Chapter 1

Introduction

This chapter differentiates between two popular programming paradigms, the procedural paradigm and the object-oriented paradigm, the latter being the focus of this book. It goes on to explain how an object-oriented program can be viewed as a set of objects that collaborate to achieve the purpose of the program. From this view, an overview of the chapters of the book is presented. The chapter then describes the two languages used throughout the book, namely the Java programming language and the graphical Unified Modelling Language. The chapter concludes by discussing the idea of design patterns, which are occasionally used in the book to provide solutions to certain problems that arise.

1.1 Programming Paradigms

The two main paradigms for computer programming are the *procedural paradigm* and the *object-oriented paradigm*. The procedural approach is exemplified by programming languages such as C and Pascal. The object-oriented approach is exemplified by programming languages such as C++, Java, and C#.

In procedural programming, code is modularized based on the processes taking place in the system. For example, in a banking application, typical processes that would be considered include the following:

- Opening accounts for customers
- Performing transactions on accounts
- Generating transaction reports at periodic intervals
- Closing accounts

An application developed using the procedural paradigm consists of a set of processes which are hierarchically connected. Processes are typically implemented as software modules. Figure 1.1 shows an application consisting of a set of processes. As can be seen from the diagram, a process can be further decomposed into a set of sub-processes. An application developed using the procedural paradigm tends to mimic the way a computer works. Thus, the application tends to follow a set of sequential steps structured around the input-processing-output model.

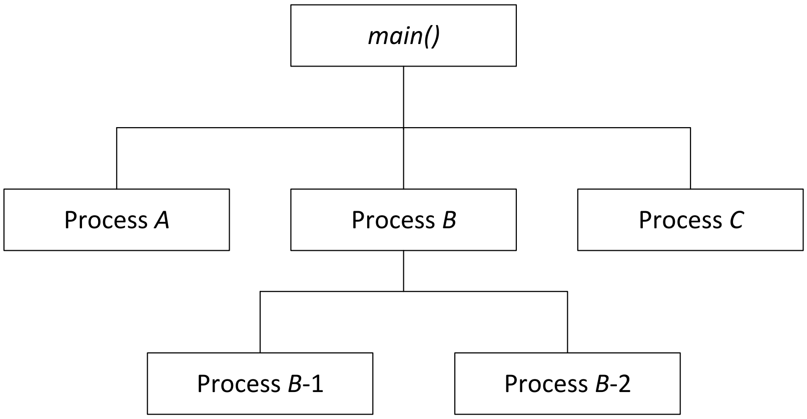

Figure 1.1: Process Decomposition in the Procedural Paradigm

Developing an application using the procedural paradigm typically involves analysing the business processes to discover the procedural tasks that need to be carried out. This is followed by *process modelling* which involves using tools such as *data flow diagrams* to understand how the processes of the application need to work together. Process models highlight the data which needs to flow from one process to another; thus, another activity known as *data modelling* is used to model the data that will be used in the application. From these models, the application is developed consisting of a set of processes and data stores.

In contrast to the procedural paradigm, the object-oriented paradigm views an application as consisting of a set of objects which collaborate to achieve the objectives of the application. An object has both state and behaviour. *State* can be regarded as the data/information that is known about an object. *Behaviour* can be regarded as the set of operations that manipulates the data of an object and so modifies its state.

Developing an object-oriented application involves discovering the objects that will play a role in the application and understanding how objects interact with each other in the problem space. The object-oriented approach usually results in software that has a closer representation of its real-world problem domain than software built using the procedural approach. Because of this, it is easier to build, debug, and maintain.

In addition, the object-oriented paradigm encourages an iterative, incremental process in developing the application. Thus, an application can be developed in stages with additional functionality being provided as an understanding of the real-world application domain evolves. This is in sharp contrast to the procedural paradigm which tended to be inflexible and required a good understanding of the entire application domain before the application could be developed.

1.2 Object-Oriented Programming

An object-oriented program consists of a set of objects that collaborate to achieve the objectives of the program. Two objects collaborate when one object requests a service from the other. The service is provided by the object receiving the request, perhaps in collaboration with other objects. The term *client object* will sometimes be used to refer to the object requesting the service and the term *server object* will sometimes be used to refer to the object providing the service. Figure 1.2 shows a client object collaborating with a server object.

