Review of object orientation

As we mentioned in the last chapter, software engineers must have a good understanding of the computing technology with which they work. In this chapter, we review an important area of that technology: object-oriented programming.

It is our expectation that most readers will have learned the basics of object-oriented programming in Java before reading this book.

If you do not know Java, but know another object-oriented language, such as C++, C#, Delphi or Smalltalk, then the exercises at the end of this chapter will help you make the transition to Java. We recommend you also make use of Java learning resources, some of which we list at the end of the chapter.

Our goal with the use of Java in this book is to give you practical illustrations of software engineering concepts.

In this chapter you will learn about the following
- The basic principles of object orientation.
- Classes and objects.
- Instance variables, attributes and associations.
- Methods, operations and polymorphism.
- Organizing classes into inheritance hierarchies.
- Evaluating alternative implementations of simple designs in Java.

2.1 What is object orientation?

Object-oriented systems make use of abstraction in order to help make software less complex. An abstraction is something that relieves you from having to deal with details. Object-oriented systems combine procedural abstraction with data abstraction. To help you better understand what this means, we will first take a look at these two types of abstraction.
Review of object orientation

Procedural abstraction and the procedural paradigm

From the earliest days of programming, software has been organized around the notion of procedures (also in some contexts called functions or routines). These provide procedural abstraction. When using a certain procedure, a programmer does not need to worry about all the details of how it performs its computations; he or she only needs to know how to call it and what it computes. The programmer’s view of the system is thus made simpler.

In the so-called procedural paradigm, the entire system is organized into a set of procedures. One ‘main’ procedure calls several other procedures, which in turn call others.

The procedural paradigm works very well when the main purpose of programs is to perform calculations with relatively simple data. However, as computers and applications have become more complex, so has the data. Systems written using the procedural paradigm are complex if each procedure works with many types of data, or if each type of data has many different procedures that access and modify it.

Data abstraction

Data abstractions can help reduce some of a system’s complexity. Records and structures were the first data abstractions to be introduced. The idea is to group together the pieces of data that describe some entity, so that programmers can manipulate that data as a unit.

However, even when using data abstraction, programmers still have to write complex code in many different places. Consider, for example, a banking system that is written using the procedural paradigm, but using records representing bank accounts. The software has to manage accounts of different types, such as checking, savings and mortgage accounts (a checking account would be called a cheque account or current account in some countries). Each type of account will have different rules for the computation of fees, interest, etc. Such a system would have procedures like the following pseudocode in many different places:

```plaintext
if account is of type checking then
    do something
else if account is of type savings then
    do something else
else
    do yet another thing
endif
```

Imagine also that clients can hold several accounts of different types, and some accounts can be held jointly; also the different account holders might have different rights. Rules to deal with issues like these would be scattered throughout the code, making change very difficult.
The object-oriented paradigm: organizing procedural abstractions in the context of data abstractions

Starting in the late 1960s, programmers began to see the advantage of organizing programs around data abstractions. They realized that they could make systems much simpler by putting all the procedures that access or modify a particular class of objects in one place, rather than having the procedures spread out all over the system. This idea is the root of the object-oriented (OO) paradigm which, by the 1990s, had become accepted as the best way to organize most systems.

In the object-oriented paradigm, a running program can be seen as a collection of objects collaborating to perform a given task.

Figure 2.1 summarizes the essential difference between the object-oriented and procedural paradigms. In the procedural paradigm (shown on the left), the code is organized into procedures that each manipulate different types of data. In the object-oriented paradigm (shown on the right), the code is organized into classes that each contain procedures for manipulating instances of that class alone. Later on, we will explain how the classes themselves can be organized into hierarchies that provide even more abstraction.

**Definition:** The object-oriented paradigm is an approach to the solution of problems in which all computations are performed in the context of objects. The objects are instances of programming constructs, normally called classes, which are data abstractions and which contain procedural abstractions that operate on the objects.

In the object-oriented paradigm, a running program can be seen as a collection of objects collaborating to perform a given task.

Figure 2.1 summarizes the essential difference between the object-oriented and procedural paradigms. In the procedural paradigm (shown on the left), the code is organized into procedures that each manipulate different types of data. In the object-oriented paradigm (shown on the right), the code is organized into classes that each contain procedures for manipulating instances of that class alone. Later on, we will explain how the classes themselves can be organized into hierarchies that provide even more abstraction.

**Figure 2.1** Organizing a system according to the procedural paradigm (left) or the object-oriented paradigm (right). The UML notation used in the right-hand diagram will be discussed in more detail later.

### 2.2 Classes and objects

Classes and objects are the aspects of object orientation that people normally think about first. In this section, we will define in more detail what we mean by these two terms.
Chapter 2
Review of object orientation

Objects

An object is a chunk of structured data in a running software system. It can represent anything with which you can associate properties and behavior. Properties characterize the object, describing its current state. Behavior is the way an object acts and reacts, possibly changing its state.

Figure 2.2 shows some of the objects and their properties that might be important to a particular banking system. The notation used in Figure 2.2 to represent objects is UML. We will show you some very simple UML notation in this chapter; we will explain it in more detail in Chapter 5 and subsequent chapters.

Figure 2.2 Several objects in a banking application

The following are some other examples of objects:

- In a payroll program, there would be objects representing each individual employee.
- In a university registration program, there would be objects representing each student, each course and each faculty member.
- In a factory automation system, there might be objects representing each assembly line, each robot, each item being manufactured, and each type of product.

In the above examples, all the objects represent things that are important to the users of the program. You use a process often called object-oriented analysis to decide which objects will be important to the users, and to work out the structure, relationships and behavior of these objects.

When performing object-oriented analysis, you do not initially need to understand how objects are physically represented using a particular programming language, nor whether they are stored in random-access memory or on disk. It is best to leave consideration of such issues until you
have completed object-oriented analysis, and moved on to *object-oriented design* (OOD). We will discuss object-oriented analysis and design in detail starting in Chapter 5.

**Classes and their instances**

Classes are the units of data abstraction in an object-oriented program. More specifically, a class is a software module that represents and defines a set of similar objects, its *instances*. All the objects with the same properties and behavior are instances of one class.

For example, Figure 2.3 shows how the bank employees Jane and Margaret from Figure 2.2 can be represented as instances of a single class `Employee`. Class `Employee` declares that all its instances have a `name`, a `dateOfBirth`, an `address` and a `position`.

As a software module, a class contains all of the code that relates to its objects, including:

- Code describing how the objects of the class are structured – i.e. the data stored in each object that implement the properties.
- The procedures, called *methods*, that implement the behavior of the objects.

In other words, in addition to defining properties such as `name` and `address`, as shown in Figure 2.3, an `Employee` class would also provide methods for creating a new employee, and changing an employee’s name, address and position. We will talk more about the contents of a class in Sections 2.3 and 2.4.

Sometimes it is hard for beginners to decide what should be a class and what should be an instance. The following two rules can help:

- In general, something should be a class if it could have instances.
- In general, something should be an instance if it is clearly a single member of the set defined by a class.

For example, in an application for managing hospitals, one of the classes might be `Doctor`, and another might be `Hospital`. You might think that `Hospital` should be an instance if there is only one of them in the system; however, the fact that in theory there could be multiple hospitals tells us that `Hospital` should be a class.
Chapter 2

Review of object orientation

Example 2.1
In the following, we indicate whether each item should be a class or an instance. If it should be a class, we describe its instances. If it should be an instance, we describe its class.

