
7 Object-Oriented Design

Chapter Objectives

- Establish key issues related to the design of object-oriented software.
- Explore techniques for identifying the classes and objects needed in a program.
- Discuss the relationships among classes.
- Describe the effect of the `static` modifier on methods and data.
- Discuss the creation of a formal object interface.
- Further explore the definition of enumerated type classes.
- Discuss issues related to the design of methods, including method overloading.
- Explore issues related to the design of graphical user interfaces.
- Discuss events generated by the mouse and the keyboard.

This chapter extends our discussion of the design of object-oriented software. We first focus on the stages of software development and the process of identifying classes and objects in the problem domain. We then discuss various issues that affect the design of a class, including static members, class relationships, interfaces, and enumerated types. We also explore design issues at the method level and introduce the concept of method overloading. A discussion of testing strategies rounds out these issues. In the Graphics Track sections of this chapter,

we discuss the characteristics of a well-designed graphical user interface (GUI) and explore mouse and keyboard events.


7.1 Software Development Activities

Creating software involves much more than just writing code. As the problems you tackle get bigger, and the solutions include more classes, it becomes crucial to carefully think through the design of the software. Any proper software development effort consists of four basic *development activities*:

- establishing the requirements
- creating a design
- implementing the design
- testing

It would be nice if these activities, in this order, defined a step-by-step approach for developing software. However, although they may seem to be sequential, they are almost never completely linear in reality. They overlap and interact. Let's discuss each development activity briefly.

Software requirements specify *what* a program must accomplish. They indicate the tasks that a program should perform, not how it performs them. Often, requirements are expressed in a document called a *functional specification*.

We discussed in [Chapter 1](#)  the basic premise that programming is really about problem solving; we create a program to solve a particular problem. Requirements are the clear expression of that problem. Until we truly know what problem we are trying to solve, we can't actually solve it.

The person or group who wants a software product developed (the *client*) will often provide an initial set of requirements. However, these initial requirements are often incomplete, ambiguous, and perhaps even contradictory. The software developer must work with the client to refine the requirements until all key decisions about what the system will do have been addressed.

Requirements often address user interface issues such as output format, screen layouts, and graphical interface components. Essentially, the requirements establish the characteristics that make the program useful for the end user. They may also apply constraints to your program, such as how fast a task must be performed.

A *software design* indicates *how* a program will accomplish its requirements. The design specifies the classes and objects needed in a program and defines how they interact. It also specifies the relationships among the classes. Low-level design issues deal with how individual methods accomplish their tasks.

A civil engineer would never consider building a bridge without designing it first. The design of software is no less essential. Many problems that occur in software are directly attributable to a lack of

good design effort. It has been shown time and again that the effort spent on the design of a program is well worth it, saving both time and money in the long run.


During software design, alternatives need to be considered and explored. Often, the first attempt at a design is not the best solution. Fortunately, changes are relatively easy to make during the design stage.

Key Concept

The effort put into design is both crucial and cost effective.

Implementation ⓘ is the process of writing the source code that will solve the problem. More precisely, implementation is the act of translating the design into a particular programming language. Too many programmers focus on implementation exclusively when actually it should be the least creative of all development activities. The important decisions should be made when establishing the requirements and creating the design.

Testing ⓘ is the act of ensuring that a program will solve the intended problem given all of the constraints under which it must perform. Testing includes running a program multiple times with various inputs



and carefully scrutinizing the results. But it means far more than that. We revisit the issues related to testing in [Section 7.9](#) .

Self-Review Questions

(see answers in [Appendix L](#) )


SR 7.1 Name the four basic activities that are involved in a software development process.

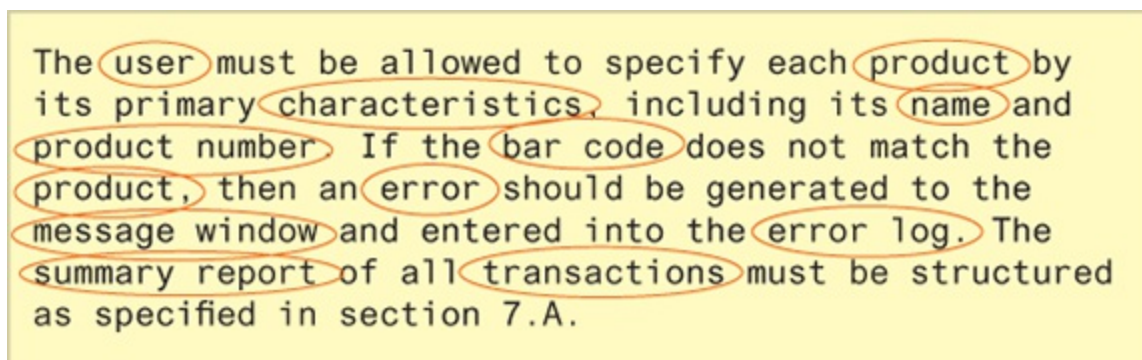
SR 7.2 Who creates/specifies software requirements, the client or the developer? Discuss.

SR 7.3 Compare and contrast the four basic development activities presented in this section with the five general problem-solving steps presented in [Chapter 1](#)  ([Section 1.6](#) )

7.2 Identifying Classes and Objects

A fundamental part of object-oriented software design is determining the classes that will contribute to the program. We have to carefully consider how we want to represent the various elements that make up the overall solution. These classes determine the objects that we will manage in the system.

One way to identify potential classes is to identify the objects discussed in the program requirements. Objects are generally nouns. You literally may want to scrutinize a problem description, or a functional specification if available, to identify the nouns found in it. For example, **Figure 7.1**  shows part of a problem description with the nouns circled.



The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Figure 7.1 A partial problem description with the nouns circled

Of course, not every noun in the problem specification will correspond to a class in your program. This activity is just a starting point that allows you to think about the types of objects a program will manage.

Key Concept

The nouns in a problem description may indicate some of the classes and objects needed in a program.

Remember that a class represents a group of objects with similar behavior. A plural noun in the specification, such as products, may indicate the need for a class that represents one of those items, such as `Product`. Even if there is only one of a particular kind of object needed in your system, it may best be represented as a class.

Classes that represent objects should generally be given names that are singular nouns, such as `Coin`, `Student`, and `Message`. A class represents a single item from which we are free to create as many instances as we choose.

Another key decision is whether to represent something as an object or as a primitive attribute of another object. For example, we may initially think that an employee's salary should be represented as an integer, and that may work for much of the system's processing. But

upon further reflection we might realize that the salary is based on the person's rank, which has upper and lower salary bounds that must be managed with care. Therefore the final conclusion may be that we'd be better off representing all of that data and the associated behavior as a separate class.

Given the needs of a particular program, we want to strike a good balance between classes that are too general and those that are too specific. For example, it may complicate our design unnecessarily to create a separate class for each type of appliance that exists in a house. It may be sufficient to have a single `Appliance` class, with perhaps a piece of instance data that indicates what type of appliance it is. Then again, it may not. It all depends on what the software is intended to accomplish.

In addition to classes that represent objects from the problem domain, we likely will need classes that support the work necessary to get the job done. For example, in addition to `Member` objects, we may want a separate class to help us manage all of the members of a club.

Keep in mind that when producing a real system, some of the classes we identify during design may already exist. Even if nothing matches exactly, there may be an old class that's similar enough to serve as the basis for our new class. The existing class may be part of the Java standard class library, part of a solution to a problem we've solved previously, or part of a library that can be bought from a third party. These are all examples of software reuse.

Assigning Responsibilities

Part of the process of identifying the classes needed in a program is the process of assigning responsibilities to each class. Each class represents an object with certain behaviors that are defined by the methods of the class. Any activity that the program must accomplish must be represented somewhere in the behaviors of the classes. That is, each class is responsible for carrying out certain activities, and those responsibilities must be assigned as part of designing a program.

The behaviors of a class perform actions that make up the functionality of a program. Thus we generally use verbs for the names of behaviors and the methods that accomplish them.

Sometimes it is challenging to determine which is the best class to carry out a particular responsibility. Consider multiple possibilities. Sometimes such analysis makes you realize that you could benefit from defining another class to shoulder the responsibility.

It's not necessary in the early stages of a design to identify all the methods that a class will contain. It is often sufficient to assign primary responsibilities and consider how those responsibilities translate to particular methods.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 7.4 How can identifying the nouns in a problem specification help you design an object-oriented solution to the problem?

SR 7.5 Is it important to identify and define all of the methods that a class will contain during the early stages of problem solution design? Discuss.

7.3 Static Class Members

We've used static methods in various situations in previous examples in the book. For example, all the methods of the `Math` class are static. Recall that a static method is one that is invoked through its class name, instead of through an object of that class.



Exploring the `static` modifier.

Not only can methods be static, but variables can be static as well. We declare static class members using the `static` modifier.

Deciding whether to declare a method or variable as static is a key step in class design. Let's examine the implications of static variables and methods more closely.

Static Variables

So far, we've seen two categories of variables: local variables that are declared inside a method, and instance variables that are declared in a class but not inside a method. The term **instance variable** ⓘ is used, because each instance of the class has its own version of the variable. That is, each object has distinct memory space for each variable so that each object can have a distinct value for that variable.

Key Concept

A static variable is shared among all instances of a class.

A *static variable*, which is sometimes called a **class variable** ⓘ, is shared among all instances of a class. There is only one copy of a static variable for all objects of the class. Therefore, changing the value of a static variable in one object changes it for all of the others. The reserved word `static` is used as a modifier to declare a static variable as follows:

```
private static int count = 0;
```

Memory space for a static variable is established when the class that contains it is referenced for the first time in a program. A local variable declared within a method cannot be static.

Constants, which are declared using the `final` modifier, are often declared using the `static` modifier. Because the value of constants cannot be changed, there might as well be only one copy of the value across all objects of the class.