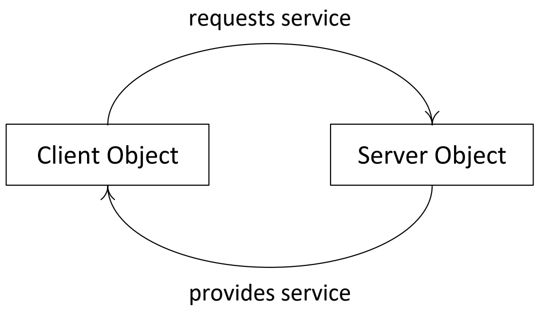

Figure 1.2: Client Object Collaborating with a Server Object

When objects collaborate in an object-oriented program, some objects may be producers of services (server objects) and others may be consumers of services (client objects). However, producer-consumer relationships or client-server relationships are not fixed and the roles may be reversed during the execution of a program. Indeed, an object can be both a service provider and a service consumer.

As an example in the real world, consider a taxi-driver and a banker. A taxi-driver provides a service of transporting a person from one location to another. A banker provides a service of granting loans to persons to purchase things they need. A banker may use the taxi-driver to go from her home to the airport. In this case, the banker is the client and consumes the service provided by the taxi-driver, the server. At some other point in time, the taxi-driver may approach the banker for a loan to purchase a new car for his taxi service. In this case, the taxi-driver is the client and consumes the service provided by the banker, the server. So, the roles of service producer and service consumer are reversed.

Object-oriented programming involves identifying the objects in the real-world problem domain and understanding the relationships between the objects. After this is done, the interactions between objects must be specified. Next, the objects must be implemented in software so that the collaborating objects will provide the functionality required of the application. This book does not cover the process of finding the objects in the real-world problem domain neither does it cover the process of modelling the relationships between objects. Rather, it is a book about writing the code for the collaborating objects in an object-oriented program. Thus, its major concern is to show how to implement objects and their relationships based on established principles and concepts from object-oriented programming.

1.3 Overview of the Book

Since an object-oriented program consists of a community of objects which collaborate to achieve the functionality of the program, the goal of this book is to show how to build this community of collaborating objects using the Java programming language. Figure 1.3 shows a set of objects in an object-oriented application.

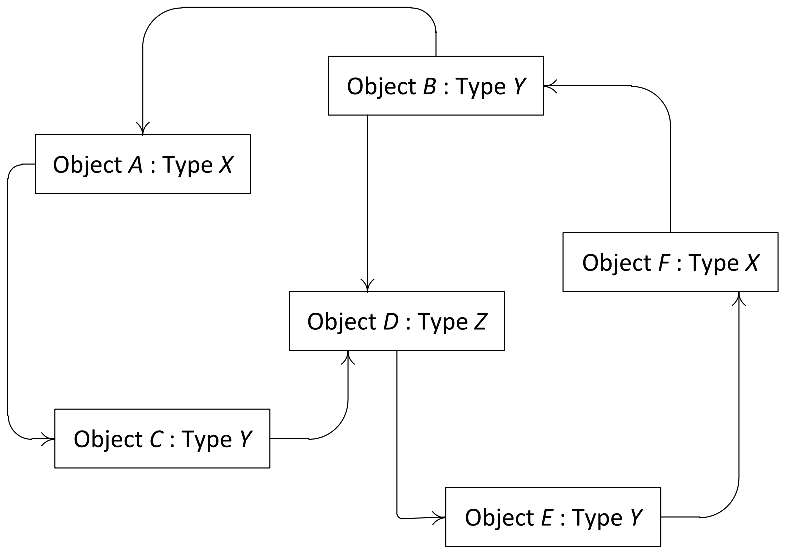

Figure 1.3: Objects Collaborating in an Object-Oriented Application

Figure 1.4 shows how the book is organized. To derive maximum benefit from the book, it should be studied in a sequential manner from the beginning to the end since many concepts must be mastered before moving on to other concepts. The book has been carefully designed to introduce new concepts only after presenting and explaining fully any pre-requisite knowledge and concepts.