Film: class; instances include 'Star Wars' and 'Casablanca'.
Reel of film: class; instances are physical reels.
Film reel with serial number SW19876: instance of ReelOfFilm.
Showing of 'Star Wars' in the Phoenix Cinema at 7 pm: instance of class ShowingOfFilm.

Exercise

E4 Which of the following items do you think should be a class, and which should be an instance? For any item that should be an instance, name a suitable class for it. If you think an item could be either a class or an instance, depending on circumstances, explain why.

(a) General Motors
(b) Automobile company
(c) Boeing 777
(d) Computer science student
(e) Mary Smith
(f) Game
(g) Board game
(h) Chess
(i) University course SEG 2100
(j) Airplane
(k) The game of chess between Tom and Jane which started at 2:30 pm yesterday.
(l) The car with serial number JM 198765T4

Naming classes

One of the first challenges in any object-oriented project is to name the classes. Notice that the class names mentioned in the last subsection such as Employee, Hospital and Doctor are nouns, have their first letter capitalized and are written in the singular. These are important conventions that should be followed in all object-oriented programs in languages like Java and C++. Being consistent about capitalization ensures that readers of the program can tell what is a class and what is not. Using the singular ensures that readers can tell that an instance of the class is a single item, not a list or collection. If you want to give a class a name consisting of more than one word, then omit the spaces and capitalize the first letter of each word, for example: PartTimeEmployee.

It is also important to choose names for classes that are neither too general nor too specific. Many words in the English language have more than one meaning, or are used with a broad meaning. For example, the word ‘bus’ could mean the physical vehicle, or a particular run along a particular route, as in, 'I will catch the 10:30 bus (but I don't care which vehicle is used). You might choose to call
the class that represents physical vehicles \texttt{BusVehicle} and the class that represents runs along a route \texttt{BusRouteRun}. Sometimes it is possible to be too specific in naming a class: for example, when filling out a form, you may be asked to specify the ‘city’ as part of an address. But not everybody lives in a city! Therefore, rather than creating a class called \texttt{City} to store, for example, a person’s place of birth, you should perhaps use the more general class name \texttt{Municipality}.

Another principle is to name classes after the things their instances represent in the real world. Unless you are dealing with low-level system design, you should avoid using words in class names that reflect the internals of a computer system such as ‘Record’, ‘Table’, ‘Data’, ‘Structure’, or ‘Information’. For example, a class named \texttt{Employee} would be acceptable, but one named \texttt{EmployeeData} would not.

### Exercises

**E5** Some of the following are not good names for classes in the scheduling system of a passenger rail company. For each name, indicate whether it is a bad class name, and if so, explain why and suggest a better name or names:

(a) \texttt{Train}

(b) \texttt{Stop}

(c) \texttt{SleepingCarData}
Identify all the classes you can think of that might be part of the following systems, and choose good names for them.

(a) A restaurant reservation system.
(b) A video rental store.
(c) A weather forecasting system.
(d) A video editing tool.

2.3 Instance variables

A variable is a place where you can put data. Each class declares a list of variables corresponding to data that will be present in each instance; such variables are called instance variables.

Attributes and associations

There are two groups of instance variables, those used to implement attributes, and those used to implement associations.

An attribute is a simple piece of data used to represent the properties of an object. For example, each instance of class Employee might have the following attributes:

- name
- dateOfBirth
- socialSecurityNumber
- telephoneNumber
- address

An association represents the relationship between instances of one class and instances of another. For example, class Employee in a business application might have the following relationships:

- supervisor (association to class Manager)
- tasksToDo (association to class Task)

We will talk about selecting and representing attributes and associations in much more detail in Chapter 5.
Variables versus objects

One common source of confusion when discussing object-oriented programs is the difference between variables and objects. These are quite distinct concepts.

At any given instant, a variable can refer to a particular object or to no object at all. Variables that refer to objects are therefore often called references. During the execution of a program, a given variable may refer to different objects. Furthermore, an object can be referred to by several different variables at the same time.

The type of a variable determines what classes of objects it may contain. We will explain the rules regarding this in later sections.

Variables can be local variables in methods; these are created when a method runs and are destroyed when a method returns. However, objects temporarily referenced by such variables may last much longer than the lifetime of the method as long as some other variable also references the object.

Exercises

E7 Identify the attributes that might be present in the following classes. Try to be reasonably exhaustive.

(a) Series (in a scheduling system for an independent television station)
(b) Passenger (in an airline system)
(c) Event (in a personal schedule system; a meeting might be a kind of event)
(d) Classroom (in a university course scheduling system)
(e) PhoneCall (in the system of a mobile telephone company)
(f) AssemblyLine (in a factory automation system)

E8 Identify some associations that might involve the classes listed in the previous exercise. For each association, indicate the other class that would be involved.

Instance variables versus class variables

If you declare that a class has an instance variable called var, then you are saying that each instance of the class will have its own slot named var. Therefore, for example, each Employee has a supervisor. The actual data put into these variables will vary from object to object: employees will have different instances of Manager as their supervisors.

Sometimes, however, you want to create a variable whose value is shared by all instances of a class. Such a variable is known as a class variable or static variable. If one instance sets the value of a class variable, then all the other instances see the same changed value.
Chapter 2
Review of object orientation

Class variables are often overused by beginners in cases when they should use instance variables. Class variables are, however, useful for storing the following types of information:

- Default or ‘constant’ values that are widely used by methods in a class.
- Lookup tables and similar structures used by algorithms inside a particular class.

2.4 Methods, operations and polymorphism

The word ‘method’ is used in object-oriented programs where the words ‘procedure’, ‘function’ or ‘routine’ might be used in other programs. Methods are procedural abstractions used to implement the behavior of a class.

An operation is a higher-level procedural abstraction. It is used to discuss and specify a type of behavior, independently of any code that implements that behavior. Several different classes can have methods with the same name that implement the abstract operation in ways suitable to each class. The word ‘method’ is used because in English it means ‘way of performing an operation’.

We call an operation polymorphic, if the running program decides, every time an operation is called, which of several identically named methods to invoke. The program makes its decision based on the class of the object in a particular variable. Polymorphism is one of the fundamental features of the object-oriented paradigm.

Definition: polymorphism is a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes.

As an illustration of polymorphism, imagine a banking application that has an abstract operation calculateInterest. In some types of account, interest is computed as a percentage of the average daily balance during a month. In other types of account, interest is computed as a percentage of the minimum daily balance during a month. In a mortgage account, to which you can only deposit (make a payment) but from which you cannot withdraw except initially, interest may be computed as a percentage of the balance at the end of the month.
In the banking system, the three classes `CheckingAccount`, `SavingsAccount` and `MortgageAccount` would each have their own method for the polymorphic operation `calculateInterest`. When a program is calculating the interest on a series of accounts, it will invoke the version of `calculateInterest` specific to the class of each account.

Exercise

E9 For each of the following sets of classes, find an appropriate superclass and the polymorphic operations that should be included in this superclass. Explain the way these operations would behave in each subclass and identify some operations that might be present in only one of the subclasses.

(a) `Square`, `Circle`, `Rectangle`
(b) `Truck`, `Ambulance`, `Bus`
(c) `Techician`, `AdministrativeAssistant`, `Manager`

### 2.5 Organizing classes into inheritance hierarchies

If several classes have attributes, associations or operations in common, it is best to avoid duplication by creating a separate superclass that contains these common aspects. Conversely, if you have a complex class, it may be good to divide its functionality among several specialized subclasses.