Static Methods

In [Chapter 3](#), we briefly introduced the concept of a *static method* (also called a *class method*). Static methods can be invoked through the class name. We don't have to instantiate an object of the class in order to invoke the method. In [Chapter 3](#), we noted that all the methods of the `Math` class are static methods. For example, in the following line of code the `sqrt` method is invoked through the `Math` class name:


```
System.out.println("Square root of 27: " + Math.sqrt(27));
```

The methods in the `Math` class perform basic computations based on values passed as parameters. There is no object state to maintain in these situations; therefore, there is no good reason to create an object in order to request these services.

A method is made static by using the `static` modifier in the method declaration. As we've seen many times, the `main` method of a Java

program must be declared with the `static` modifier; this is done so that `main` can be executed by the interpreter without instantiating an object from the class that contains `main`.

Because static methods do not operate in the context of a particular object, they cannot reference instance variables, which exist only in an instance of a class. The compiler will issue an error if a static method attempts to use a nonstatic variable. A static method can, however, reference static variables, because static variables exist independent of specific objects. Therefore, the `main` method can access only static or local variables.

The program in [Listing 7.1](#)  instantiates several objects of the `Slogan` class, printing each one out in turn. At the end of the program, it invokes a method called `getCount` through the class name, which returns the number of `Slogan` objects that were instantiated in the program.

Listing 7.1

```
//*****  
  
//  SloganCounter.java      Author: Lewis/Loftus  
//  
//  Demonstrates the use of the static modifier.  
//*****  
  

```



```
public class SloganCounter
{
    //-----
    //  Creates several Slogan objects and prints the number of
    //  objects that were created.
    //-----
    public static void main(String[] args)
    {
        Slogan obj;

        obj = new Slogan("Remember the Alamo.");
        System.out.println(obj);

        obj = new Slogan("Don't Worry. Be Happy.");
        System.out.println(obj);

        obj = new Slogan("Live Free or Die.");
        System.out.println(obj);


        obj = new Slogan("Talk is Cheap.");
        System.out.println(obj);

        obj = new Slogan("Write Once, Run Anywhere.");
        System.out.println(obj);
        System.out.println();
    }
}
```

```
        System.out.println("Slogans created: " +  
Slogan.getCount());  
    }  
}
```

Output

```
Remember the Alamo.  
Don't Worry. Be Happy.  
Live Free or Die.  
Talk is Cheap.  
Write Once, Run Anywhere.  
  
Slogans created: 5
```

Listing 7.2  shows the `Slogan` class. The constructor of `Slogan` increments a static variable called `count`, which is initialized to zero when it is declared. Therefore, `count` serves to keep track of the number of instances of `Slogan` that are created.

Listing 7.2

```
/*******  
  
// Slogan.java      Author: Lewis/Loftus
```

```

//
// Represents a single slogan string.
//*****

public class Slogan
{
    private String phrase;
    private static int count = 0;

    //-----

    // Constructor: Sets up the slogan and counts the number
    of
    // instances created.
    //-----

    public Slogan(String str)
    {
        phrase = str;
        count++;
    }

    //-----

    // Returns this slogan as a string.
    //-----

    -----

```

```
public String toString()
{
    return phrase;
}

//-----

// Returns the number of instances of this class that have
been
// created.
//-----

public static int getCount()
{
    return count;
}
}
```

The `getCount` method of `Slogan` is also declared as `static`, which allows it to be invoked through the class name in the `main` method. Note that the only data referenced in the `getCount` method is the integer variable `count`, which is static. As a static method, `getCount` cannot reference any nonstatic data.

The `getCount` method could have been declared without the `static` modifier, but then its invocation in the `main` method would have to

have been done through an instance of the `Slogan` class instead of the class itself.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 7.6 What is the difference between a static variable and an instance variable?

SR 7.7 Assume you are defining a `BankAccount` class whose objects each represent a separate bank account. Write a declaration for a variable of the class that will hold the combined total balance of all the bank accounts represented by the class.

SR 7.8 Assume you are defining a `BankAccount` class whose objects each represent a separate bank account. Write a declaration for a variable of the class that will hold the minimum balance that each account must maintain.

SR 7.9 What kinds of variables can the `main` method of any program reference? Why?

7.4 Class Relationships

The classes in a software system have various types of relationships to each other. Three of the more common relationships are dependency, aggregation, and inheritance.

We've seen dependency relationships in many examples in which one class “uses” another. This section revisits the dependency relationship and explores the situation where a class depends on itself. We then explore aggregation, in which the objects of one class contain objects of another, creating a “has-a” relationship. Inheritance, which we introduced in [Chapter 1](#), creates an “is-a” relationship between classes. We defer our detailed examination of inheritance until [Chapter 9](#).

Dependency

In many previous examples, we've seen the idea of one class being dependent on another. This means that one class relies on another in some sense. Often the methods of one class will invoke the methods of the other class. This establishes a “uses” relationship.

Generally, if class `A` uses class `B`, then one or more methods of class `A` invoke one or more methods of class `B`. If an invoked method is

static, then `A` merely references `B` by name. If the invoked method is not static, then `A` must have access to a specific instance of class `B` in order to invoke the method. That is, `A` must have a reference to an object of class `B`.

The way in which one object gains access to an object of another class is an important design decision. It occurs when one class instantiates the objects of another, but that's often the basis of an aggregation relationship. The access can also be accomplished by passing one object to another as a method parameter.

In general, we want to minimize the number of dependencies among classes. The less dependent our classes are on each other, the less impact changes and errors will have on the system.


Dependencies Among Objects of the Same Class

In some cases, a class depends on itself. That is, an object of one class interacts with another object of the same class. To accomplish this, a method of the class may accept as a parameter an object of the same class. Designing such a class drives home the idea that a class represents a particular object.

The `concat` method of the `String` class is an example of this situation. The method is executed through one `String` object and is passed another `String` object as a parameter. For example:

```
str3 = str1.concat(str2);
```

The `String` object executing the method (`str1`) appends its characters to those of the `String` passed as a parameter (`str2`). A new `String` object is returned as a result and stored as `str3`.

The `RationalTester` program shown in [Listing 7.3](#)  demonstrates a similar situation. A rational number is a value that can be represented as a ratio of two integers (a fraction). The `RationalTester` program creates two objects representing rational numbers and then performs various operations on them to produce new rational numbers.

Listing 7.3

```
//*****  
  
// RationalTester.java      Author: Lewis/Loftus  
//  
// Driver to exercise the use of multiple Rational objects.  
//*****
```



```

public class RationalTester
{
    //-----

    //  Creates some rational number objects and performs
    various
    //  operations on them.

    //-----

    public static void main(String[] args)
    {
        RationalNumber r1 = new RationalNumber(6, 8);
        RationalNumber r2 = new RationalNumber(1, 3);
        RationalNumber r3, r4, r5, r6, r7;

        System.out.println("First rational number: " + r1);
        System.out.println("Second rational number: " + r2);

        if (r1.isLike(r2))
            System.out.println("r1 and r2 are equal.");
        else
            System.out.println("r1 and r2 are NOT equal.");

        r3 = r1.reciprocal();
        System.out.println("The reciprocal of r1 is: " + r3);
        r4 = r1.add(r2);
    }
}

```

```

        r5 = r1.subtract(r2);
        r6 = r1.multiply(r2);
        r7 = r1.divide(r2);

        System.out.println("r1 + r2: " + r4);
        System.out.println("r1 - r2: " + r5);
        System.out.println("r1 * r2: " + r6);
        System.out.println("r1 / r2: " + r7);
    }
}

```

Output

```

First rational number: 3/4
Second rational number: 1/3
r1 and r2 are NOT equal.
The reciprocal of r1 is: 4/3
r1 + r2: 13/12
r1 - r2: 5/12
r1 * r2: 1/4
r1 / r2: 9/4

```

The `RationalNumber` class is shown in [Listing 7.4](#). Keep in mind as you examine this class that each object created from the `RationalNumber` class represents a single rational number. The

`RationalNumber` class contains various operations on rational numbers, such as addition and subtraction.

Listing 7.4

```

//*****

//  RationalNumber.java      Author: Lewis/Loftus
//
//  Represents one rational number with a numerator and
//  denominator.
//*****

public class RationalNumber
{
    private int numerator, denominator;

    //-----

    //  Constructor: Sets up the rational number by ensuring a
    //  nonzero
    //  denominator and making only the numerator signed.
    //-----

    public RationalNumber(int numer, int denom)
    {
        if (denom == 0)
            denom = 1;
    }
}
```

```
// Make the numerator "store" the sign
```

```
if (denom < 0)
```

```
{
```

```
    numer = numer * -1;
```

```
    denom = denom * -1;
```

```
}
```

```
    numerator = numer;
```

```
    denominator = denom;
```

```
    reduce();
```

```
}
```

```
//-----
```

```
// Returns the numerator of this rational number.
```

```
//-----
```

```
public int getNumerator()
```

```
{
```

```
    return numerator;
```

```
}
```

```
//-----
```

```
// Returns the denominator of this rational number.
```

```
//-----
```

```
-----  
  
public int getDenominator()  
{  
    return denominator;  
}
```

```
//-----
```

```
-----  
  
// Returns the reciprocal of this rational number.  
//-----  
  
-----  
  
public RationalNumber reciprocal()  
{  
    return new RationalNumber(denominator, numerator);  
}
```

```
//-----  
  
-----  
  
// Adds this rational number to the one passed as a  
parameter.  
  
// A common denominator is found by multiplying the  
individual  
  
// denominators.  
//-----
```

```
-----  
  
public RationalNumber add(RationalNumber op2)  
{  
    int commonDenominator = denominator *  
op2.getDenominator();
```

```
        int numerator1 = numerator * op2.getDenominator();  
        int numerator2 = op2.getNumerator() * denominator;  
        int sum = numerator1 + numerator2;  
  
        return new RationalNumber(sum, commonDenominator);  
    }  
  
    //-----  
    -----
```

```
        // Subtracts the rational number passed as a parameter  
        from this  
        // rational number.  
    //-----  
    -----
```

```
    public RationalNumber subtract(RationalNumber op2)  
    {  
        int commonDenominator = denominator *  
op2.getDenominator();  
        int numerator1 = numerator * op2.getDenominator();  
        int numerator2 = op2.getNumerator() * denominator;  
        int difference = numerator1 - numerator2;  
  
        return new RationalNumber(difference,  
commonDenominator);  
    }  
  
    //-----  
    -----
```

```

// Multiplies this rational number by the one passed as a
// parameter.
//-----

public RationalNumber multiply(RationalNumber op2)
{
    int numer = numerator * op2.getNumerator();
    int denom = denominator * op2.getDenominator();

    return new RationalNumber(numer, denom);
}

//-----

// Divides this rational number by the one passed as a
parameter
// by multiplying by the reciprocal of the second
rational.
//-----

public RationalNumber divide(RationalNumber op2)
{
    return multiply(op2.reciprocal());
}

//-----

// Determines if this rational number is equal to the one

