out.print("Enter student gpa: ");
        try {
            gpa = Double.valueOf(inStream.readLine()).doubleValue();
        } catch (Exception iox)
        {
            gpa = gpa_lo - 1;
        }
        if ((gpa >= gpa_lo) && (gpa <= gpa_hi))
            break;
        else
            System.out.println("Invalid gpa--must be between "+ gpa_lo + " and "+ gpa_hi);
    }
    return gpa;
}

public int get_valid_hours(int hours_lo, int hours_hi)
{
    int hours;
    while (true)
    {
        System.out.print("Enter student hours completed: ");
        try {
            hours = Integer.valueOf(inStream.readLine()).intValue();
        } catch (Exception iox)
        {
            hours = hours_lo - 1;
        }
        if ((hours >= hours_lo) && (hours <= hours_hi))
            break;
        else
            System.out.println("Invalid hours--must be between "+ hours_lo + " and "+ hours_hi);
    }
return hours;
}

public void check_for_graduation(double min_gpa, double min_hours)
{
    if ((gpa>=min_gpa) && (hours >= min_hours))
        System.out.println(name + " is eligible to graduate.");
    else
        System.out.println(name + " is not eligible to graduate.");
}

public static void main(String args[])
{
    Student student = new Student(0.0,4.0,1,Integer.MAX_VALUE);
    student.check_for_graduation(2.0,126);
}

3.5.2 Object-Oriented Abstraction

We now carry this development on step further by adding a reuse capability to the Abstract Data Type. This permits us to define new abstract data types that extend ADTs that are already defined by adding new components to the already-defined class. This is done through the mechanism of inheritance and results in what is called Object-Orientation. This approach will be described in detail in Chapter 8.

In our case, we construct a new type of student in Listing 3.7. This new type is defined by the class NewStudent, which extends the class Student. This means the data components, the constructors, and the methods of Student are all reused by NewStudent.

Objects of class NewStudent must meet an additional requirement for graduation, namely, that of having completed a major. This is checked by printing a prompt asking if the student has a major and accepting 'y' or 'n' in response. Therefore, in addition to the data components reused from Student, NewStudent adds a char component named major. The constructor for NewStudent reuses the constructor for Student by calling super(...);
and then provides one additional assignment, this one to the major component. One new method is added to get_valid_major while the methods to get the other three components are reused from Student. Finally, a new version of check_for_graduation is provided for NewStudent that carries out the modified test for graduation. This reuse of one class in another is called inheritance.

The main method in Listing 3.7 shows how this reuse can be applied effectively. Here a variable student of class Student is first associated with a Student object. This object is then tested for graduation. Next, the same variable is associated with a NewStudent. This is permitted because a variable declared to be a Student may be associated with an object of any class that extends Student (such as NewStudent) as well as with an object of class Student. Now when student.check_for_graduation is called, the NewStudent version is executed so a check for the major will be included.
import java.io.*;
import Student;

public class Listing_3_7
{
    public static class NewStudent extends Student
    {
        protected char major;

        public NewStudent(double gpa_lo, double gpa_hi, int hours_lo, int hours_hi, BufferedReader inReader)
        {
            super(gpa_lo, gpa_hi, hours_lo, hours_hi, inReader);
            this.major = get_valid_major();
        }

        public char get_valid_major()
        {
            char response = ' ';
            while (true)
            {
                System.out.print("Does the student have a major? (y or n) ");
                try {
                    response = inStream.readLine().toLowerCase().charAt(0);
                } catch (Exception iox) {
                    iox.printStackTrace();
                }
                if ((response=='y') || (response=='n'))
                    break;
                else
                    System.out.println("Response must be y or n.");
            }
            return response;
        }

        public void check_for_graduation(double min_gpa, double min_hours)
        {
            if ((gpa>=min_gpa) && (hours >= min_hours) && (major=='y'))
                System.out.println(name + " is eligible to graduate.");
            else
                System.out.println(name + " is not eligible to graduate.");
        }
    }

    public static void main(String args[])
    {
        BufferedReader inReader = new BufferedReader(new InputStreamReader(System.in));
        Student student = new Student(0.0,5.0,1,200,inReader);
        student.check_for_graduation(2.0,130);
    }
}
student = new NewStudent(0.0, 4.0, 1, Integer.MAX_VALUE, inReader);
student.check_for_graduation(2.0, 126);