For example, imagine you are creating a banking application in which there are several kinds of accounts. Some things are common to all accounts, such as having a balance and an owner, as well as being able to deposit money in the account, open it and close it. Other things differentiate the accounts – for example, a mortgage account has a negative balance as well as a property (e.g. a house) as collateral; a savings account might have certain privileges associated with it such as higher interest for keeping a high balance in it. In this example we would say that class `Account` should be the superclass of subclasses `SavingsAccount`, `CheckingAccount` and `MortgageAccount`.

The relationship between a subclass and an immediate superclass is called a generalization. The subclass is called a specialization. A hierarchy with one or more generalizations is called an inheritance hierarchy, a generalization hierarchy or an isa hierarchy. The reason for the latter name will become clear shortly.

You can draw inheritance hierarchies graphically as shown in Figure 2.4. The little triangle symbolizes one or more generalizations sharing the same superclass, and points to the superclass. It is clearest when such diagrams are drawn with the superclass at the top and the subclasses below, although other arrangements are also allowed.
It is also possible to show inheritance hierarchies textually using indentation, like this:

```plaintext
Account
  SavingsAccount
  CheckingAccount
  MortgageAccount
```

You control inheritance by creating an inheritance hierarchy. Once you define which classes are superclasses and which classes are their subclasses, inheritance automatically occurs.

For example, all the features of Account are also present in SavingsAccount, CheckingAccount and MortgageAccount. Figure 2.5 expands on Figure 2.4, showing a variety of attributes and operations possessed by Account that would also be inherited by the three subclasses. Attributes are shown in the middle of the class box; operations are shown at the bottom. The inherited features are not explicitly shown in the subclasses to make the diagram clearer; however, any new features exclusive to each subclass are shown.

Organizing classes into inheritance hierarchies is a key skill in object-oriented design and programming. It is easy to make mistakes and create invalid generalizations. One of the most important rules to adhere to is the isa rule. The isa rule says that class A can only be a valid subclass of class B if it makes sense, in English, to say, ‘an A is a B’. For example it makes sense to say ‘a SavingsAccount is an Account’; it does not make sense to say the inverse, ‘an Account is a SavingsAccount’. You should test all superclass–subclass pairs (generalizations) against the isa rule. It is for this reason that inheritance hierarchies are often called isa hierarchies.

When you detect a violation of the isa rule, it is a clear indication that you have made an invalid generalization. However, not all cases where the isa rule holds are good generalizations. Other important points you should check are:

- If you have given the subclass or superclass ambiguous names (such as ‘Bus’ as described earlier), you will often create bad generalizations.
A subclass must retain its distinctiveness throughout its life. For example, if you decided to create a subclass `OverdrawnAccount`, the isa rule appears to hold: 'An overdrawn account is an account'. However, an overdrawn account will not remain a distinct type of account once enough money is deposited into it. Therefore, this is not a good generalization; in fact, the class `OverdrawnAccount` should not be a separate class.

All the inherited features must make sense in each subclass. In Figure 2.5 you must ensure that each of the three subclasses can have a `balance`, an `openedDate` and a `creditOrOverdraftLimit`. You must also make sure that it makes sense to perform the operations `credit` and `debit` in each subclass, and that all methods of these operations will behave consistently. You may think that `debit` would not apply to `MortgageAccount`; however, remember that when the account is first created, a large debit is made. We will discuss this issue more in the next section.

It is a common mistake for designers to overlook these three checks. If the checks are overlooked, the resulting code then needs many special conditions to deal with unwanted inheritance, and it becomes hard to understand.

Key conclusions we can draw from the above are: generalizations and their resulting inheritance help to avoid duplication and improve reuse; but poorly designed generalizations can actually cause more problems than they solve.

---

**The Liskov Substitution Principle**

The Liskov Substitution Principle says this: if you have a variable whose type is a superclass (e.g. `Account`), then the program should work properly if you place an instance of that superclass or any of its subclasses in the variable. The program using the variable should not be able to tell which class is being used, and should not care.
Chapter 2
Review of object orientation

Example 2.2 Organize the following set of classes into hierarchies: Circle, Point, Rectangle, Matrix, Ellipse, Line, Plane.

Figure 2.6 shows one possible solution – there can often be more than one acceptable answer to this kind of question.

\[ \text{MathematicalObject} \]
\[ \quad \text{Shape} \quad \text{Point} \quad \text{Matrix} \]
\[ \quad \text{Shape2D} \quad \text{Shape3D} \]
\[ \quad \text{Ellipse} \quad \text{Polygon} \quad \text{Line} \quad \text{Plane} \]
\[ \quad \text{Circle} \quad \text{Quadrilateral} \]
\[ \quad \text{Rectangle} \]

The following are some possible changes to Figure 2.6 that can be debated:

- You could consider a Point to be a degenerate Shape. But how many dimensions does it have? A point could have any number of dimensions. Perhaps what is needed is to have separate classes Point2D and Point3D.

- A Line, similarly, can be 2-dimensional or 3-dimensional.

- The fact that Circle is shown as a subclass of Ellipse is interesting. Mathematically, a circle has all the properties of an ellipse. An ellipse has two foci; in a circle these two foci are constrained to be equal to each other – at the center. In an object-oriented system, subclasses must have all the properties of their superclass; in the case of the ellipse, a valid operation is to change one of the foci. This implies we should be able to change one focus of a Circle, which is a bit odd. We could permit this as long as doing so automatically changes the other focus so that the circle remains a circle with one center. However, this solution is not entirely satisfactory since every instance of Circle must still have two attributes to store the foci. An alternative sub-hierarchy, showing a different way of arranging the attributes of circles and ellipses, is shown in Figure 2.7. Yet another option is to get rid of the Circle class entirely and just use the Ellipse class; you might then add a Boolean attribute constrainAsCircle to Ellipse if you wanted certain ellipses to always remain circles.
Exercises

E10 Which of the following would not form good superclass–subclass pairs (generalizations), and why? Hint: look for violations of the isa rule, poor naming, and other problems.

(a) Money – CanadianDollars
(b) Bank – Account
(c) OrganizationUnit – Division
(d) SavingsAccount – CheckingAccount
(e) Account – Account12876
(f) People – Customer
(g) Student – GraduateStudent
(h) Continent – Country
(i) Municipality – Neighborhood

E11 What problems could arise by making Quadrilateral and Rectangle subclasses of Polygon? What alternatives are possible? What are the advantages and disadvantages of each alternative?

E12 Organize each of the following sets of items into inheritance hierarchies of classes. Hints:

- For each set of items, you will have several distinct hierarchies.
- You will need to add additional classes to act as superclasses. You will also need to change some names, and you will discover that two items may correspond to a single class.
- Think of important attributes present in your classes. Make sure that attributes in a superclass will be present in each of its subclasses.
- Remember to use the isa rule.
### Chapter 2
Review of object orientation

#### a) Vehicle
- Car
- Amphibious vehicle
- Airplane
- Electric motor
- Jet engine
- Wheel
- Truck
- Bicycle
- Transmission

#### b) Edition of book
- Copy of book
- Volume
- Magazine
- Work of literature
- Issue of magazine
- Publication
- Chapter
- Author
- Publisher
- Copy of issue of magazine

#### c) Schedule
- Bus
- Trip
- Bus route
- Express bus
- Tour bus
- Route
- Chartered bus
- Unscheduled trip

#### d) Student
- Course
- Professor
- Course section
- Program of studies
- Administrative assistant
- Technician
- Classroom
- Meeting room
- Building
- Registration system
- Gymnasium
- Exam
- Luxury bus
- Route

#### e) Currency
- Exchange rate
- Bank
- Financial instrument
- Debit card
- Check
- Visa
- MasterCard
- Bank account
- Canadian dollars
- US dollars
- Visa

#### f) Hotel room
- Meeting room
- Ballroom
- Suite
- Hilton (the hotel chain)
- Ottawa Hilton
- Meeting organizer
- Catered function
- Booking
- Guest
- Reservation
- Meeting
- Conference
- Conference room
- Item on bill

#### g) Insurance policy
- Claim
- Deductible
- Insurance client
- Insured property
- Automobile policy
- Home policy
- Life insurance
- Beneficiary
- Policy renewal
The effect of inheritance hierarchies on polymorphism and variable declarations

2.6 The effect of inheritance hierarchies on polymorphism and variable declarations

Much of the power of the object-oriented paradigm comes from polymorphism and inheritance working together. In this section we will investigate this synergy.