```

```

passed

    // as a parameter. Assumes they are both reduced.

    //-----

-----

    public boolean isLike(RationalNumber op2)
    {
        return (numerator == op2.getNumerator() &&
                denominator == op2.getDenominator() );
    }

    //-----

-----

    // Returns this rational number as a string.

    //-----

-----

    public String toString()
    {
        String result;
        if (numerator == 0)
            result = "0";
        else
            if (denominator == 1)
                result = numerator + "";
            else
                result = numerator + "/" + denominator;
        return result;
    }

```



```

//-----
-----

// Reduces this rational number by dividing both the
numerator

// and the denominator by their greatest common divisor.

//-----
-----

private void reduce()
{
    if (numerator != 0)
    {
        int common = gcd(Math.abs(numerator), denominator);

        numerator = numerator / common;
        denominator = denominator / common;
    }
}

//-----
-----

// Computes and returns the greatest common divisor of the
two
// positive parameters. Uses Euclid's algorithm.

//-----
-----

private int gcd(int num1, int num2)
{
    while (num1 != num2)

```

```
        if (num1 > num2)
            num1 = num1 - num2;
        else
            num2 = num2 - num1;

    return num1;
}
}
```

The methods of the `RationalNumber` class, such as `add`, `subtract`, `multiply`, and `divide`, use the `RationalNumber` object that is executing the method as the first (left) operand and the `RationalNumber` object passed as a parameter as the second (right) operand.

The `isLike` method of the `RationalNumber` class is used to determine if two rational numbers are essentially equal. It's tempting, therefore, to call that method `equals`, similar to the method used to compare `String` objects (discussed in [Chapter 5](#)). In [Chapter 9](#), however, we will discuss how the `equals` method is somewhat special due to inheritance, and that it should be implemented in a particular way. So to avoid confusion we call this method `isLike` for now.

Note that some of the methods in the `RationalNumber` class, including `reduce` and `gcd`, are declared with private visibility. These methods are `private` because we don't want them executed directly from

outside a `RationalNumber` object. They exist only to support the other services of the object.

Aggregation


Some objects are made up of other objects. A car, for instance, is made up of its engine, its chassis, its wheels, and several other parts. Each of these other parts could be considered a separate object. Therefore, we can say that a car is an *aggregation*—it is composed, at least in part, of other objects. Aggregation is sometimes described as a *has-a relationship*. For instance, a car has a chassis.

Key Concept

An aggregate object is composed of other objects, forming a has-a relationship.

In the software world, we define an **aggregate object** ⓘ as any object that contains references to other objects as instance data. For example, an `Account` object contains, among other things, a `String` object that represents the name of the account owner. We sometimes forget that strings are objects, but technically that makes each `Account` object an aggregate object.

Aggregation is a special type of dependency. That is, a class that is defined in part by another class is dependent on that class. The methods of the aggregate object generally invoke the methods of the objects from which it is composed.

Let's consider another example. The program `StudentBody` shown in [Listing 7.5](#)  creates two `Student` objects. Each `Student` object is composed, in part, of two `Address` objects, one for the student's address at school and another for the student's home address. The `main` method does nothing more than create the `Student` objects and print them out. Once again we are passing objects to the `println` method, relying on the automatic call to the `toString` method to create a valid representation of the object that is suitable for printing.

Listing 7.5

[illegible]

```
-----  
    // Creates some Address and Student objects and prints  
    them.
```


```
    //-----
```

```
-----  
    public static void main(String[] args)  
    {  
        Address school = new Address("800 Lancaster Ave.",  
        "Villanova",  
        "PA", 19085);  
        Address jHome = new Address("21 Jump Street",  
        "Blacksburg",  
        "VA", 24551);  
        Student john = new Student("John", "Smith", jHome,  
        school);  
  
        Address mHome = new Address("123 Main Street", "Euclid",  
        "OH",  
        44132);  
        Student marsha = new Student("Marsha", "Jones", mHome,  
        school);  
  
        System.out.println(john);  
        System.out.println();  
        System.out.println(marsha);  
    }  
}
```

Output

```
John Smith
Home Address:
21 Jump Street
Blacksburg, VA 24551
School Address:
800 Lancaster Ave.
Villanova, PA 19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH 44132
School Address:
800 Lancaster Ave.
Villanova, PA 19085
```

The `Student` class shown in [Listing 7.6](#)  represents a single student. This class would have to be greatly expanded if it were to represent all aspects of a student. We deliberately keep it simple for now so that the object aggregation is clearly shown. The instance data of the `Student` class includes two references to `Address` objects. We refer to those objects in the `toString` method as we create a string representation of the student. By concatenating an `Address` object to

another string, the `toString` method in `Address` is automatically invoked.

Listing 7.6

```
//*****

//  Student.java      Author: Lewis/Loftus
//
//  Represents a college student.
//*****

public class Student
{
    private String firstName, lastName;
    private Address homeAddress, schoolAddress;

    //-----

    //  Constructor: Sets up this student with the specified
    values.
    //-----

    public Student(String first, String last, Address home,
                    Address school)
    {
        firstName = first;
        lastName = last;
```

```

        homeAddress = home;
        schoolAddress = school;
    }

    //-----
    // Returns a string description of this Student object.
    //-----

    public String toString()
    {
        String result;

        result = firstName + " " + lastName + "\n";
        result += "Home Address:\n" + homeAddress + "\n";
        result += "School Address:\n" + schoolAddress;

        return result;
    }
}

```

The `Address` class is shown in [Listing 7.7](#). It represents a street address. Note that nothing about the `Address` class indicates that it is part of a `Student` object. The `Address` class is kept generic by design and therefore could be used in any situation in which a street address is needed.

Listing 7.7

```
//*****

//  Address.java      Author: Lewis/Loftus
//
//  Represents a street address.
//*****

public class Address
{
    private String streetAddress, city, state;
    private long zipCode;

    //-----
    //  Constructor: Sets up this address with the specified
    data.
    //-----

    public Address(String street, String town, String st, long
zip)
    {
        streetAddress = street;
        city = town;
        state = st;
        zipCode = zip;
    }
}
```

```


    }

    //-----
    -----
    // Returns a description of this Address object.
    //-----
    -----
    public String toString()
    {
        String result;

        result = streetAddress + "\n";
        result += city + ", " + state + " " + zipCode;

        return result;
    }
}

```

The more complex an object, the more likely it will need to be represented as an aggregate object. In UML, aggregation is represented by a connection between two classes, with an open diamond at the end near the class that is the aggregate. **Figure 7.2**  shows a UML class diagram for the `StudentBody` program.

