CHAPTER 1

The Object-Oriented Paradigm

Overview

In this chapter

This chapter introduces you to the object-oriented paradigm by comparing and contrasting it with something familiar: standard structured programming.

The object-oriented paradigm grew out of a need to meet the challenges of past practices using standard structured programming. By being clear about these challenges, we can better see the advantages of object-oriented programming, as well as gain a better understanding of this mechanism.

This chapter will not make you an expert on object-oriented methods. It will not even introduce you to all of the basic object-oriented concepts. It will, however, prepare you for the rest of this book, which will explain the proper use of object-oriented design methods as practiced by the experts.

In this chapter,

- I discuss a common method of analysis, called functional decomposition.
- I address the problem of requirements and the need to deal with change (the scourge of programming!).
- I describe the object-oriented paradigm and show its use in action.

- I point out special object methods.
- I provide a table of important object terminology used in this chapter on page 21.

Before The Object-Oriented Paradigm: Functional Decomposition

Functional decomposition is a natural way to deal with complexity

Let's start out by examining a common approach to software development. If I were to give you the task of writing code to access a description of shapes that were stored in a database and then display them, it would be natural to think in terms of the steps required. For example, you might think that you would solve the problem by doing the following:

1. Locate the list of shapes in the database.
2. Open up the list of shapes.
3. Sort the list according to some rules.
4. Display the individual shapes on the monitor.

You could take any one of these steps and further break down the steps required to implement it. For example, you could break down Step 4 as follows:

For each shape in the list, do the following:

4a. Identify type of shape.

4b. Get location of shape.

4c. Call appropriate function that will display shape, giving it the shape's location.

This is called *functional decomposition* because the analyst breaks down (decomposes) the problem into the functional steps that compose it. You and I do this because it is easier to deal with smaller pieces than it is to deal with the problem in its entirety. It is the same approach I might use to write a recipe for making lasagna,

or instructions to assemble a bicycle. We use this approach so often and so naturally that we seldom question it or ask if there are other alternatives.

The challenge with this approach: dealing with change

The problem with functional decomposition is that it does not help us prepare the code for possible changes in the future, for a graceful evolution. When change is required, it is often because I want to add a new variation to an existing theme. For example, I might have to deal with new shapes or new ways to display shapes. If I have put all of the logic that implements the steps into one large function or module, then virtually any change to the steps will require changes to that function or module.

And change creates opportunities for mistakes and unintended consequences. Or, as I like to say,

> *Many bugs originate with changes to code.*

Verify this assertion for yourself. Think of a time when you wanted to make a change to your code, but were afraid to put it in because you knew that modifying the code in one place could break it somewhere else. Why might this happen? Must the code pay attention to all of its functions and how they might be used? How might the functions interact with one another? Were there too many details for the function to pay attention to, such as the logic it was trying to implement, the things with which it was interacting, the data it was using? As it is with people, trying to focus on too many things at once begs for errors when anything changes.

And no matter how hard you try, no matter how well you do your analysis, you can never get all of the requirements from the user. Too much is unknown about the future. Things change. They always do . . .

> *And nothing you can do will stop change. But you do not have to be overcome by it.*

The Problem of Requirements

Requirements always change

Ask software developers what they know to be true about the requirements they get from users. They will often say:

- Requirements are incomplete.
- Requirements are usually wrong.
- Requirements (and users) are misleading.
- Requirements do not tell the whole story.

One thing you will never hear is, "not only were our requirements complete, clear, and understandable, but they laid out all of the functionality we were going to need for the next five years!"

In my thirty years of experience writing software, the main thing I have learned about requirements is that . . .

Requirements always change.

I have also learned that most developers think this is a bad thing. But few of them write their code to handle changing requirements well.

Requirements change for a very simple set of reasons:

- The users' view of their needs change as a result of their discussions with developers and from seeing new possibilities for the software.
- The developers' view of the users' problem domain changes as they develop software to automate it and thus become more familiar with it.
- The environment in which the software is being developed changes. (Who anticipated, five years ago, Web development as it is today?)

This does not mean you and I can give up on gathering good requirements. It does mean that we must write our code to accommodate change. It also means we should stop beating ourselves up (or our customers, for that matter) for things that will naturally occur.