Figure 2.8 shows an expanded version of the hierarchy of two-dimensional shapes from Figure 2.6, also incorporating the EllipticalShape class from Figure 2.7, as well as a modified Polygon hierarchy. We will use Figure 2.8 to illustrate several important points; you should study it and try to understand it before proceeding.

Figure 2.8 A hierarchy of shapes showing polymorphism and overriding
Figure 2.8 is a four-level hierarchy with four generalizations. The classes at the very bottom of the hierarchy are called leaf classes.

The following explains certain details of some of the classes:

- An **Ellipse** is defined using the lengths of two axes: the longer one is called the *major* axis, and the shorter one the *minor* axis. The *semi-major* axis is half the major axis; in a circle, the semi-major axis and the semi-minor axis are equal to the radius.

- A **RegularPolygon** is any shape whose vertices can be all placed on the circumference of a circle and whose side lengths are equal; for example, an equilateral triangle, square or regular pentagon.

- An **ArbitraryPolygon** is any polygon that is neither a rectangle nor regular. It is defined by a set of points.

Class **Shape2D** lists seven operations. Since this is the ultimate superclass of the hierarchy, these seven operations are all inherited by each of the other eight classes. This means that each operation must *make sense* and *behave consistently* in all the classes. In this example, the various subclasses will use different methods for most operations. We will discuss this further, below.

Three of the operations in **Shape2D** modify the shape. The effect of these operations is illustrated in Figure 2.9.

- **rotate**: takes one argument, the number of degrees to rotate the shape. The shape is modified as a result of the rotation.

- **translate**: takes two arguments, an x-amount and a y-amount and moves the shape in the x- and y-directions.

- **changeScale**: takes one argument, a percentage, and makes the shape bigger or smaller, keeping its center the same.
The \texttt{getCenter} operation simply returns the value of the \texttt{center} instance variable. The \texttt{getArea} and \texttt{getPerimeterLength} operations compute a value and return it. The \texttt{getBoundingRect} operation returns a non-rotated \texttt{Rectangle} that would be just big enough to fit around the shape – this is also illustrated in Figure 2.9.

### Abstract classes and abstract methods

There are separate methods in four different classes to compute the operation \texttt{rotate}. Each method takes advantage of properties unique to its class:

- **Circle**: rotating a circle does not change it! Therefore the \texttt{rotate} method in class \texttt{Circle} would do absolutely nothing. The method would exist but would immediately return.

- **SimplePolygon** and **Ellipse**: these classes have an attribute called \texttt{orientation}, which the \texttt{rotate} method simply has to modify.

- **ArbitraryPolygon**: rotating one of these would be a little more complex. See a textbook on computer graphics to learn precisely how to do it.

However, it is not possible to write a method to rotate instances of the superclasses of these four classes. This is because there is not enough information available in those classes to do the rotation. This leads us to two important conclusions:

1. The \texttt{rotate} operation found in \texttt{Shape2D} is an example of an \textit{abstract operation}. If a class has an abstract operation, it means that no method for that operation exists in the class, although the operation makes logical sense for it and for all the classes below it in the hierarchy. Abstract operations are shown in italics in Figure 2.8. Leaf subclasses have to have or inherited implementations of each operation – in other words, you can have abstract operations anywhere except leaf classes.

2. The four classes, \texttt{Shape2D}, \texttt{EllipticalShape}, \texttt{Polygon} and \texttt{SimplePolygon}, must be \textit{abstract classes}. An abstract class is one that cannot have any instances. Any class, except a leaf class, can be declared abstract; however, a class that has one or more abstract methods \textit{must} be declared abstract. The main purpose of an abstract class is to hold features that will be inherited by its subclasses. If a class is not abstract, then it is said to be \textit{concrete}, and instances of it can be created. Leaf classes must be concrete, although it is also possible to have concrete classes higher in the inheritance hierarchy.

You should also note the following other interesting facts about abstract classes and methods in the shape hierarchy of Figure 2.8:

- In addition to \texttt{rotate}, all but two of the other operations in class \texttt{Shape2D} are abstract. As required, these have concrete implementations by the time the leaf classes are reached. However, the concrete implementations do not actually have to be defined \textit{in} the leaf classes – they can be defined higher in the hierarchy. For example, \texttt{rotate} is defined in the abstract class \texttt{SimplePolygon}. 
Class SimplePolygon is abstract, even though it has two concrete methods. This is because it neither has nor inherits concrete implementations of operations changeScale, getArea and getPerimeterLength.

There is an abstract operation getBoundingRect in class Shape2D of Figure 2.8. It has a concrete implementation in Polygon, since it is possible to design a general algorithm for computing the bounding rectangle if you can compute the vertices of a shape – and class Polygon does have such a method, called getVertices.

Class Polygon declares the operation getVertices, yet the operation does not exist in its superclass Shape2D. This is because it only makes sense to talk about vertices of polygons; no vertices exist in smooth-curved shapes such as ellipses.

Operation getVertices is abstract in Polygon, even though the concrete method getBoundingRect calls it. Such calling of an abstract operation by a concrete method is quite legal and in fact is considered good design practice.

Operation getVertices has concrete implementations in the three leaf classes below Polygon, but not in the immediate subclass SimplePolygon, because there is not enough information to compute the vertices in that class.

The attribute semiMajorAxis is present in EllipticalShape; however, it is not accessed by any method in that class. This is because Circle accesses it using the method name getRadius – it would be odd to be able to talk about the semi-major axis of a circle even though mathematically it is equivalent to the radius.

### Overriding

In addition to the implementation of getBoundingRect in Polygon, there is also another concrete implementation in class Rectangle (which is a subclass of Polygon). This second concrete implementation is said to override the version of getBoundingRect that otherwise would be inherited from Polygon. The getBoundingRect method in Rectangle computes the same result as the method in Polygon, but the overriding version in Rectangle can be more efficient: in those cases where the Rectangle is not rotated, its bounding rectangle is the Rectangle itself.

In general, there are three valid reasons for overriding methods: restriction, extension and optimization:

**Overriding for restriction** occurs when the overriding method prevents a violation of certain constraints that are present in the subclass, but were not present in the superclass.

For example, imagine there was a changeScale(x,y) method in Shape2D that allowed a shape to be distorted by having its width and height modified by different percentages. It would be reasonable to use this method to modify any ArbitraryPolygon, Ellipse or un-rotated Rectangle. However, scaling a Circle in this way would mean that it would no longer be a Circle – it would be an Ellipse.
The effect of inheritance hierarchies on polymorphism and variable declarations

You might therefore consider creating an overriding version of `changeScale(x, y)` in `Circle` which throws an exception if `x` and `y` are not equal to each other. Similarly, non-uniform scaling of a `RegularPolygon` should be forbidden.