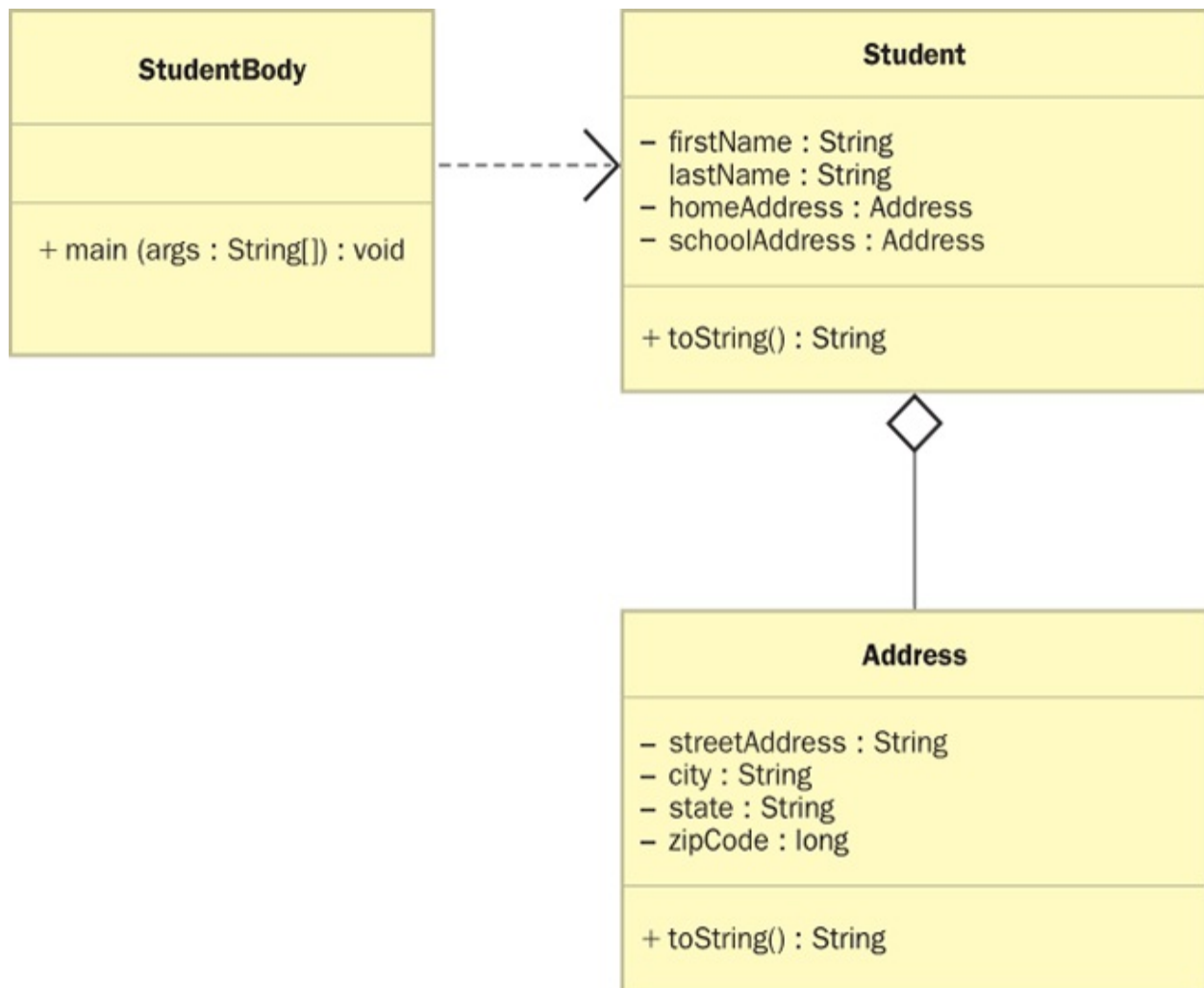


Figure 7.2 A UML class diagram showing aggregation

Note that in previous UML diagram examples and in [Figure 7.2](#), strings are not represented as separate classes with aggregation relationships, though technically they could be. Strings are so fundamental to programming that often they are represented as though they were a primitive type in a UML diagram.

The `this` Reference

Before we leave the topic of relationships among classes, we should examine another special reference used in Java programs called the `this` reference. The word `this` is a reserved word in Java. It allows an object to refer to itself. As we have discussed, a nonstatic method is invoked through (or by) a particular object or class. Inside that method, the `this` reference can be used to refer to the currently executing object.

For example, in a class called `ChessPiece` there could be a method called `move`, which could contain the following line:

```
if (this.position == piece2.position)
    result = false;
```


In this situation, the `this` reference is being used to clarify which position is being referenced. The `this` reference refers to the object through which the method was invoked. So when the following line is used to invoke the method, the `this` reference refers to `bishop1`:

```
bishop1.move();
```

However, when a different object is used to invoke the method, the `this` reference refers to that object. Therefore, when the following

invocation is used, the `this` reference in the `move` method refers to `bishop2`:

```
bishop2.move();
```

Often, the `this` reference is used to distinguish the parameters of a constructor from their corresponding instance variables with the same names. For example, the constructor of the `Account` class was presented in [Chapter 4](#)  as follows:

```
public Account(String owner, long account, double initial)
{
    name = owner;
    acctNumber = account;
    balance = initial;
}
```

When writing that constructor, we deliberately came up with different names for the parameters to distinguish them from the instance variables `name`, `acctNumber`, and `balance`. This distinction is arbitrary. The constructor could have been written as follows using the `this` reference:

```
public Account(String name, long acctNumber, double balance)
```

```
{  
    this.name = name;  
    this.acctNumber = acctNumber;  
    this.balance = balance;  
}
```

In this version of the constructor, the `this` reference specifically refers to the instance variables of the object. The variables on the right-hand side of the assignment statements refer to the formal parameters. This approach eliminates the need to come up with different yet equivalent names. This situation sometimes occurs in other methods but comes up often in constructors.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 7.10 Describe a dependency relationship between two classes.

SR 7.11 Explain how a class can have an association with itself.

SR 7.12 What is an aggregate object?

SR 7.13 What does the `this` reference refer to?


7.5 Interfaces

We've used the term interface to refer to the set of public methods through which we can interact with an object. That definition is consistent with our use of it in this section, but now we are going to formalize this concept using a particular language construct in Java.

A Java *interface* is a collection of constants and abstract methods. An *abstract method* is a method that does not have an implementation. That is, there is no body of code defined for an abstract method. The header of the method, including its parameter list, is simply followed by a semicolon. An interface cannot be instantiated.

Key Concept

An interface is a collection of abstract methods and therefore cannot be instantiated.

Listing 7.8  shows an interface called `Complexity`. It contains two abstract methods: `setComplexity` and `getComplexity`.

Listing 7.8

```

//*****

// Complexity.java      Author: Lewis/Loftus
//
// Represents the interface for an object that can be
// assigned an
// explicit complexity.
//*****

public interface Complexity
{
    public void setComplexity(int complexity);
    public int getComplexity();
}

```

An abstract method can be preceded by the reserved word `abstract`, though in interfaces it usually is not. Methods in interfaces have public visibility by default.

A class *implements* an interface by providing method implementations for each of the abstract methods defined in the interface. A class that implements an interface uses the reserved word `implements` followed by the interface name in the class header. If a class asserts that it implements a particular interface, it must provide a definition for all methods in the interface. The compiler will produce errors if any of the methods in the interface are not given a definition in the class.

The `Question` class, shown in [Listing 7.9](#), implements the `Complexity` interface. Both the `setComplexity` and `getComplexity` methods are implemented. They must be declared with the same signatures as their abstract counterparts in the interface. In the `Question` class, the methods are defined simply to set or return a numeric value representing the complexity level of the question that the object represents.

Listing 7.9

```
//*****  
  
//  Question.java      Author: Lewis/Loftus  
//  
//  Represents a question (and its answer).  
//*****  
  
  
public class Question implements Complexity  
{  
    private String question, answer;  
    private int complexityLevel;  
  
    //-----  
    -----  
    //  Constructor: Sets up the question with a default  
    complexity.
```

```
//-----  
-----  
public Question(String query, String result)  
{  
    question = query;  
    answer = result;  
    complexityLevel = 1;  
}
```

```
//-----  
-----  
// Sets the complexity level for this question.  
//-----  
-----
```

```
public void setComplexity(int level)  
{  
    complexityLevel = level;  
}
```

```
//-----  
-----  
// Returns the complexity level for this question.  
//-----  
-----
```

```
public int getComplexity()  
{  
    return complexityLevel;  
}
```

```
//-----  
-----  
// Returns the question.  
//-----  
-----  
public String getQuestion()  
{  
    return question;  
}  
  
//-----  
-----  
// Returns the answer to this question.  
//-----  
-----  
public String getAnswer()  
{  
    return answer;  
}  
  
//-----  
-----  
// Returns true if the candidate answer matches the  
answer.  
//-----  
-----  
public boolean answerCorrect(String candidateAnswer)
```


```

    {
        return answer.equals(candidateAnswer);
    }

    //-----
    -----
    // Returns this question (and its answer) as a string.
    //-----
    -----
    public String toString()
    {
        return question + "\n" + answer;
    }
}

```

Note that the `Question` class also implements additional methods that are not part of the `Complexity` interface. Specifically, it defines methods called `getQuestion`, `getAnswer`, `answerCorrect`, and `toString`, which have nothing to do with the interface. The interface guarantees that the class implements certain methods, but it does not restrict it from having additional ones. It is common for a class that implements an interface to have other methods.

Listing 7.10  shows a program called `MiniQuiz`, which uses some `Question` objects.

Listing 7.10

```

//*****

// MiniQuiz.java      Author: Lewis/Loftus
//
// Demonstrates the use of a class that implements an
// interface.
//*****

import java.util.Scanner;

public class MiniQuiz
{
    //-----
    -----
    // Presents a short quiz.
    //-----
    -----

    public static void main(String[] args)
    {
        Question q1, q2;
        String possible;

        Scanner scan = new Scanner(System.in);

        q1 = new Question("What is the capital of Jamaica?",
                           "Kingston");

```

```
q1.setComplexity(4);

q2 = new Question("Which is worse, ignorance or
apathy?",
"I don't know and I don't care");
q2.setComplexity(10);

System.out.print(q1.getQuestion());
System.out.println(" (Level: " + q1.getComplexity() +
")");
possible = scan.nextLine();
if (q1.answerCorrect(possible))
    System.out.println("Correct");
else
    System.out.println("No, the answer is " +
q1.getAnswer());

System.out.println();
System.out.print(q2.getQuestion());
System.out.println(" (Level: " + q2.getComplexity() +
")");
possible = scan.nextLine();
if (q2.answerCorrect(possible))
    System.out.println("Correct");
else
    System.out.println("No, the answer is " +
q2.getAnswer());
}
```

```
}
```

Output

```
What is the capital of Jamaica? (Level: 4)
```


```
Kingston
```

```
Correct
```

```
Which is worse, ignorance or apathy? (Level: 10)
```

```
apathy
```

```
No, the answer is I don't know and I don't care
```

An interface and its relationship to a class that implements it can be shown in a UML class diagram. An interface is represented similarly to a class node except that the designation `<<interface>>` is inserted above the interface name. A dotted arrow with a closed arrowhead is drawn from the class to the interface that it implements. **Figure 7.3**  shows a UML class diagram for the `MiniQuiz` program.