Change happens! Deal with it.

- In all but the simplest cases, requirements will always change, no matter how well we do the initial analysis!
- Rather than complaining about changing requirements, we should change the development process so that we can address change more effectively.

Dealing with Changes: Using Functional Decomposition

Using modularity to contain variation

Look a little closer at the problem of displaying shapes. How can I write the code so that it is easier to handle shifting requirements? Rather than writing one large function, I could make it more modular.

For example, in Step 4c on page 4, where I *"Call appropriate function that will display shape, giving it the shape's location,"* I could write a module like that shown in Example 1-1.

Example 1-1 Using Modularity to Contain Variation

```
function: display shape
input: type of shape, description of shape
action:
   switch (type of shape)
      case square: put display function for square here
      case circle: put display function for circle here
```

Then, when I receive a requirement to be able to display a new type of shape—a triangle, for instance—I only need to change this module (hopefully!).

Problems with modularity in a functional decomposition approach

There are some problems with this approach, however. For example, I said that the inputs to the module were the type of shape and a description of the shape. Depending upon how I am storing shapes, it may or may not be possible to have a consistent description of shapes that will work well for all shapes. What if the description of the shape is sometimes stored as an array of points? Would that still work?

Modularity definitely helps to make the code more understandable, and understandability makes the code easier to maintain. But modularity does not always help code deal with all of the variation it might encounter.

Low cohesion, tight coupling

With the approach that I have used so far, I find that I have two significant problems, which go by the terms *low cohesion* and *tight coupling*. In his book *Code Complete*, Steve McConnell gives an excellent description of both cohesion and coupling. He says,

- *Cohesion* refers to how "closely the operations in a routine are related."[1]

I have heard other people refer to cohesion as *clarity* because the more that operations are related in a routine (or a class), the easier it is to understand things.

- *Coupling* refers to "the strength of a connection between two routines. Coupling is a complement to cohesion. Cohesion describes how strongly the internal contents of a routine are

1. McConnell, S., *Code Complete: A Practical Handbook of Software Construction,* Redmond: Microsoft Press, 1993, p. 81. (Note: McConnell did not invent these terms, we just happen to like his definitions of them best.)

related to each other. Coupling describes how strongly a routine is related to other routines. The goal is to create routines with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other routines (loose coupling)."[2]

Changing a function, or even data used by a function, can wreak havoc on other functions

Most programmers have had the experience of making a change to a function or piece of data in one area of the code that then has an unexpected impact on other pieces of code. This type of bug is called an "unwanted side effect." That is because while we get the impact we want (the change), we also get other impacts we don't want—bugs! What is worse, these bugs are often difficult to find because we usually don't notice the relationship that caused the side effects in the first place (if we had, we wouldn't have changed it the way we did).

In fact, bugs of this type lead me to a rather startling observation:

We really do not spend much time fixing bugs.

I think fixing bugs takes a short period of time in the maintenance and debugging process. The overwhelming amount of time spent in maintenance and debugging is on *finding* bugs and taking the time to avoid unwanted side effects. The actual fix is relatively short!

Since unwanted side effects are often the hardest bugs to find, having a function that touches many different pieces of data makes it more likely that a change in requirements will result in a problem.

2. ibid, p. 87.

The devil is in the side effects.

- A focus on functions is likely to cause side effects that are difficult to find.
- Most of the time spent in maintenance and debugging is not spent on fixing bugs, but in *finding* them and seeing how to avoid unwanted side effects from the fix.

Functional decomposition focuses on the wrong thing

With functional decomposition, changing requirements causes my software development and maintenance efforts to thrash. I am focused primarily on the functions. Changes to one set of functions or data impact other sets of functions and other sets of data, which in turn impact other functions that must be changed. Like a snowball that picks up snow as it rolls downhill, a focus on functions leads to a cascade of changes from which it is difficult to escape.

Dealing with Changing Requirements

How do people do things?

To figure out a way around the problem of changing requirements and to see if there is an alternative to functional decomposition, let's look at how people do things. Let's say that you were an instructor at a conference. People in your class had another class to attend following yours, but didn't know where it was located. One of your responsibilities is to make sure everyone knows how to get to their next class.