As another example of overriding for restriction, imagine adding a concrete version of `debit` in `MortgageAccount` in Figure 2.5 that restricts your ability to withdraw money from the account: `MortgageAccount` might allow you to only withdraw a fixed amount when the account is first opened. Any other attempt to withdraw money would throw an exception.

Overriding for restriction can have some undesirable effects. It is important to ensure that all polymorphic methods implementing an abstract operation behave consistently. For example, if the implementations of `debit` in some classes may throw an exception, while other implementations of `debit` do not declare that they too may throw the exception, then consistency is being violated. The programmer can solve this problem by declaring that the exception may be thrown by any of the polymorphic implementations of `debit`, even though he or she knows that certain of the methods will not in practice do so. Users of the operation must therefore always prepare for the exception (in Java, by using a `try–catch` construct).

- **Overriding for extension** occurs when the overriding method does basically the same thing as the version in the superclass, but adds some extra capability needed in the subclass. For example, in Figure 2.5, there might be a version of `debit` in `SavingsAccount` that would charge an additional fee if your bank balance was less than $1500.

- **Overriding for optimization** occurs when the overriding method in the subclass has exactly the same effect as the overridden method, except that it is more efficient. Above, we described a case of this in which `getBoundingRect` can often be computed more efficiently in the `Rectangle` class than in the general case.

Exercises

**E13** This question requires knowledge of very basic geometry. Describe in one paragraph how the different polymorphic implementations of the following operations from Figure 2.8 would work in classes `Rectangle`, `RegularPolygon`, `Circle` and their superclasses. You do not need to write any code; instead just describe what attributes would be used and/or modified, and the formula to be used (if any).

(a) `translate`

(b) `changeScale`

(c) `getArea`

(d) `getCenter`
Chapter 2
Review of object orientation

E14 Explain how you would incorporate the operations flipHorizontally and flipVertically into the hierarchy of Figure 2.8. Describe which classes (if any) should declare these to be abstract operations, and which classes should have methods for them.

E15 Explain how you would incorporate the following classes into the hierarchy of Figure 2.8. Describe the attributes and operations that would be present in these classes.
   (a) IsoscelesTriangle
   (b) Square
   (c) Star

E16 Describe what the methods addPoint and removePoint in class ArbitraryPolygon would have to do. Hint: think about what attributes would be affected, and how. You do not need to write any code.

E17 Imagine you want to create an operation called getEnclosingCircle in the hierarchy of Figure 2.8. This operation would compute the smallest circle that can completely enclose any shape. Describe the methods that you think would be needed to implement this operation.

Variables and dynamic binding

Imagine you are programming in an object-oriented language and declare a variable called aShape that has type Shape2D. What this means is that as the program runs, the variable can contain objects of any concrete class in the hierarchy of Shape2D.

If you then attempt to invoke the operation getBoundingRect on the variable aShape, the program will make the decision about what method to run 'on the fly'. The decision-making process is called dynamic binding (or sometimes late binding or virtual binding).

You can imagine that the following procedure is used to perform dynamic binding:

1. The program looks in the class of the object actually stored in the variable. If there is a concrete method for the operation in that class, then it runs the method.
2. Otherwise, it checks in the immediate superclass to see if there is a method there; if so, it runs the method.
3. The program repeats step 2, looking in successively higher superclasses until it finds a concrete method and runs it.
4. If no method is found, then there is an error.
The effect of inheritance hierarchies on polymorphism and variable declarations

Therefore, for example, if you had an instance of `RegularPolygon` in the `aShape` variable, and invoked the operation `getBoundingRect`, the program would look first in `RegularPolygon`, then `SimplePolygon` and finally `Polygon` before it finds a method to run.

If `aShape` had contained an instance of `Rectangle`, however, then the program would find a `getBoundingRect` method in that class immediately.

It would be inefficient if programs ran the above dynamic binding algorithm for every procedure call, therefore an optimized approach using a lookup table is used instead. However, programmers do not normally need to be aware of the optimized mechanism.

Dynamic binding is what gives polymorphism its power. It relieves programmers from the burden of having to write conditional statements to explicitly choose which code to run; with dynamic binding, that work is done automatically by the programming language.

Dynamic binding is only needed when the compiler determines that there is more than one possible method that could be executed by a particular call. Therefore, for example, if you declared a variable to have type `Rectangle`, and you could be sure that `Rectangle` would have no subclasses, then only a `Rectangle` could be put in that variable. In such a case, the compiler can statically determine precisely which method to call.

Exercise

E18 In which of the following situations would dynamic binding be needed? Assume that the compiler knows that no new classes or methods can be added to the hierarchy.

You have a variable of type: You invoke the operation:

a) Rectangle getPerimeterLength
b) SimplePolygon getCenter
c) Polygon getBoundingRect
d) EllipticalShape getScale
e) RegularPolygon translate

Interfaces

An interface in Java is very much like an abstract class, except that it can have neither instance variables nor concrete methods – it is basically a named list of abstract operations. We instead create several implementing classes (rather than subclasses) of an interface that must implement the abstract operations. A class can implement multiple interfaces, but can have only one superclass.
You will see many interfaces built into Java: for example, Comparable is an interface that defines operations that allow objects to be compared, and Runnable is an interface that allows an object to execute as a thread.

A key feature that gives interfaces their power is that you can declare a variable with an interface as its type. This means that an instance of any class that implements the interface can be put in the variable. With the variable, you can then call any of the operations defined in the interface – dynamic binding operates in the same way as with generalization.

We will see in Chapters 5, 6 and 9 that interfaces are very useful for creating good-quality designs.

2.7 Concepts that define object orientation

We have looked at several important aspects of object orientation. It is now time to summarize what we have presented and, at the same time, point out the essential features that distinguish an object-oriented language or system from one that is not object oriented. To be called object oriented, a language needs to have the following features:

- **Identity.** The language must allow a programmer to refer to an object without having to refer to the instance variables contained in the object. Every object has a unique identity; therefore objects that contain instance variables with the same values must be recognized as different objects.

- **Classes.** The programmer must be able to organize the code into classes, each of which describes the structure and function of a set of objects.

- **Inheritance.** There has to be a mechanism to organize these classes into inheritance hierarchies, where features inherit from superclasses to subclasses.

- **Polymorphism.** There has to be a mechanism by which several methods, in related classes, can have the same name and implement the same abstract operation. There must consequently be a dynamic binding mechanism that allows the choice of which method to run to be made during execution of the program.

Sometimes, languages or systems are sold that purport to be object oriented; however, without these key capabilities the term object oriented should not be applied. The term ‘object based’ is sometimes used instead of ‘object oriented’ for technologies which have features like objects or classes but which are perhaps missing inheritance, polymorphism or both.

The following four concepts are enhanced by the presence of the points listed above, and are also integral to object-oriented languages and systems. They allow us to engineer software effectively. We will revisit some of these issues later in the book.

- **Abstraction.** As discussed at the beginning of the chapter, creating an abstraction means creating a simplified representation of something that you
can work with in place of the original thing. Abstractions help you deal with complexity because you can reason about the simpler abstractions instead of the full details of something. There are many abstractions in an object-oriented program:

- **An object** is an abstraction of something of interest to the program, normally something in the real world such as a bank account.

- **A class** is an abstraction of a set of objects; at the same time it also acts as an abstract container for the methods that operate on those objects. The abstraction is improved if fewer methods are public.