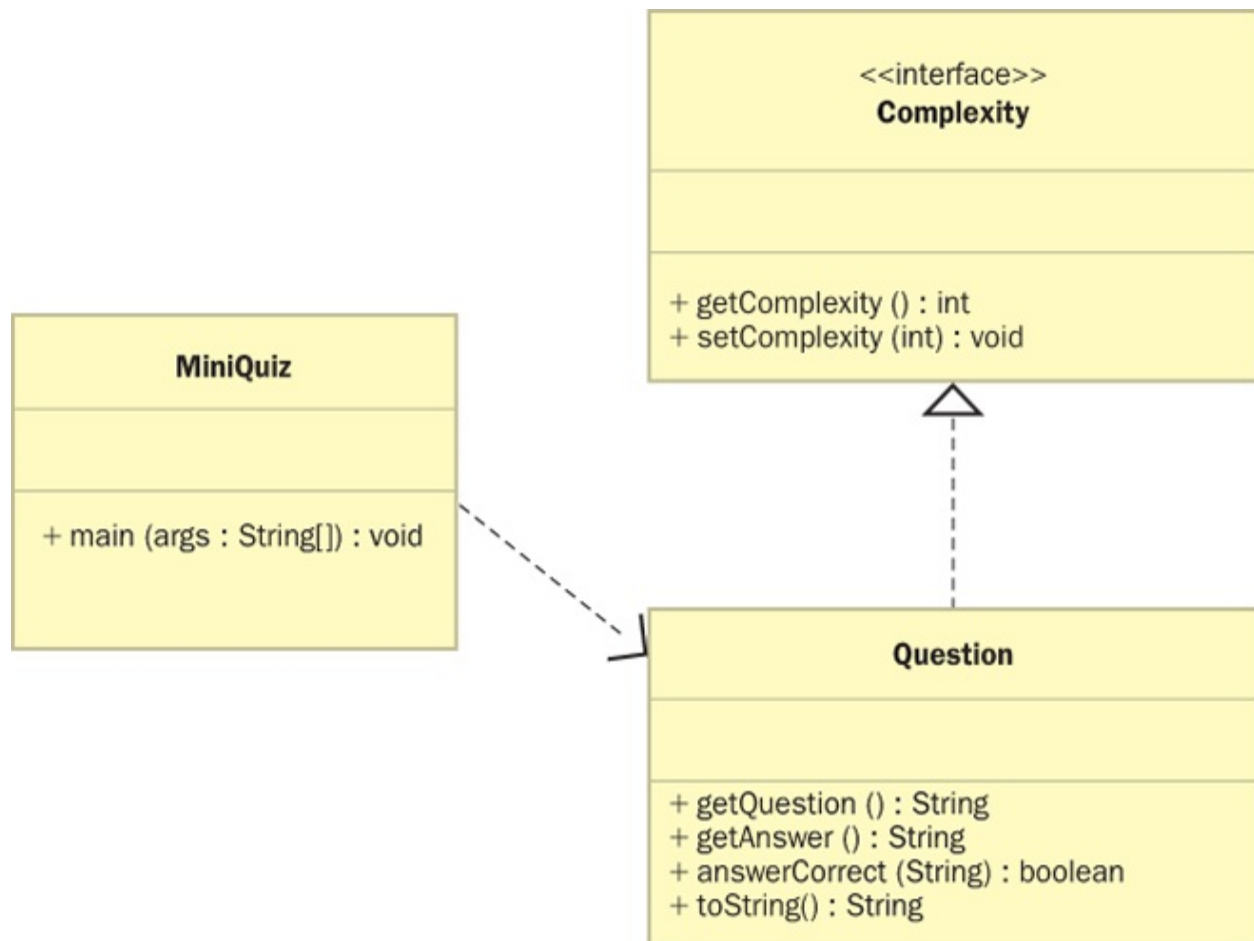


Figure 7.3 A UML class diagram for the `MiniQuiz` program


Multiple classes can implement the same interface, providing alternative definitions for the methods. For example, we could implement a class called `Task` that also implements the `Complexity` interface. In it we could choose to manage the complexity of a task in a different way (though it would still have to implement all the methods of the interface).

A class can implement more than one interface. In these cases, the class must provide an implementation for all methods in all interfaces

listed. To show that a class implements multiple interfaces, they are listed in the `implements` clause, separated by commas. For example:

```
class ManyThings implements Interface1, Interface2, Interface3
{
    // contains all methods of all interfaces
}
```

In addition to, or instead of, abstract methods, an interface can also contain constants, defined using the `final` modifier. When a class implements an interface, it gains access to all the constants defined in it.

The interface construct formally defines the ways in which we can interact with a class. It also serves as a basis for a powerful programming technique called polymorphism, which we discuss in [Chapter 10](#) .

The `Comparable` Interface

The Java standard class library contains interfaces as well as classes. The `Comparable` interface, for example, is defined in the `java.lang` package. The `Comparable` interface contains only one method, `compareTo`, which takes an object as a parameter and returns an integer.

The intention of this interface is to provide a common mechanism for comparing one object to another. One object calls the method and passes another as a parameter as follows:

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

As specified by the documentation for the interface, the integer that is returned from the `compareTo` method should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`. It is up to the designer of each class to decide what it means for one object of that class to be less than, equal to, or greater than another.

In [Chapter 5](#), we mentioned that the `String` class contains a `compareTo` method that operates in this manner. Now we can clarify that the `String` class has this method because it implements the `Comparable` interface. The `String` class implementation of this method bases the comparison on the lexicographic ordering defined by the Unicode character set.

The `Iterator` Interface

The `Iterator` interface is another interface defined as part of the Java standard class library. It is used by a class that represents a collection of objects, providing a means to move through the collection one object at a time.

In [Chapter 5](#), we defined the concept of an iterator, using a loop to process all elements in the collection. Most iterators, including objects of the `Scanner` class, are defined using the `Iterator` interface.

The two primary methods in the `Iterator` interface are `hasNext`, which returns a boolean result, and `next`, which returns an object. Neither of these methods takes any parameters. The `hasNext` method returns true if there are items left to process, and `next` returns the next object. It is up to the designer of the class that implements the `Iterator` interface to decide the order in which objects will be delivered by the `next` method.

We should note that, according to the spirit of the interface, the `next` method does not remove the object from the underlying collection; it simply returns a reference to it. The `Iterator` interface also has a method called `remove`, which takes no parameters and has a `void` return type. A call to the `remove` method removes the object that was most recently returned by the `next` method from the underlying collection.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 7.14 What is the difference between a class and an interface?

SR 7.15 Define a Java interface called `Nameable`. Classes that implement this interface must provide a `setName` method that requires a single `String` parameter and returns nothing, and a `getName` method that has no parameters and returns a `String`.

SR 7.16 True or False? Explain.

- a. A Java interface can include only abstract methods, nothing else.
- b. An abstract method is a method that does not have an implementation.
- c. All of the methods included in a Java interface definition must be abstract.
- d. A class that implements an interface can define only those methods that are included in the interface.
- e. Multiple classes can implement the same interface.
- f. A class can implement more than one interface.
- g. All classes that implement an interface must provide the exact same definitions of the methods that are included in the interface.

7.6 Enumerated Types Revisited

In [Chapter 3](#), we introduced the concept of an enumerated type, which defines a new data type and lists all possible values of that type. We gave an example that defined an enumerated type called `Season`, which was declared as follows:

```
enum Season {winter, spring, summer, fall}
```

We mentioned that an enumerated type is a special kind of class, and that the values of the enumerated type are objects. The values are, in fact, instances of its own enumerated type. For example, `winter` is an object of the `Season` class. Let's explore this concept a bit further.

Suppose we declare a variable of the `Season` type as follows:

```
Season time;
```

Key Concept

The values of an enumerated type are static variables of that type.

Because an enumerated type is a special kind of class, the variable `time` is an object reference variable. Furthermore, as an enumerated type, it can be assigned only the values listed in the `Season` definition. These values (`winter`, `spring`, `summer`, and `fall`) are actually references to `Season` objects that are stored as `public static` variables within the `Season` class. Thus we can make an assignment such as the following:

```
time = Season.spring;
```

Now let's take this idea a step further. In [Listing 7.11](#), we redefine the `Season` type, giving it a more substantial definition. Note that we still use the `enum` reserved word to declare the enumerated type, and we still list all possible values of the type. In addition, in this definition we add a private `String` called `span`, a constructor for the `Season` class, and a method named `getSpan`. Each value in the list of values for the enumerated type invokes the constructor, passing it a character string that is then stored in the `span` variable of each value.

Listing 7.11

```

//*****

// Season.java      Author: Lewis/Loftus
//
// Enumerates the values for Season.
//*****

public enum Season
{
    winter ("December through February"),
    spring ("March through May"),
    summer ("June through August"),
    fall ("September through November");

    private String span;

    //-----

    // Constructor: Sets up each value with an associated
    string.

    //-----

    Season(String months)
    {
        span = months;
    }

    //-----

```

```

-----
// Returns the span message for this value.
//-----
-----

public String getSpan()
{
    return span;
}
}

```

The `main` method of the `SeasonTester` class, shown in [Listing 7.12](#), prints each value of the `Season` enumerated type, as well as the span statement for each. Every enumerated type contains a static method called `values` that returns a list of all possible values for that type. This list is an iterator, so we can use the enhanced version of a `for` loop to process each value.

Listing 7.12

```

//*****

// SeasonTester.java          Author: Lewis/Loftus
//
// Demonstrates the use of a full enumerated type.
//*****

```



```

public class SeasonTester
{
    //-----

    // Iterates through the values of the Season enumerated
    type.
    //-----

    public static void main(String[] args)
    {
        for (Season time : Season.values())
            System.out.println(time + "\t" + time.getSpan());
    }
}

```

Output

```

winter  December through February
spring  March through May
summer  June through August
fall    September through November

```

In addition to the list of possible values defined in every enumerated type, we can include any number of attributes or methods of our own choosing. This provides various opportunities for creative class design.