If you were to follow a structured programming approach, you might do the following:

1. Get list of people in the class.
2. For each person on this list:
   a. Find the next class they are taking.
   b. Find the location of that class.

c. Find the way to get from your classroom to the person's next class.

d. Tell the person how to get to their next class.

To do this would require the following procedures:

1. A way of getting the list of people in the class
2. A way of getting the schedule for each person in the class
3. A program that gives someone directions from your classroom to any other classroom
4. A control program that works for each person in the class and does the required steps for each person

Doubtful you'd follow this approach

I doubt that you would actually follow this approach. Instead, you would probably post directions to go from this classroom to the other classrooms and then tell everyone in the class, "I have posted the locations of the classes following this in the back of the room, as well as the locations of the other classrooms. Please use them to go to your next classroom." You would expect that everyone would know what their next class was, that they could find the classroom they were to go to from the list, and could then follow the directions for going to the classrooms themselves.

What is the difference between these approaches?

- In the first one—giving explicit directions to everyone—you have to pay close attention to a lot of details. No one other than you is responsible for anything. You will go crazy!
- In the second case, you give general instructions and then expect that each person will figure out how to do the task himself or herself.

Shifting responsibility from yourself to individuals . . .

The biggest difference is this **shift of responsibility**. In the first case, you are responsible for everything; in the second case, students are responsible for their own behavior. In both cases, the same things must be implemented, but the organization is very different.

What is the impact of this?

To see the effect of this reorganization of responsibilities, let's see what happens when some new requirements are specified.

Suppose I am now told to give special instructions to graduate students who are assisting at the conference. Perhaps they need to collect course evaluations and take them to the conference office before they can go to the next class. In the first case, I would have to modify the control program to distinguish the graduate students from the undergraduates, and then give special instructions to the graduate students. It's possible that I would have to modify this program considerably.

. . . can minimize changes

However, in the second case—where people are responsible for themselves—I would just have to write an additional routine for graduate students to follow. The control program would still just say, "Go to your next class." Each person would simply follow the instructions appropriate for himself or herself.

Why the difference?

This is a significant difference for the control program. In one case, it would have to be modified every time there was a new category of students with special instructions that they might be expected to follow. In the other one, new categories of students have to be responsible for themselves.

What makes it happen?

There are three different things going on that make this happen. They are:

- The people are responsible for themselves, instead of the control program being responsible for them. (Note that to accomplish this, a person must also be aware of what type of student he or she is.)
- The control program can talk to different types of people (graduate students and regular students) as if they were exactly the same.
- The control program does not need to know about any special steps that students might need to take when moving from class to class.

Different perspectives

To fully understand the implications of this, it's important to establish some terminology. In *UML Distilled*, Martin Fowler describes three different perspectives in the software development process.[3] These are described in Table 1-1.

Table 1-1 Perspectives in the Software Development Process

Perspective	Description
Conceptual	This perspective "represents the concepts in the domain under study. . . . a conceptual model should be drawn with little or no regard for the software that might implement it . . ."
Specification	"Now we are looking at software, but we are looking at the interfaces of the software, not the implementation."
Implementation	At this point we are at the code itself. "This is probably the most often-used perspective, but in many ways the specification perspective is often a better one to take."

3. Fowler, M., Scott, K., *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 2nd Edition*, Reading, Mass.: Addison-Wesley, 1999, pp. 51–52.

How perspectives help

Look again at the previous example of "Go to your next class." Notice that you—as the instructor—are communicating with the people at the *conceptual level*. In other words, you are telling people what you want, not how to do it. However, the way they go to their next class is very specific. They are following specific instructions and in doing so are working at the *implementation level*.

Communicating at one level (conceptually) while performing at another level (implementation) results in the requestor (the instructor) not knowing exactly what is happening, only knowing conceptually what is happening. This can be very powerful. Let's see how to take these notions and write programs that take advantage of them.

The Object-Oriented Paradigm

Using objects shifts responsibility to a more local level

The object-oriented paradigm is centered on the concept of the object. Everything is focused on objects. I write code organized around objects, not functions.