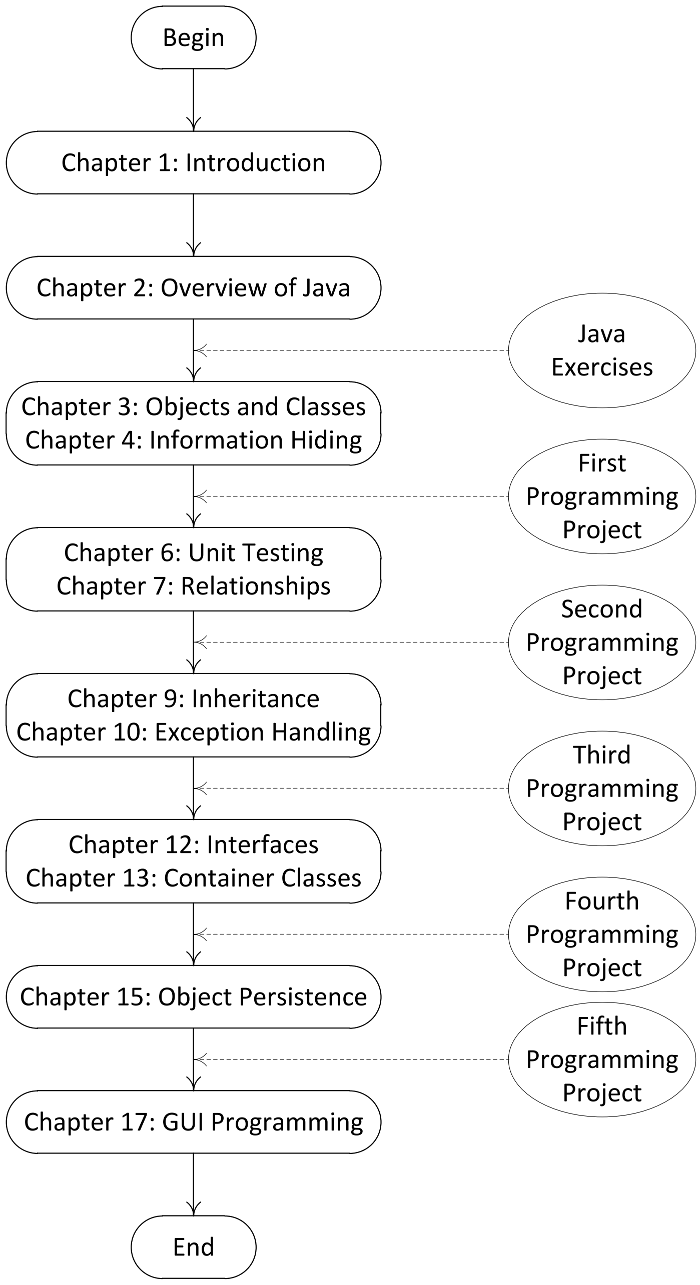
Begin
Chapter 1: Introduction
Chapter 2: Overview of Java
Java Exercises
Chapter 3: Objects and Classes
Chapter 4: Information Hiding
First Programming Project
Chapter 6: Unit Testing
Chapter 7: Relationships
Second Programming Project
Chapter 9: Inheritance
Chapter 10: Exception Handling
Third Programming Project
Chapter 12: Interfaces
Chapter 13: Container Classes
Fourth Programming Project
Chapter 15: Object Persistence
Fifth Programming Project
Chapter 17: GUI Programming
End

Figure 1.4: Outline of the book

1.3.1 Chapter 2

This book uses Java as the programming language to impart the concepts and principles of object-oriented programming. Thus, before doing anything else, it provides a general overview of the key features of the Java programming language in Chapter 2. It assumes that you already know how to write a program in another programming language. Chapter 2 also contains a number of exercises to get you up to speed with Java as quickly as possible.

1.3.2 Chapters 3 and 4

It is clear from Figure 1.3 that an object is one of the most fundamental concepts in object-oriented programming. The objects in the diagram have different *types*. The types are shown after the colon in each rectangle. For example, the type of object `A` is `X`. There can be more than one object of the same type. For example, the type of object `F` is also `X`. The first object-oriented concept covered in the book is that of objects and their types. The class structure provides a means of creating objects of the same type and this is the subject of Chapter 3.