Key Concept

We can add attributes and methods to the definition of an enumerated type.

Self-Review Question

(see answer in [Appendix L](#) )

SR 7.17 Using the enumerated type `Season` as defined in this section, what is the output from the following code sequence?

```
Season time1, time2;  
time1 = Season.winter;  
time2 = Season.summer;  
System.out.println(time1);  
System.out.println(time2.name());  
System.out.println(time1.ordinal());  
System.out.println(time2.getSpan());
```

7.7 Method Design

Once you have identified classes and assigned basic responsibilities, the design of each method will determine how exactly the class will define its behaviors. Some methods are straightforward and require little thought. Others are more interesting and require careful planning.

An *algorithm* is a step-by-step process for solving a problem. A recipe is an example of an algorithm. Travel directions are another example of an algorithm. Every method implements an algorithm that determines how that method accomplishes its goals.

An algorithm is often described using *pseudocode*, which is a mixture of code statements and English phrases. Pseudocode provides enough structure to show how the code will operate, without getting bogged down in the syntactic details of a particular programming language or becoming prematurely constrained by the characteristics of particular programming constructs.

This section discusses two important aspects of program design at the method level: method decomposition and the implications of passing objects as parameters.

Method Decomposition


Occasionally, a service that an object provides is so complicated that it cannot reasonably be implemented using one method. Therefore, we sometimes need to decompose a method into multiple methods to create a more understandable design. As an example, let's examine a program that translates English sentences into Pig Latin.

Key Concept

A complex service provided by an object can be decomposed to make use of private support methods.

Pig Latin is a made-up language in which each word of a sentence is modified, in general, by moving the initial sound of the word to the end and adding an “ay” sound. For example, the word *happy* would be written and pronounced *appyhay* and the word *birthday* would become *irthdaybay*. Words that begin with vowels simply have a “yay” sound added on the end, turning the word *enough* into *enoughyay*.

Consonant blends such as “ch” and “st” at the beginning of a word are moved to the end together before adding the “ay” sound. Therefore, the word *grapefruit* becomes *apefruitgray*.

The `PigLatin` program shown in [Listing 7.13](#)  reads one or more sentences, translating each into Pig Latin.

Listing 7.13

```
//*****

//  PigLatin.java      Author: Lewis/Loftus
//
//  Demonstrates the concept of method decomposition.
//*****

import java.util.Scanner;

public class PigLatin
{
    //-----

    //  Reads sentences and translates them into Pig Latin.
    //-----

    public static void main(String[] args)
    {
        String sentence, result, another;

        Scanner scan = new Scanner(System.in);

        do
        {
            System.out.println();

            System.out.println("Enter a sentence (no
punctuation):");
```

```

        sentence = scan.nextLine();

        System.out.println();

        result = PigLatinTranslator.translate(sentence);

        System.out.println("That sentence in Pig Latin is:");

        System.out.println(result);

        System.out.println();

        System.out.print("Translate another sentence (y/n)?
");

        another = scan.nextLine();

    }

    while (another.equalsIgnoreCase("y"));

}
}

```

Output

```

Enter a sentence (no punctuation):
Do you speak Pig Latin
That sentence in Pig Latin is:
oday ouyay eakspay igpay atinlay

Translate another sentence (y/n)? y


Enter a sentence (no punctuation):
Play it again Sam

```

That sentence in Pig Latin is:

```
ayplay ityay againyay amsay
```

Translate another sentence (y/n)? n

The workhorse behind the `PigLatin` program is the `PigLatinTranslator` class, shown in [Listing 7.14](#) . The `PigLatinTranslator` class provides one fundamental service, a static method called `translate`, which accepts a string and translates it into Pig Latin. Note that the `PigLatinTranslator` class does not contain a constructor because none is needed.

Listing 7.14

[illegible]

```

public class PigLatinTranslator
{
    //-----

    // Translates a sentence of words into Pig Latin.
    //-----

    public static String translate(String sentence)
    {
        String result = "";

        sentence = sentence.toLowerCase();

        Scanner scan = new Scanner(sentence);

        while (scan.hasNext())
        {
            result += translateWord(scan.next());
            result += " ";
        }

        return result;
    }

    //-----

    // Translates one word into Pig Latin. If the word begins
    with a

```



```

        // vowel, the suffix "yay" is appended to the word.
    Otherwise,

        // the first letter or two are moved to the end of the
    word,

        // and "ay" is appended.

//-----

-----

    private static String translateWord(String word)
    {
        String result = "";

        if (beginsWithVowel(word))
            result = word + "yay";
        else
            if (beginsWithBlend(word))
                result = word.substring(2) + word.substring(0,2) +
"ay";
            else
                result = word.substring(1) + word.charAt(0) +
"ay";

        return result;
    }

//-----

-----

    // Determines if the specified word begins with a vowel.

//-----

```

```
-----  
private static boolean beginsWithVowel(String word)
```

```
{
```

```
    String vowels = "aeiou";
```

```
    char letter = word.charAt(0);
```

```
    return (vowels.indexOf(letter) != -1);
```

```
}
```

```
-----  
-----
```

```
    // Determines if the specified word begins with a  
particular
```

```
    // two-character consonant blend.
```

```
-----  
-----
```

```
private static boolean beginsWithBlend(String word)
```

```
{
```

```
    return ( word.startsWith("bl") || word.startsWith("sc")
```

```
||
```

```
        word.startsWith("br") || word.startsWith("sh")
```

```
||
```

```
        word.startsWith("ch") || word.startsWith("sk")
```

```
||
```

```
        word.startsWith("cl") || word.startsWith("sl")
```

```
||
```

```
        word.startsWith("cr") || word.startsWith("sn")
```


```

    ||
    word.startsWith("dr") || word.startsWith("sm")
    ||
    word.startsWith("dw") || word.startsWith("sp")
    ||
    word.startsWith("fl") || word.startsWith("sq")
    ||
    word.startsWith("fr") || word.startsWith("st")
    ||
    word.startsWith("gl") || word.startsWith("sw")
    ||
    word.startsWith("gr") || word.startsWith("th")
    ||
    word.startsWith("kl") || word.startsWith("tr")
    ||
    word.startsWith("ph") || word.startsWith("tw")
    ||
    word.startsWith("pl") || word.startsWith("wh")
    ||
    word.startsWith("pr") || word.startsWith("wr")
);
}
}

```

The act of translating an entire sentence into Pig Latin is not trivial. If written in one big method, it would be very long and difficult to follow. A better solution, as implemented in the `PigLatinTranslator`

class, is to decompose the `translate` method and use several other support methods to help with the task.

The `translate` method uses a `Scanner` object to separate the string into words. Recall that one role of the `Scanner` class (discussed in [Chapter 3](#) ) is to separate a string into smaller elements called tokens. In this case, the tokens are separated by space characters so we can use the default white space delimiters. The `PigLatin` program assumes that no punctuation is included in the input.


The `translate` method passes each word to the private support method `translateWord`. Even the job of translating one word is somewhat involved, so the `translateWord` method makes use of two other private methods, `beginsWithVowel` and `beginsWithBlend`.

The `beginsWithVowel` method returns a `boolean` value that indicates whether the word passed as a parameter begins with a vowel. Note that instead of checking each vowel separately, the code for this method declares a string that contains all the vowels, and then invokes the `String` method `indexOf` to determine whether the first character of the word is in the vowel string. If the specified character cannot be found, the `indexOf` method returns a value of `-1`.

The `beginsWithBlend` method also returns a `boolean` value. The body of the method contains only a `return` statement with one large expression that makes several calls to the `startsWith` method of the

`String` class. If any of these calls returns true, then the `beginsWithBlend` method returns true as well.

Note that the `translateWord`, `beginsWithVowel`, and `beginsWithBlend` methods are all declared with private visibility. They are not intended to provide services directly to clients outside the class. Instead, they exist to help the `translate` method, which is the only true service method in this class, to do its job. By declaring them with private visibility, they cannot be invoked from outside this class. If the `main` method of the `PigLatin` class attempted to invoke the `translateWord` method, for instance, the compiler would issue an error message.

Figure 7.4  shows a UML class diagram for the `PigLatin` program. Note the notation showing the visibility of various methods.

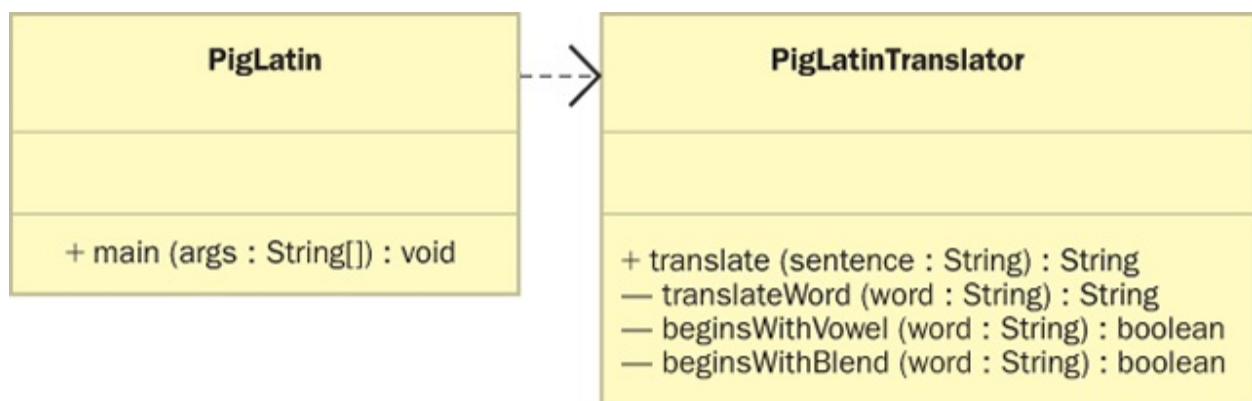


Figure 7.4 A UML class diagram for the `PigLatin` program

Whenever a method becomes large or complex, we should consider decomposing it into multiple methods to create a more understandable

class design. First, however, we must consider how other classes and objects can be defined to create better overall system design. In an object-oriented design, method decomposition must be subordinate to object decomposition.