What is an object? Objects have traditionally been defined as data with *methods* (the object-oriented term for functions). Unfortunately, this is a very limiting way of looking at objects. I will look at a better definition of objects shortly (and again in Chapter 8, "Expanding Our Horizons"). When I talk about the data of an object, these can be simple things like numbers and character strings, or they can be other objects.

The advantage of using objects is that I can define things that are responsible for themselves. (See Table 1-2.) Objects inherently know what type they are. The data in an object allow it to know what state it is in and the code in the object allows it to function properly (that is, do what it is supposed to do).

Table 1-2 Objects and Their Responsibilities

This Object . . .	Is Responsible For . . .
Student	Knowing which classroom they are in Knowing which classroom they are to go to next Going from one classroom to the next
Instructor	Telling people to go to next classroom
Classroom	Having a location
Direction giver	Given two classrooms, giving directions from one classroom to the other

In this case, the objects were identified by looking at the entities in the problem domain. I identified the responsibilities (or methods) for each object by looking at what these entities need to do. This is consistent with the technique of finding objects by looking for the nouns in the requirements and finding methods by looking for verbs. I find this technique to be quite limiting and will show a better way throughout the book. For now, it is a way to get us started.

How to think about objects

The best way to think about what an object is, is to think of it as something with responsibilities. A good design rule is that objects should be responsible for themselves and should have those responsibilities clearly defined. This is why I say one of the responsibilities of a student object is knowing how to go from one classroom to the next.

Or, taking Fowler's perspective

I can also look at objects using the framework of Fowler's perspectives:

- At the *conceptual level*, an object is a set of responsibilities.[4]

4. I am roughly paraphrasing Bertrand Meyer's work of Design by Contract as outlined in *Object-Oriented Software Construction*, Upper Saddle River, N.J.: Prentice Hall, 1997, p. 331.

- At the *specification level*, an object is a set of methods that can be invoked by other objects or by itself.
- At the *implementation level*, an object is code and data.

Unfortunately, object-oriented design is often taught and talked about only at the implementation level—in terms of code and data—rather than at the conceptual or specification level. But there is great power in thinking about objects in these latter ways as well!

Objects have interfaces for other objects to use

Since objects have responsibilities and objects are responsible for themselves, there has to be a way to tell objects what to do. Remember that objects have data to tell the object about itself and methods to implement functionality. Many methods of an object will be identified as callable by other objects. The collection of these methods is called the object's *public interface*.

For example, in the classroom example, I could write the **`Student`** object with the method `gotoNextClassroom()`. I would not need to pass any parameters in because each student would be responsible for itself. That is, it would know:

- What it needs to be able to move
- How to get any additional information it needs to perform this task

Organizing objects around the class

Initially, there was only one kind of student—a regular student who goes from class to class. Note that there would be many of these "regular students" in my classroom (my system). But what if I want to have more *kinds* of students? It seems inefficient for each student type to have its own set of methods to tell it what it can do, especially for tasks that are common to all students.

A more efficient approach would be to have a set of methods associated with all students that each one could use or tailor to their own

needs. I want to define a "general student" to contain the definitions of these common methods. Then, I can have all manner of specialized students, each of whom has to keep track of his or her own private information.

In object-oriented terms, this general student is called a *class*. A class is a definition of the behavior of an object. It contains a complete description of:

- The data elements the object contains
- The methods the object can do
- The way these data elements and methods can be accessed

Since the data elements an object contains can vary, each object of the same type may have different data but will have the same functionality (as defined in the methods).

Objects are instances *of classes*

To get an object, I tell the program that I want a new object of this type (that is, the class that the object belongs to). This new object is called an *instance* of the class. Creating instances of a class is called *instantiation*.

Working with objects in the example

Writing the "Go to the next classroom" example using an object-oriented approach is much simpler. The program would look like this:

1. Start the control program.
2. Instantiate the collection of students in the classroom.
3. Tell the collection to have the students go to their next class.
4. The collection tells each student to go to their next class.
5. Each student:
   a. Finds where his next class is
   b. Determines how to get there

c. Goes there

4. Done.

The need for an abstract type

This works fine until I need to add another student type, such as the graduate student.