- **A superclass** is an abstraction of a set of subclasses: you can declare a variable to be of a certain class, and not care that instances of its subclasses may be put in the variable. An **interface** is a similar but even better abstraction since it has fewer details defined (only abstract operations).

- **A method** is a procedural abstraction that hides its implementation: you can call the method without having to know the implementation.

- **An operation** is an abstraction of a set of methods. Better abstraction is achieved by giving an operation fewer parameters.

- **Attributes and associations** are abstractions of the underlying instance variables used to implement them.

- **Modularity.** An object-oriented system can be constructed entirely from a set of classes, where each class takes care of a particular subset of the functionality (functionality related to a given type of data), rather than having the functionality spread out over many parts of the system.

- **Encapsulation.** A class acts as a container to hold its features (variables and methods) and defines an interface that allows only some of them to be seen from outside.

- Abstraction, modularity and encapsulation each help provide **information hiding.** This arises when software developers using some feature of a programming language or system do not need to know all the details; they only need to know sufficient details to use the feature. The result is that the developers have less confusing detail to understand and will therefore make fewer mistakes. Hence they can work effectively with larger systems.

**Exercise**

**E19** Search the Internet for programming languages, databases or other tools on the market that call themselves object oriented. See if you can determine whether the claim of being object oriented is valid.
Chapter 2
Review of object orientation

History of object orientation – programming languages
The first object-oriented programming language was Simula-67. This language allows programmers to simulate the way objects behave in the real world. For example, a simulation application might model cars approaching an intersection controlled by traffic lights. The objects in this simulation would include cars, lights and traffic lanes. When running a Simula program, each object is represented by a 'chunk' of data. All the procedures that operate on that object are found together in a class, so that the programmer can easily change the behavior of a car or a traffic light without having to search through the entire program.

Although Simula-67 was intended as a special-purpose simulation language, software developers gradually recognized that a wide variety of programs would be easier to develop and understand if organized this way. Although Simula is still used today, mostly in Scandinavia, it never gained widespread popularity.

In the early 1980s a new object-oriented language called Smalltalk gained popularity. Smalltalk was developed at Xerox PARC (Palo Alto Research Center). This research lab is also credited with giving rise to many other inventions, which we take for granted today: graphical user interfaces, the mouse, the laser printer, etc.

Smalltalk has many features that were innovative at the time. It has a simple syntax that is quite unlike that of other popular languages. It has a large library of reusable code – and programmers have access to all the source code for the library. Smalltalk popularized bytecode, platform independence and garbage collection, as now found in Java.

Smalltalk is still used today and has a loyal following, but it was rapidly overtaken in the late 1980s by a new language called C++. The developer of C++, Bjarne Stroustrup, recognized the advantage of object orientation but also recognized that there were tremendous numbers of programmers of the C language who wanted to take advantage of their C expertise and C’s execution speed. He thus added object-oriented extensions to C and the new language became rapidly dominant.

However, over 15 years of experience has shown that C++ has certain drawbacks. Its syntax is quite complex and it is too easy to create code that has bugs. Large C++ programs have thus been found to be hard to maintain – they deteriorate rapidly as many programmers make changes.

In 1991, a group of engineers at Sun Microsystems started a project to design a programming language that could be used in consumer ‘smart devices’. Knowing the strengths and weaknesses of C++, Smalltalk, and a third language called Objective-C, they invented a language initially called Oak. This borrowed the C syntax from C++, and many of its other essential features from Smalltalk. Some of the more troublesome features of C++, such as multiple inheritance and the ability to create pointers to arbitrary parts of memory, were eliminated.

Unfortunately, the team faced difficulties trying to sell Oak. It was only when the Internet gained popularity, with the advent of the World Wide Web in 1994, that Sun saw an opportunity to exploit the technology. The new language, renamed Java, was formally presented in 1995 at the SunWorld ’95 conference.

More recently, Microsoft has entered the fray with its language C# (C-Sharp). C# has very many similarities with Java, but some subtle and interesting differences. C# is one of several languages that can run on Microsoft’s Common Language Runtime, and is part of its .Net framework. Anyone who knows Java should be able to learn C# quite easily.

We will continue this history of object orientation in Chapter 5, where we will look at methods and notations for describing object-oriented systems.
2.8 A program for manipulating postal codes

On the book’s web site (www.lloseng.com) you will find a Java program designed to illustrate the most important features of Java, including inheritance, polymorphism, string manipulation, access control. The program also illustrates an important software engineering concept: separation of the user interface from the functional part of a system.

The example is divided into three elements, as illustrated in Figure 2.10. The first element is a hierarchy representing postal codes of different countries. The second element is a new exception class. The third element is the PostalTest class that allows the user to enter postal codes and test the facilities of the PostalCode hierarchy.

The PostalCode hierarchy

The following are some design decisions you should study in PostalCode and its subclasses:

- PostalCode is declared as abstract, meaning that no instances can be created. Two of its operations, validate and getCountry, are abstract, meaning that they must be given concrete implementations in subclasses.

- The operation validate is protected, and is called by the constructor. Its concrete implementations in each subclass will throw a PostalCodeException (described below) if the format of the code is invalid.

- All the instance variables are declared private. All other classes, including subclasses, can only access them using methods. This helps to improve encapsulation.

- There is a toString method, as should be provided in most Java classes.

There are three examples of subclasses of PostalCode. Each of these implements the two abstract operations. For example, the validate method of one subclass,
Chapter 2
Review of object orientation

CanadianPostalCode, ensures that the format is XNX NXN, where N is a number and X is a letter; the first letter is also taken from a restricted set. The other implementations of validate ensure that US postal codes have an all-numeric format, while British postal codes adhere to their more complex alphanumeric format.

The PostalCodeException class
PostalCodeException illustrates the concept of the user-defined exception class. Instances of this class are thrown when an error is found while validating a postal code. A class that manipulates postal codes could choose to handle such exceptions in any way it wishes.

The user interface class PostalTest

The user interface class, PostalTest, has only a static main method and one private static helper method called getInput. The code prompts the user for input and then attempts to create an instance of one of the subclasses of PostalCode. If a PostalCodeException is thrown, it tries to create an instance of other subclasses until none remain. Then it prints out information about the result. Clearly this is not a sophisticated user interface, nevertheless it is sufficient to test the facilities of the PostalCode hierarchy.

It would be possible to put all the code from PostalTest into PostalCode – the main method in PostalCode would then simply be used to test the class. This is, in fact, a design alternative that some people would choose. We prefer to advocate the complete separation of the classes that do the user interface work from the functional classes.

PostalTest is a rather degenerate class in the object-oriented sense, since it will never have any instances. If any instances were created, then they could do nothing since there are no constructors, instance variables or instance methods. The main method and its helper methods are class methods (also called static methods), reminiscent of the procedural paradigm. For the purposes of having a simple test class, we believe this is acceptable; however, you should be careful not to force class methods to do work that would be better done in instance methods.

Exercises

E20 Run the postal code program. Then carefully read through the code for all six classes. Use the Java documentation to look up any methods or classes you do not understand.

E21 The way the program is written, letters in Canadian postal codes are only accepted if they are upper case. On the other hand, letters in British postal codes are accepted whether they are upper case or not. This is inconsistent.
Modify the program so that user input of upper or lower case is accepted, and the input is converted to upper case immediately.

E22 Describe how you would design the following modifications to the postal code program. Think carefully about whether there should be one method, or several different polymorphic methods. In the latter case, think about whether there should be an abstract method in the superclass and concrete methods in the subclasses, or else a concrete method in the superclass and one or more overriding methods in the subclasses.