Method Parameters Revisited

Another important issue related to method design involves the way parameters are passed into a method. In Java, all parameters are passed *by value*. That is, the current value of the actual parameter (in the invocation) is copied into the formal parameter in the method header. We mentioned this issue in [Chapter 4](#); let's examine it now in more detail.


Essentially, parameter passing is like an assignment statement, assigning to the formal parameter a copy of the value stored in the actual parameter. This issue must be considered when making changes to a formal parameter inside a method. The formal parameter is a separate copy of the value that is passed in, so any changes made to it have no effect on the actual parameter. After control returns to the calling method, the actual parameter will have the same value as it did before the method was called.

However, when we pass an object to a method, we are actually passing a reference to that object. The value that gets copied is the address of the object. Therefore, the formal parameter and the actual parameter become aliases of each other. If we change the state of the

object through the formal parameter reference inside the method, we are changing the object referenced by the actual parameter, because they refer to the same object. On the other hand, if we change the formal parameter reference itself (to make it point to a new object, for instance), we have not changed the fact that the actual parameter still refers to the original object.

Key Concept

When an object is passed to a method, the actual and formal parameters become aliases.

The program in [Listing 7.15](#)  illustrates the nuances of parameter passing. Carefully trace the processing of this program and note the values that are output. The `ParameterTester` class contains a `main` method that calls the `changeValues` method in a `ParameterModifier` object. Two of the parameters to `changeValues` are `Num` objects, each of which simply stores an integer value. The other parameter is a primitive integer value.

Listing 7.15

```
//*****
```

```

// ParameterTester.java      Author: Lewis/Loftus
//
// Demonstrates the effects of passing various types of
// parameters.
//*****

public class ParameterTester
{
    //-----
    -----
    // Sets up three variables (one primitive and two objects)
    to
    // serve as actual parameters to the changeValues method.
    Prints
    // their values before and after calling the method.
    //-----
    -----
    public static void main(String[] args)
    {
        ParameterModifier modifier = new ParameterModifier();

        int a1 = 111;
        Num a2 = new Num(222);
        Num a3 = new Num(333);

        System.out.println("Before calling changeValues:");
        System.out.println("a1\ta2\ta3");
    }
}

```



```

        System.out.println(a1 + "\t" + a2 + "\t" + a3 + "\n");

        modifier.changeValues(a1, a2, a3);

        System.out.println("After calling changeValues:");
        System.out.println("a1\ta2\ta3");
        System.out.println(a1 + "\t" + a2 + "\t" + a3 + "\n");
    }
}

```

Output

Before calling changeValues

a1	a2	a3
111	222	333

Before changing the values:



f1	f2	f3
111	222	333

After changing the values:

f1	f2	f3
999	888	777

After calling changeValues:

a1	a2	a3
111	888	333

Listing 7.16  shows the `ParameterModifier` class and **Listing 7.17**  shows the `Num` class. Inside the `changeValues` method, a modification is made to each of the three formal parameters: the integer parameter is set to a different value, the value stored in the first `Num` parameter is changed using its `setValue` method, and a new `Num` object is created and assigned to the second `Num` parameter. These changes are reflected in the output printed at the end of the `changeValues` method.

Listing 7.16

```
//*****

//  ParameterModifier.java      Author: Lewis/Loftus
//
//  Demonstrates the effects of changing parameter values.
//*****

public class ParameterModifier
{
    //-----

    //  Modifies the parameters, printing their values before
    and
    //  after making the changes.
```

```

//-----
-----

public void changeValues(int f1, Num f2, Num f3)
{
    System.out.println("Before changing the values:");
    System.out.println("f1\tf2\tf3");
    System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");

    f1 = 999;
    f2.setValue(888);
    f3 = new Num(777);

    System.out.println("After changing the values:");
    System.out.println("f1\tf2\tf3");
    System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");
}
}

```

Listing 7.17

```

//*****

//  Num.java      Author: Lewis/Loftus
//
//  Represents a single integer as an object.
//*****

```

```
public class Num
{
    private int value;

    //-----
    // Sets up the new Num object, storing an initial value.
    //-----
    public Num(int update)
    {
        value = update;
    }

    //-----
    // Sets the stored value to the newly specified value.
    //-----
    public void setValue(int update)
    {
        value = update;
    }

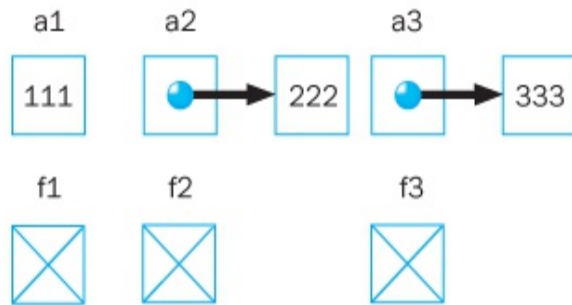
    //-----
    // Returns the stored integer value as a string.
```

```
//-----  
-----  
public String toString()  
{  
    return value + "";  
}  
}
```

However, note the final values that are printed after returning from the method. The primitive integer was not changed from its original value, because the change was made to a copy inside the method. Likewise, the last parameter still refers to its original object with its original value. This is because the new `Num` object created in the method was referred to only by the formal parameter. When the method returned, that formal parameter was destroyed and the `Num` object it referred to was marked for garbage collection. The only change that is “permanent” is the change made to the state of the second parameter. **Figure 7.5**  shows the step-by-step processing of this program.

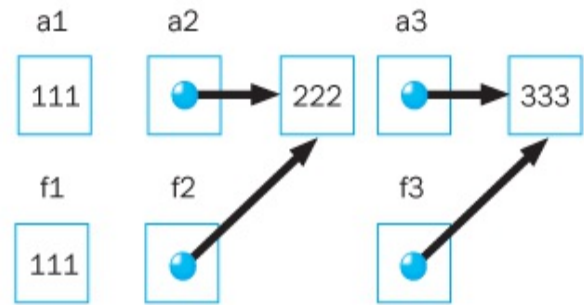
STEP 1

Before invoking changeValues



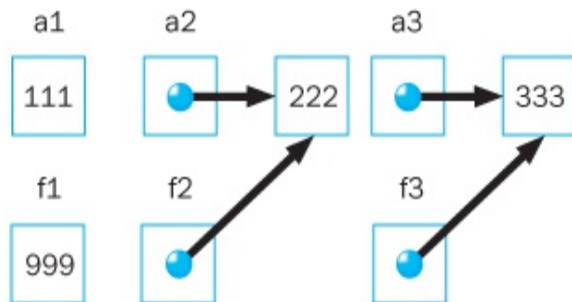
STEP 2

tester.changeValues (a1, a2, a3);



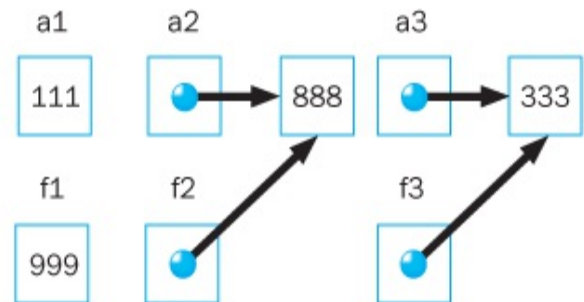
STEP 3

f1 = 999;



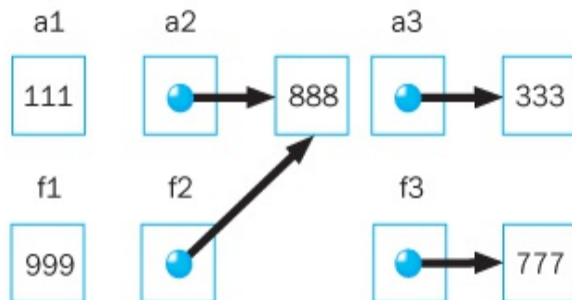
STEP 4

f2.setValue (888);



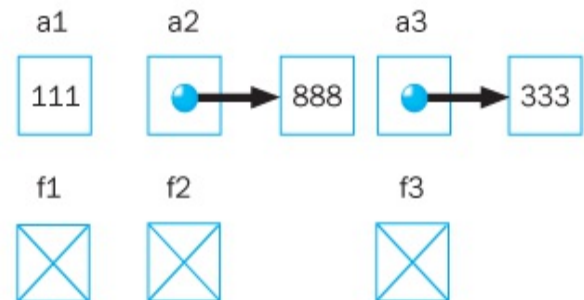
STEP 5

f3 = new Num (777);



STEP 6

After returning from changeValues



 = Undefined

Figure 7.5 Tracing the parameters in the `ParameterTesting` program

Self-Review Questions

(see answers in [Appendix L](#) )

SR 7.18 What is method decomposition?

SR 7.19 Answer the following questions about the `PigLatinTranslator` class.

- a. No constructor is defined. Why not?
- b. Some of the defined methods are private. Why?
- c. A `Scanner` object is declared in the `translate` method. What is it used to scan?

SR 7.20 Identify the resultant sequence of calls/returns of `PigLatinTranslator` support methods when `translate` is invoked with the following actual parameters.

- a. `"animal"`
- b. `"hello"`
- c. `"We are the champions"`

SR 7.21 How are objects passed as parameters?

7.8 Method Overloading

As we've discussed, when a method is invoked, the flow of control transfers to the code that defines the method. After the method has been executed, control returns to the location of the call and processing continues.