I have a dilemma. It appears that I must allow any type of student into the collection (either regular or graduate student). The problem facing me is how do I want the collection to refer to its constituents? Since I am talking about implementing this in code, the collection will actually be an array or something of some type of object. If the collection were named something like, **RegularStudents**, then I would not be able to put **GraduateStudents** into the collection. If I say that the collection is just a group of objects, how can I be sure that I do not include the wrong type of object (that is, something that doesn't do "Go to your next class")?

The solution is straightforward. I need a general type that encompasses more than one specific type. In this case, I want a **Student** type that includes both **RegularStudent**s and **GraduateStudent**s. In object-oriented terms, we call **Student** an *abstract class*.

Abstract classes define what a set of classes can do

Abstract classes define what other, related, classes can do. These "other" classes are classes that represent a particular type of related behavior. Such a class is often called a *concrete class* because it represents a specific, or nonchanging, implementation of a concept.

In the example, the abstract class is **Student**. There are two types of **Student**s represented by the concrete classes, **RegularStudent**s and **GraduateStudent**s. **RegularStudent** is one kind of **Student** and **GraduateStudent** is also a kind of **Student**.

This type of relationship is called an *is-a* relationship, which is formally called *inheritance*. Thus, the **RegularStudent** class *inherits from* **Student**. Other ways to say this would be, the **GraduateStudent** *derives from, specializes,* or *is a subclass of* **Student**.

Going the other way, "the **Student** class is the *base class*, *generalizes,* or is the *superclass of* **GraduateStudent** and of **RegularStudent**.

Abstract classes act as placeholders for other classes

Abstract classes act as placeholders for other classes. I use them to define the methods their derived classes must implement. Abstract classes can also contain common methods that can be used by all derivations. Whether a derived class uses the default behavior or replaces it with its own variation is up to the derivation (this is consistent with the mandate that objects be responsible for themselves).

This means that I can have the controller contain **Student**s. The reference type used will be **Student**. The compiler can check that anything referred to by this **Student** reference is, in fact, a kind of **Student**. This gives the best of both worlds:

- The collection only needs to deal with **Student**s (thereby allowing the instructor object just to deal with students).
- Yet, I still get type checking (only **Student**s that can "Go to their next classroom" are included).
- And, each kind of **Student** is left to implement its functionality in its own way.

Abstract classes are more than classes that do not get instantiated.

Abstract classes are often described as classes that do not get instantiated. This definition is accurate—at the implementation level. But that is too limited. It is more helpful to define abstract classes at the conceptual level. Thus, at the conceptual level, abstract classes are simply placeholders for other classes.

That is, they give us a way to assign a name to a set of related classes. This lets us treat this set as one concept.

In the object-oriented paradigm, you must constantly think about your problem from all three levels of perspective.

Visibility

Since the objects are responsible for themselves, there are many things they do not need to expose to other objects. Earlier, I mentioned the concept of the *public interface*—those methods that are accessible by other objects. In object-oriented systems, the main types of accessibility are:

- *Public*—Anything can see it.
- *Protected*—Only objects of this class and derived classes can see it.
- *Private*—Only objects from this class can see it.

Encapsulation

This leads to the concept of *encapsulation*. Encapsulation has often been described simply as hiding data. Objects generally do not expose their internal data members to the outside world (that is, their visibility is protected or private).

But encapsulation refers to more than hiding data. In general, encapsulation means *any kind of hiding*.

In the example, the instructor did not know which were the regular students and which were the graduate students. The type of student is hidden from the instructor (I am encapsulating the type of student). As you will see later in the book, this is a very important concept.

Polymorphism

Another term to learn is *polymorphism*.

In object-oriented languages, we often refer to objects with one type of reference that is an abstract class type. However, what we are actually referring to are specific instances of classes derived from their abstract classes.

Thus, when I tell the objects to do something conceptually through the abstract reference, I get different behavior, depending upon the specific type of derived object I have. Polymorphism derives from *poly* (meaning many) and *morph* (meaning form). Thus, it means

many forms. This is an appropriate name because I have many different forms of behavior for the same call.