When objects collaborate with each other, it is important that they don't know too much about each other. Objects that don't know much about other objects are highly desirable from a software development point of view since they can be developed, tested, and debugged in relative isolation from objects of other types. These objects are also easy to reuse in other situations since they don't come with "extra baggage". Thus, an important principle in object-oriented programming is *information hiding* or hiding the internal secrets of an object from other objects. This topic is discussed in Chapter 4. This chapter also shows how to group objects into larger units known as *packages* to promote a more coarse-grained kind of information hiding. In terms of information hiding, a class is something like a house and a package is something like a gated-community.

1.3.3 Chapters 6 and 7

Regardless of the programming paradigm used to develop an application, the application must be tested to ensure that it is fit for use and that it achieves its intended objectives. The object-oriented approach results in the compartmentalization of an application into a set of collaborating objects where the objects know very little about each other. This provides an opportunity to test objects separately from each other. The process can also be automated, ensuring that the testing procedure can be saved and repeated in the future. This type of automated testing is called *unit testing* and can be undertaken using a unit testing framework. Chapter 6 gives a thorough description of testing an object-oriented application using the well-known JUnit testing framework.

In order for two objects to collaborate, they must be related in some way. Looking again at the diagram in Figure 1.3, we see that `A` and `B` are related in some way (because of the connecting line) but `A` and `E` are not related to each other. When two objects are related, it is important to ask the question: how many of one object can be related to the other object? The *relationships* in an object-oriented program are important because they must be implemented correctly in code in order for the program to preserve the relationships which currently exist in the real world. For example, if a customer can hold more than one account in a bank, it is important to choose the right data structures to implement the relationship so that at any point in time we can find all the accounts that belong to a particular customer. Relationships between objects are discussed in detail in Chapter 7.

1.3.4 Chapters 9 and 10

Figure 1.3 shows a set of collaborating objects in an application. An important question that comes to mind is: is it possible to reuse code from one object when developing another object? In the procedural paradigm, functions are often written that can be used in many different modules. It is highly desirable if the features which are programmed for one object can somehow be utilized when writing another object. This is indeed attainable in object-oriented programming; in fact, it is referred to as *inheritance* and is one of the hallmarks of object-oriented programming. Chapter 9 discusses the principle of inheritance and provides detailed examples showing how it is implemented in Java. It also discusses the related topic of *polymorphism*.

When objects are collaborating in an object-oriented program, things can go wrong; unexpected things can happen. An object-oriented program (and indeed, any program) should be able to deal gracefully with the exceptional conditions that can occur while the program is executing. Chapter 10 discusses how to build robust programs using the *exception handling* mechanism in Java. This mechanism allows exception objects to be created and thrown when an error condition occurs. These exception objects can be caught and dealt with by other objects resulting in errors being handled in a graceful manner.

1.3.5 Chapters 12 and 13

Chapter 12 of the book describes one of the most powerful features of object-oriented programming known as an *interface*. An interface allows a group of unrelated objects to be treated in a similar manner which simplifies the writing of code. The chapter also gives an appreciation of the kinds of polymorphic behaviour that are made possible when multiple interfaces are implemented. Interfaces also make it possible to write code to handle objects about which nothing is known, except that they implement the features of an interface. Thus, code can be written in an extensible manner to facilitate future enhancements.

As previously discussed, objects can relate to other objects in many ways. Some objects contain collections of other objects. For example, in a banking application, a bank object can contain thousands of customer objects. A data structure for storing a set of objects is called a *collection*. An important issue in object-oriented programming is how to store the objects in a collection to facilitate the type of access required by the application. The Java *Collections* Framework provides a number of built-in objects that are specially designed to store collections of other objects. These include objects such as linked lists, hash tables, trees, etc. Chapter 13 discusses the Java *Collections* Framework. It also shows how to generalize a collection so that it can store any kind of object using a feature known as *generics*.