Often the method name is sufficient to indicate which method is being called by a specific invocation. But in Java, as in other object-oriented languages, you can use the same method name with different parameter lists for multiple methods. This technique is called **method overloading** ⓘ. It is useful when you need to perform similar methods on different types of data.

The compiler must still be able to associate each invocation to a specific method declaration. If the method name for two or more methods is the same, additional information is used to uniquely identify the version that is being invoked. In Java, a method name can be used for multiple methods as long as the number of parameters, the types of those parameters, and/or the order of the types of parameters is distinct.

Key Concept

The versions of an overloaded method are distinguished by the number, type, and order of their parameters.

For example, we could declare a method called `sum` as follows:

```
public int sum(int num1, int num2)
{
    return num1 + num2;
}
```

Then we could declare another method called `sum`, within the same class, as follows:

```
public int sum(int num1, int num2, int num3)
{
    return num1 + num2 + num3;
}
```

Now, when an invocation is made, the compiler looks at the number of parameters to determine which version of the `sum` method to call. For instance, the following invocation will call the second version of the `sum` method:

```
sum(25, 69, 13);
```

A method's name, along with the number, type, and order of its parameters, is called the method's **signature** ⓘ. The compiler uses the complete method signature to *bind* a method invocation to the appropriate definition.

The compiler must be able to examine a method invocation to determine which specific method is being invoked. If you attempt to specify two method names with the same signature, the compiler will issue an appropriate error message and will not create an executable program. There can be no ambiguity.

Note that the return type of a method is not part of the method signature. That is, two overloaded methods cannot differ only by their return type. This is because the value returned by a method can be ignored by the invocation. The compiler would not be able to distinguish which version of an overloaded method is being referenced in such situations.



Examples of method overloading.

The `println` method is an example of a method that is overloaded several times, each accepting a single type. The following is a partial list of its various signatures:

- `println(String s)`
- `println(int i)`
- `println(double d)`
- `println(char c)`
- `println(boolean b)`

The following two lines of code actually invoke different methods that have the same name:

```
System.out.println("Number of students: ");  
System.out.println(count);
```

The first line invokes the version of `println` that accepts a string. The second line, assuming `count` is an integer variable, invokes the version of `println` that accepts an integer.

We often use a `println` statement that prints several distinct types, such as

```
System.out.println("Number of students: " + count);
```

Remember, in this case the plus sign is the string concatenation operator. First, the value in the variable `count` is converted to a string representation, then the two strings are concatenated into one longer string, and finally the definition of `println` that accepts a single string is invoked.

Constructors can be overloaded, and often are. By providing multiple versions of a constructor, we provide multiple ways to set up an object.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 7.22 How are overloaded methods distinguished from each other?

SR 7.23 For each of the following pairs of method headers, state whether or not the signatures are distinct. If not, explain why not.

a. `String describe(String name, int count)`
`String describe(int count, String name)`

b. `void count()`
`int count()`

c. `int howMany(int compareValue)`
`int howMany(int ceiling)`

d. `boolean greater(int value1)`
`boolean greater(int value1, int value2)`

SR 7.24 The `Num` class is defined in [Section 7.7](#). Overload the constructor of that class by defining a second constructor which takes no parameters and sets the `value` attribute to zero.

7.9 Testing

The term *testing* can be applied in many ways to software development. Testing certainly includes its traditional definition: the act of running a completed program with various inputs to discover problems. But it also includes any evaluation that is performed by human or machine to assess the quality of the evolving system. These evaluations should occur long before a single line of code is written.

The goal of testing is to find errors. By finding errors and fixing them, we improve the quality of our program. It's likely that later on someone else will find any errors that remain hidden during development. The earlier the errors are found, the easier and cheaper they are to fix. Taking the time to uncover problems as early as possible is almost always worth the effort.

Running a program with specific input and producing the correct results establishes only that the program works for that particular input. As more and more test cases execute without revealing errors, our confidence in the program rises, but we can never really be sure that all errors have been eliminated. There could always be another error still undiscovered. Because of that, it is important to thoroughly test a program in as many ways as possible and with well-designed test cases.

Key Concept

Testing a program can never guarantee the absence of errors.

It is possible to prove that a program is correct, but that technique is enormously complex for large systems, and errors can be made in the proof itself. Therefore, we generally rely on testing to determine the quality of a program.

After determining that an error exists, we determine the cause of the error and fix it. After a problem is fixed, we should run previous tests again to make sure that while fixing the problem we didn't create another. This technique is called *regression testing*.

Reviews

One technique used to evaluate design or code is called a **review** ⓘ, which is a meeting in which several people carefully examine a design document or section of code. Presenting our design or code to others causes us to think more carefully about it and permits others to share their suggestions with us. The participants discuss its merits and problems, and create a list of issues that must be addressed. The goal of a review is to identify problems, not to solve them, which usually takes much more time.

A design review should determine whether the requirements have been addressed. It should also assess the way the system is decomposed into classes and objects. A code review should determine how faithfully the design satisfies the requirements and how faithfully the implementation represents the design. It should identify any specific problems that would cause the design or the implementation to fail in its responsibilities.

Sometimes a review is called a *walkthrough*, because its goal is to step carefully through a document and evaluate each section.

Defect Testing

Since the goal of testing is to find errors, it is often referred to as **defect testing** ⓘ. With that goal in mind, a good test is one that uncovers any deficiencies in a program. This might seem strange, because we ultimately don't want to have problems in our system. But keep in mind that errors almost certainly exist. Our testing efforts should make every attempt to find them. We want to increase the reliability of our program by finding and fixing the errors that exist, rather than letting users discover them.

Key Concept

A good test is one that uncovers an error.

A **test case** ⓘ consists of a set of inputs, user actions, or other initial conditions, along with the expected output. A test case should be appropriately documented so that it can be repeated later as needed. Developers often create a complete *test suite*, which is a set of test cases that covers various aspects of the system.

Key Concept

It is not feasible to exhaustively test a program for all possible input and user actions.

Because programs operate on a large number of possible inputs, it is not feasible to create test cases for all possible input or user actions. Nor is it usually necessary to test every single situation. Two specific test cases may be so similar that they actually do not test unique aspects of the program. To do both would be a wasted effort. We'd rather execute a test case that stresses the program in some new way. Therefore, we want to choose our test cases carefully. To that end, let's examine two approaches to defect testing: black-box testing and white-box testing.

As the name implies, *black-box testing* treats the thing being tested as a black box. In black-box testing, test cases are developed without regard to the internal workings. Black-box tests are based on inputs and outputs. An entire program can be tested using a black-box

technique, in which case the inputs are the user-provided information and user actions such as button pushes. A test case is successful only if the input produces the expected output. A single class can also be tested using a black-box technique, which focuses on the system interface (its public methods) of the class. Certain parameters are passed in, producing certain results. Black-box test cases are often derived directly from the requirements of the system or from the stated purpose of a method.

The input data for a black-box test case are often selected by defining equivalence categories. An **equivalence category** ⓘ is a collection of inputs that are expected to produce similar outputs. Generally, if a method will work for one value in the equivalence category, we have every reason to believe it will work for the others. For example, the input to a method that computes the square root of an integer can be divided into two equivalence categories: nonnegative integers and negative integers. If it works appropriately for one nonnegative value, it will likely work for all nonnegative values. Likewise, if it works appropriately for one negative value, it will likely work for all negative values.

Equivalence categories have defined boundaries. Because all values of an equivalence category essentially test the same features of a program, only one test case inside the equivalence boundary is needed. However, because programming often produces “off by one” errors, the values on and around the boundary should be tested exhaustively. For an integer boundary, a good test suite would include at least the exact value of the boundary, the boundary minus 1, and

the boundary plus 1. Test cases that use these cases, plus at least one from within the general field of the category, should be defined.

Let's look at an example. Consider a method whose purpose is to validate that a particular integer value is in the range 0 to 99, inclusive. There are three equivalence categories in this case: values below 0, values in the range of 0 to 99, and values above 99. Black-box testing dictates that we use test values that surround and fall on the boundaries, as well as some general values from the equivalence categories. Therefore, a set of black-box test cases for this situation might be: -500, -1, 0, 1, 50, 98, 99, 100, and 500.

White-box testing ⓘ, also known as *glass-box testing*, exercises the internal structure and implementation of a method. A white-box test case is based on the logic of the code. The goal is to ensure that every path through a program is executed at least once. A white-box test maps the possible paths through the code and ensures that the test cases cause every path to be executed. This type of testing is often called **statement coverage** ⓘ.

Paths through code are controlled by various control flow statements that use conditional expressions, such as `if` statements. In order to have every path through the program executed at least once, the input data values for the test cases need to control the values for the conditional expressions. The input data of one or more test cases should cause the condition of an `if` statement to evaluate to `true` in at least one case and to `false` in at least one case. Covering both true and false values in an `if` statement guarantees that both the

paths through the `if` statement will be executed. Similar situations can be created for loops and other constructs.

In both black-box and white-box testing, the expected output for each test should be established prior to running the test. It's too easy to be persuaded that the results of a test are appropriate if you haven't first carefully determined what the results should be.

Self-Review Question

(see answer in [Appendix L](#) )

SR 7.25 Select the term from the following list that best matches each of the following phrases:

black-box, defects, regression, review, test case, test suite, walkthrough, white-box

- a. Running previous test cases after a change is made to a program to help ensure that the change did not introduce an error.
- b. A meeting in which several people collectively evaluate an artifact.
- c. A review that steps carefully through a document, evaluating each section.
- d. The goal of testing is to discover these.
- e. A description of the input and corresponding expected output of a code unit being tested.
- f. A set of test cases that covers various aspects of a system.

- g. With this testing approach, test cases are based solely on requirement specifications.
- h. With this testing approach, test cases are based on the internal workings of the program.