In the example, the instructor tells the students to "Go to your next classroom." However, depending upon the type of student, they will exhibit different behavior (hence polymorphism).

Review of Object-Oriented Terminology

Term	Description
Object	An entity that has responsibilities. I implement these by writing a class (in code) that defines data members (the variables associated with the objects) and methods (the functions associated with the objects).
Class	The repository of methods. Defines the data members of objects. Code is organized around the class.
Encapsulation	Typically defined as data-hiding, but better thought of as any kind of hiding.
Inheritance	Having one class be a special kind of another class. These specialized classes are called derivations of the base class (the initial class). The base class is sometimes called the superclass while the derived classes are sometimes called the subclasses.
Instance	A particular example of a class (it is always an object).
Instantiation	The process of creating an instance of a class.
Polymorphism	Being able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to.
Perspectives	There are three different perspectives for looking at objects: *conceptual*, *specification*, and *implementation*. These distinctions are helpful in understanding the relationship between abstract classes and their derivations. The abstract class defines how to solve things conceptually. It also gives the specification for communicating with any object derived from it. Each derivation provides the specific implementation needed.

Object-Oriented Programming in Action

New example

Let's re-examine the shapes example discussed at the beginning of the chapter. How would I implement it in an object-oriented manner? Remember that it has to do the following:

1. Locate the list of shapes in the database.
2. Open up the list of shapes.
3. Sort the list according to some rules.
4. Display the individual shapes on the monitor.

To solve this in an object-oriented manner, I need to define the objects and the responsibilities they would have.

Using objects in the Shape program

The objects I would need are:

Class	Responsibilities (Methods)
`ShapeDataBase`	*`getCollection`*—get a specified collection of shapes
`Shape` (an abstract class)	*`display`*—defines interface for Shapes *`getX`*—return X location of Shape (used for sorting) *`getY`*—return Y location of Shape (used for sorting)
`Square` (derived from Shape)	*`display`*—display a square (represented by this object)
`Circle` (derived from Shape)	*`display`*—display a circle (represented by this object)
`Collection`	*`display`*—tell all contained shapes to display `sort`—sort the collection of shapes
`Display`	*`drawLine`*—draw a line on the screen *`drawCircle`*—draw a circle on the screen

Running the program

The main program would now look like this:

1. Main program creates an instance of the database object.
2. Main program asks the database object to find the set of shapes I am interested in and to instantiate a collection object containing all of the shapes (actually, it will instantiate circles and squares that the collection will hold).
3. Main program asks the collection to sort the shapes.
4. Main program asks the collection to display the shapes.
5. The collection asks each shape it contains to display itself.
6. Each shape displays itself (using the **`Display`** object) according to the type of shape I have.

Why this helps—handling new requirements

Let's see how this helps to handle new requirements (remember, requirements always change). For example, consider the following new requirements:

- **Add new kinds of shapes (such as a triangle).** To introduce a new kind of shape, only two steps are required:
  - Create a new derivation of **`Shape`** that defines the shape.
  - In the new derivation, implement a version of the display method that is appropriate for that shape.
- **Change the sorting algorithm.** To change the method for sorting the shapes, only one step is required:
  - Modify the method in **`Collection`**. Every shape will use the new algorithm.

Bottom line: The object-oriented approach has limited the impact of changing requirements.

Encapsulation revisited

There are several advantages to encapsulation. The fact that it hides things from the user directly implies the following:

- Using things is easier because the user does not need to worry about implementation issues.
- Implementations can be changed without worrying about the caller. (Since the caller didn't know how it was implemented in the first place, there shouldn't be any dependencies.)
- The insides of an object are unknown to outside objects—they are used by the object to help implement the function specified by the object's interface.

Benefit: reduced side effects

Finally, consider the problem of unwanted side effects that arise when functions are changed. This kind of bug is addressed effectively with encapsulation. The internals of objects are unknown to other objects. If I use encapsulation and follow the strategy that objects are responsible for themselves, then the only way to affect an object will be to call a method on that object. The object's data and the way it implements its responsibilities are shielded from changes caused by other objects.

Encapsulation saves us.

- The more I make my objects responsible for their own behaviors, the less the controlling programs have to be responsible for.
- Encapsulation makes changes to an object's internal behavior transparent to other objects.
- Encapsulation helps to prevent unwanted side effects.