(a) There should be an operation `length` that returns the number of characters in a postal code.

(b) There should be a file that contains postal codes, one per line. There should then be an operation called `isOnRecord` that returns `true` if a postal code is in this file. Do not worry for now about the efficiency of this operation in the case of very large files, although you should be aware that this would be a concern in a production-quality system. Hint: investigate class `FileInputStream`.

(c) For each country, there should be a file that contains, on each line, a postal code prefix followed by the name of a destination of such postal codes. For example, class `BritishPostalCode` might use the file `BritishPostalDestinations.txt`, and on one of its lines it might contain 'SW Southwest-London'. The parts of the program that set the destination should read these files.

E23 Implement the designs you prepared in the above exercise.

E24 Add a new subclass representing postal codes for the fictitious country of Ootumlia, whose format is always one or two letters, followed by a space, followed by two numbers. You will have to modify the `PostalTest` class to accommodate your new subclass, although you must not modify the `PostalCode` class.

### 2.9 Classes for representing geometric points

In this section we illustrate the use of the mathematical class library in Java. We also illustrate how a seemingly simple problem can be solved in several rather different ways. You will have the chance to analyze the advantages and disadvantages of various alternatives.

The classes described in this section represent points on a 2-dimensional plane. From mathematics, we know that to represent a point on a plane, you can use x and y coordinates, which are called *Cartesian coordinates*. Alternatively, you can use *polar coordinates*, represented by a radius (often called rho) and an angle (often called theta). In the code we have provided, you can interchangeably work with a given point as Cartesian coordinates or polar coordinates.
Chapter 2
Review of object orientation

Java already has classes for representing geometric points. Take a few moments to look at classes \texttt{Point2D} and \texttt{Point} in the Java documentation.

We will call the point class presented here \texttt{PointCP}; its main distinguishing feature from the built-in Java classes is that it can handle both Cartesian and polar coordinates. We also provide a class called \texttt{PointCPTest} which, like \texttt{PostalTest}, simply provides a user interface for testing. The public methods of both classes are shown in Figure 2.11. The code for these classes can also be found at the book's web site (www.lloseng.com).

Class \texttt{PointCP} contains two private instance variables that can either store \(x\) and \(y\), or else \(rho\) and \(theta\). No matter which storage format is used, all four possible parameters can be computed. Users of the class can also call methods \texttt{convertStorageToCartesian} or \texttt{convertStorageToPolar} in order to explicitly convert the internal storage of an instance to the alternative format.

The above design of \texttt{PointCP} is certainly not the only possible design. Table 2.1 shows several alternative designs; the above design is Design 1.

**Exercises**

**E25** Answer the following questions with respect to the above designs of the \texttt{PointCP} class.

(a) Discuss why it might be useful to allow users of class \texttt{PointCP} (Design 1) to explicitly change the internal storage format, using \texttt{convertStorageToCartesian} or \texttt{convertStorageToPolar}.

(b) What might be a potential hidden weakness of these methods? Hint: what could happen if one is called, then the other, and this process is repeated numerous times?

(c) Write a short program to test whether the weakness you discussed in part b is, in fact, real.

**E26** Create a table describing the various advantages (pros) and disadvantages (cons) of each of the five design alternatives. Some of the factors to consider are: simplicity of code, efficiency when creating instances, efficiency when
Classes for representing geometric points

Table 2.1  Alternative designs for the PointCP class

<table>
<thead>
<tr>
<th>Design</th>
<th>How Cartesian coordinates are computed</th>
<th>How polar coordinates are computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design 1: Store one type of coordinates using a single pair of instance variables, with a flag indicating which type is stored</td>
<td>Simply returned if Cartesian is the storage format, otherwise computed</td>
<td>Simply returned if polar is the storage format, otherwise computed</td>
</tr>
<tr>
<td>Design 2: Store polar coordinates only</td>
<td>Computed on demand, but not stored</td>
<td>Simply returned</td>
</tr>
<tr>
<td>Design 3: Store Cartesian coordinates only</td>
<td>Simply returned</td>
<td>Computed on demand, but not stored</td>
</tr>
<tr>
<td>Design 4: Store both types of coordinates, using four instance variables</td>
<td>Simply returned</td>
<td>Simply returned</td>
</tr>
<tr>
<td>Design 5: Abstract superclass with designs 2 and 3 as subclasses</td>
<td>Depends on the concrete class used</td>
<td>Depends on the concrete class used</td>
</tr>
</tbody>
</table>

doing computations that require both coordinate systems, and amount of memory used.

**E27** Implement and test Design 5. You will also have to make some small changes to PointCPTest. Hints: a) Do you still need the variable typeCoord? b) Do still you need the third argument in the constructor?

**E28** Run a performance analysis in which you compare the performance of Design 5, as you implemented it in the previous exercise, with Design 1. Determine the magnitude of the differences in efficiency, and verify the hypotheses you developed in E26.

**E29** To run a performance analysis, you will have to create a new test class that randomly generates large numbers of instances of PointCP and performs operations on them, such as retrieving polar and Cartesian coordinates. You should then run this test class with the two versions of PointCP – Design 1 and Design 5.

**E30** Summarize your results in a table: the columns of the table would be the two designs; the rows of the table would be the operations. The values reported in the table would be the average computation speed. Make sure you explain your results.
Chapter 2
Review of object orientation

E31 Study the PointCPTest class. It has a complex pair of loops for obtaining input from the user.

(a) Discuss whether you think the design is clear, and if not, why not.
(b) Design, but do not yet implement, an alternative to PointCPTest that does not have the nested loops. What are the drawbacks of this alternative design?
(c) Implement and test your alternative design.

E32 In Design 5 of Table 2.1, we suggested creating an abstract superclass. Another alternative (we can call it Design 6) would instead involve turning PointCP into an interface. Different classes corresponding to designs 1 to 4 would implement this interface.

(a) Design and implement this approach (with two different implementing classes).
(b) What advantages and disadvantages does this approach have?

2.10 Measuring the quality and complexity of a program

It is very important for engineers to be able to measure properties of the materials and devices they work with. A civil engineer, for example, needs to know the load capacity of a beam so that he or she can decide on its required thickness or support. In software engineering, we work with pure information as represented in programs, designs and other documents. Our goals of measurement include: better prediction of the time and effort required for development, and, as was discussed in Section 1.5, improved control of aspects of quality such as reliability, usability and maintainability.

A metric is a well-defined method and formula for computing some value of interest to a software engineer. Below are some of the metrics relevant to the basic principles of object-oriented programming and design we have discussed in this chapter. Each metric is useful as a rough indicator of some quality such as maintainability, or of work involved in development. However, each metric also has disadvantages, which we will address.

Lines of code: You will often notice people describe the amount of work they have accomplished in terms of the number of lines of code they have written. This is a very easy metric to compute and is easy to understand. In large systems, the term KLOC is used, which means thousands of lines of code. A program with more lines of code will typically take more time to develop and maintain than a program with fewer lines of code. Unfortunately, this is not always the case: a smaller program may be more technically complex than a larger program and therefore require more development time; also, either program may be better designed and therefore have fewer defects; finally, a programmer can add duplicate or unneeded lines to make the system appear bigger than it should be.
For these reasons it is considered unfair to judge a programmer’s abilities based solely on the number of lines of code she or he has developed; it is also not reasonable to predict future maintenance based exclusively on this metric.

**Uncommented lines of code:** Sometimes instead of counting all the lines in a source code file, only the lines containing actual source code statements are included; blank lines and those with just comments are left out. This can result in a less biased metric: a programmer could otherwise add extra unneeded comments or blank lines to make the amount of code appear greater. However, the other problems with lines of code mentioned above still remain.