1.3.6 Chapter 15

A software application needs to store data in some *persistence medium* so that it is available for future use after the application shuts down. Chapter 15 explains how to store the objects in an object-oriented application in a persistence medium. Three types of persistence are discussed: a text file, a relational database, and Java object serialization. Persistence using a relational database is particularly valuable since databases are widely used by organizations around the world. Additional information on accessing several popular relational databases is available at the book Web site.

1.3.7 Chapter 17

Chapter 3 of the book discusses the idea of an object-oriented application consisting of three layers. These layers are the user interface, the business processing, and data storage layer. Most of the chapters of the book are concerned with the objects in the business processing layer and the data storage layer. To complete the picture of an object-oriented application, the last chapter of the book explains how to develop a *graphical user interface* that interacts with objects from the business processing layer and the data storage layer. Incidentally, building a GUI requires knowledge of the material covered in most of the other chapters of the book. Thus, Chapter 17 takes the opportunity to link together all the important concepts in object-oriented programming covered throughout the book.

1.3.8 Remaining Chapters

There are five other chapters in the book which have not yet been mentioned. These chapters are called "Programming Projects" and they explain how to build a complete object-oriented program based on the material presented in the other chapters. These chapters appear when all the relevant material required for writing the program has been thoroughly discussed in the preceding chapters. Each of the five chapters starts by describing the problem in narrative form. The chapter then gives a detailed definition of the functionality required. Next, the chapter gives a detailed explanation of how the functionality can be implemented using the object-oriented concepts presented earlier in the book. The chapter ends with a complete working Java program that solves the problem described. Often, alternative solutions are presented so that you will realize that there are competing ways to implement an object-oriented program with different trade-offs. Of course, all the source code is available for download from this Web site.

1.4 Java and the Unified Modelling Language

This book discusses object-oriented programming using two languages. One is the programming language, Java. The other is a graphical language known as the *Unified Modelling Language* (UML). The book does not focus exclusively on the Java programming language; rather, Java is used as a vehicle for implementing the object-oriented concepts presented in the book. Thus, it avoids many features of the Java programming language which can detract a beginner from learning object-oriented principles. The book assumes that readers already have experience in writing programs in another programming language (including Java). Chapter 2 of the book provides an overview of the Java programming language so that you can get up to speed with the language as quickly as possible.

In writing an object-oriented program, it is useful to have a visual understanding of the objects that collaborate to provide the functionality of the program. Throughout the book, whenever it is appropriate to illustrate an object-oriented concept, this is done using the symbols and notation of the UML. The UML diagrams show the collaborating objects in an application as well as the relationships between these objects. The diagrams provide sufficient information about the collaborating objects in an application so that you can get a good idea of how the objects will collaborate in the actual program. The UML is a widely used modelling language for object-oriented applications. Although only a small subset of the UML is used in this book, it is enough for you to get a very good understanding of how it can be used as a modelling tool.

1.5 Design Patterns

The book occasionally refers to design patterns, a well-known concept in object-oriented software engineering. A *design pattern* is a reusable solution to a commonly occurring problem in object-oriented software development in a given context. The book does not attempt to cover design patterns in a meaningful way. There are many good books that already do that especially the landmark book on Design Patterns [Gamma et al, 1995]. Occasionally, some of the problems addressed by existing design patterns show up in the book; when they do, a design pattern is presented which can be used to solve the problem. In particular, the book highlights the usefulness of design patterns such as Singleton, Iterator, and Model-View Separation when developing object-oriented applications.

Exercises

1. How is the object-oriented paradigm for software development different from the procedural paradigm?

2. Why is an object-oriented program a closer representation of a real-world problem than a procedural program?

3. Differentiate between a client object and server object in an object-oriented program. Explain why objects can be both producers and consumers of services in the same object-oriented program.

4. What is the Unified Modelling Language (UML)? Why is it an important tool in object-oriented software development?

5. Typical processes in a banking application that would result from using the procedural approach are listed in Section 1.1. Suggest a set of objects that could provide the same functionality in an object-oriented application.

6. A design pattern is a solution to a commonly occurring problem in object-oriented programming (in a given context). How do you think design patterns can be useful to a software developer?