Special Object Methods

Creating and destroying

I have talked about methods that are called by other objects or possibly used by an object itself. But what happens when objects are

created? What happens when they go away? If objects are self-contained units, then it would be a good idea to have methods to handle these situations.

These special methods do, in fact, exist and are called *constructors* and *destructors*.

Constructors initialize, or set up, an object

A constructor is a special method that is automatically called when the object is created. Its purpose is to handle starting up the object. This is part of an object's mandate to be responsible for itself. The constructor is the natural place to do initializations, set default information, set up relationships with other objects, or do anything else that is needed to make a well-defined object. All object-oriented languages look for a constructor method and execute it when the object is created.

By using constructors properly it is easier to eliminate (or at least minimize) uninitialized variables. This type of error usually occurs from carelessness on the part of the developer. By having a set, consistent place for all initializations throughout your code (that is, the constructors of your objects) it is easier to ensure that initializations take place. Errors caused by uninitialized variables are easy to fix but hard to find, so this convention (with the automatic calling of the constructor) can increase the efficiency of programmers.

Destructors clean up an object when it is no longer needed (when it has been deleted)

A destructor is a special method that helps an object clean up after itself when the object goes out of existence; that is, when the object is destroyed. All object-oriented languages look for a destructor method and execute it when the object is being deleted. As with the constructor, the use of the destructor is part of the object's mandate to be responsible for itself.

Destructors are typically used for releasing resources when objects are no longer needed. Since Java has garbage collection (auto-cleanup of objects no longer in use), destructors are not as important

in Java as they are in C++. In C++, it is common for an object's destructor also to destroy other objects that are used only by this object.

Summary

In this chapter

In this chapter, I have shown how object orientation helps us minimize consequences of shifting requirements on a system and how it contrasts with functional decomposition.

I covered a number of the essential concepts in object-oriented programming and have introduced and described the primary terminology. These are essential to understanding the concepts in the rest of this book. (See Tables 1-3 and 1-4.)

Table 1-3 Object-Oriented Concepts

Concept	Review
Functional decomposition	Structured programmers usually approach program design with *functional decomposition.* Functional decomposition is the method of breaking down a problem into smaller and smaller functions. Each function is subdivided until it is manageable.
Changing requirements	Changing requirements are inherent to the development process. Rather than blaming users or ourselves about the seemingly impossible task of getting good and complete requirements, we should use development methods that deal with changing requirements more effectively.
Objects	Objects are defined by their responsibilities. Objects simplify the tasks of programs that use them by being responsible for themselves.
Constructors and destructors	An object has special methods that are called when it is created and deleted. These special methods are: • *Constructors*, which initialize or set up an object. • *Destructors*, which clean up an object when it is deleted. All object-oriented languages use constructors and destructors to help manage objects.

Table 1-4 Object-Oriented Terminology

Term	Definition
Abstract class	Defines the methods and common attributes of a set of classes that are conceptually similar. Abstract classes are never instantiated.
Attribute	Data associated with an object (also called a data member).
Class	Blueprint of an object—defines the methods and data of an object of its type.
Constructor	Special method that is invoked when an object is created.
Encapsulation	Any kind of hiding. Objects encapsulate their data. Abstract classes encapsulate their derived concrete classes.
Derived class	A class that is specialized from a superclass. Contains all of the attributes and methods of the superclass but may also contain other attributes or different method implementations.
Destructor	Special method that is invoked when an object is deleted.
Functional decomposition	A method of analysis in which a problem is broken into smaller and smaller functions.
Inheritance	The way that a class is specialized, used to relate derived classes from their abstractions.
Instance	A particular object of a class.
Instantiation	The process of creating an instance of a class.
Member	Either data or method of a class.
Method	Functions that are associated with an object.
Object	An entity with responsibilities. A special, self-contained holder of both data and methods that operate on that data. An object's data are protected from external objects.
Polymorphism	The ability of related objects to implement methods that are specialized to their type.
Superclass	A class from which other classes are derived. Contains the master definitions of attributes and methods that all derived classes will use (and possibly will override).