**Percentage of lines with comments:** It is considered a sign of more maintainable code if it has lots of informative comments – in some systems up to 50% is desirable for this metric. However, the comments have to be informative. Also, well-structured code with better choices for variable and method names can be self-documenting and therefore require fewer comments.

**Number of classes:** This is often a good indicator of the overall size of a design. Its main weakness is that the number of classes can be affected significantly by the quality of the design. Some programmers, particularly when they are used to procedural programming, will create too few classes that are too complex; on the other hand, some programmers will create redundant and useless classes.

**Number of methods per class:** If a class has a very large number of methods it is often a sign that it is too complex.

**Number of public methods per class:** Similar to the above, this should be very small. Too many public methods suggests that methods that ought to be private are being made public; alternatively, classes may simply be too complex.

**Number of public instance variables per class:** Ideally this should be zero – it is good practice to make them all as private as possible.

**Number of parameters per method:** A low number is better here – most methods should take zero or one parameter.

**Number of lines of code per method:** It is considered better to have more, but smaller methods. In
Chapter 6 we will see a design pattern that directly leads to this.

**Depth of the inheritance hierarchy**: Very complex inheritance hierarchies can be quite difficult to maintain. At the same time, having no inheritance at all limits opportunities for reuse.

**Number of overridden methods per class**: A number too high here suggests problems in the design. A subclass is supposed to be a specialization of its superclass, not something completely different.

**Exercises**

<table>
<thead>
<tr>
<th>E33</th>
<th>Compute values of each of the metrics described above for the following. Where appropriate, compute values for the entire system, each package, each individual class, and each method.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) The PostalCode system.</td>
</tr>
<tr>
<td></td>
<td>(b) The various designs of the PointCP system.</td>
</tr>
<tr>
<td></td>
<td>(c) Some other system you have developed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E34</th>
<th>Analyze your data from the previous exercise. Rank the metrics in the order in which you think they might:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) Act as indicators of the amount of work that would have been required to develop the code.</td>
</tr>
<tr>
<td></td>
<td>(b) Act as indicators of the maintainability of the system.</td>
</tr>
</tbody>
</table>

**2.11 Difficulties and risks in programming language choice and OO programming**

The following are some of the factors arising from the material in this chapter that can pose a risk to software engineering projects:

- **Language evolution and deprecated features**. Every programming language evolves, such that code written for earlier versions will not run or gives warning messages threatening that it will not run in the future. This has been true for Java – a list of deprecated classes and methods is available as part of the standard Java documentation.
  
  Resolution. Pay careful attention to the documentation describing which features of Java are deprecated.

- **Efficiency can be a concern in some object-oriented systems**. Most implementations of Java run using a virtual machine. This means that Java code tends not to be as efficient as code written in a language such as C++. Java’s exception handling and safety checking also can consume considerable
CPU time. But even object-oriented C++ code can be less efficient than purely procedural code if it uses dynamic binding extensively and allocates objects excessively. Some projects have failed because, when complete, the system did not provide adequate performance.

Resolution. Prototype the system early, especially those parts that involve complex algorithms, in order to determine whether performance will be satisfactory. Learn about the different programming strategies that make a Java program run faster. Consider languages other than Java for number-crunching applications. Profile the running system to discover places where inefficiencies lie, then selectively rewrite code to eliminate the worst inefficiencies.

2.12 Summary

In this chapter, we have reviewed the main principles of object orientation. Object-oriented systems use classes and objects to provide software engineers with a useful combination of data and procedural abstraction.

Some of the key features of object-oriented systems are that they provide inheritance hierarchies and polymorphism. It is important to learn to use these facilities correctly, since abusing them can result in designs that are difficult to maintain. For example, you should check carefully to ensure that all generalizations follow the 'isa' rule, and you should make sure that all features present in a superclass also make sense in each subclass.

2.13 For more information

The following are just a few of the many books and web sites that present information about basic object orientation and Java. Since Java is evolving, and since new books and web sites about it appear almost weekly, check your favorite bookstore and search the web for other material.

Books to help you learn Java and OO principles

Chapter 2

Review of object orientation


Book on programming in general


Book on metrics


Web sites about Java


■ The Java Lobby: http://www.javalobby.org is an excellent site containing Java news and products

■ JavaWorld, an online magazine about Java: http://www.javaworld.com

Tools to help you develop Java code

We recommend that you use an integrated tool to help you develop Java code. The following are some popular alternatives:

■ Borland JBuilder: Borland Corporation: http://www.borland.com

■ CodeWarrior: http://www.metrowerks.com

■ The Eclipse open source development environment: http://www.eclipse.org

Project exercises

The following are additional advanced exercises to help you tune up your Java and programming skills.

E35 Write a package that implements some of the hierarchy of two-dimensional shapes, discussed earlier in this chapter, including the abstract classes and the concrete classes Circle and Rectangle. Your main program should construct some random shapes of the concrete classes, do some transformations on these shapes, and then print out as much information as possible about the resulting shape, including perimeter, area, and bounding rectangle. Use the PointCP class presented earlier where possible. Hints:

■ For a circle, the area is πr² and the perimeter (i.e. the circumference) is 2πr.
To compute the bounding rectangle of an object you have to compute its maximum and minimum points in the x- and y-directions.

As a challenging bonus, you can try to implement ArbitraryPolygon. Use one of the collection classes to store the points. To compute the area, you can divide it into triangles and sum the area of the triangles. The area of a triangle is $0.5 \times \text{base} \times \text{height}$. To compute the bounding rectangle you will have to search through the points to find the maximum and minimum $x$- and $y$-coordinates.

As another bonus you can try to implement the class Ellipse. The area is $\pi \times a \times b$ where $a$ is the semi-minor axis and $b$ is the semi-major axis. The approximate perimeter is $\pi(3(a + b) - \sqrt{(a + 3b)(3a + b)})$. Computing the bounding rectangle of an ellipse is a challenging problem if the ellipse is rotated.

E36 Compare the performance of ArrayList, Vector and ordinary arrays. You should do a series of experiments where you do each of the following tests with the three types of collection, timing the execution of each run. You should run each case several times on the same computer to obtain stable average timings.

(a) Construct very large collections by putting random integers into each collection one at a time. The random integers should range in value from zero to nine. You should make each collection large enough so that the run takes at least 10 seconds to add the integers in the case of an ArrayList. You will have to do some initial experiments to find out what is a good size. You would use the same size of collection for ArrayList, Vector and the array. The ArrayList and Vector can be created by successively adding items and allowing them to grow, while the array has to be created at its full size and then populated with its contents. You could also try to experiment with the case where you do create the ArrayList and Vector initially with their full size.

(b) Construct very large collections as in (i). Then use iterators to sum the elements. Subtract the construction time to get a measure of how much time the iteration takes. Use a for loop for the array, and an Iterator for the Vector and ArrayList.

(c) Again, construct collections as in (i). Then iterate through the collections removing all the even numbers. Subtract the construction time to get a measure of how much time deletion takes. You can only easily do this for Vector and ArrayList.

(d) Once again, construct collections. Then iterate though them adding an extra element after every number 9 encountered. Subtract the original construction time to get an idea of how long adding elements randomly into collections takes. You can only easily do this for Vector and ArrayList.

Write up the results of your experiments as a formal laboratory report. Present your data in suitable tables, and draw conclusions from an analysis of the data. From your conclusions, develop recommendations to designers.