THE EXPERT'S VOICE® IN JAVA

Beginning Java 8 Fundamentals

Language Syntax, Arrays, Data Types, Objects, and Regular Expressions

COVERS JAVA 8 FEATURES SUCH AS COMPACT PROFILES, AND THE NEW DATE AND TIME APIS

Kishori Sharan

Foreword by John Zukowski, Co-Founding Author of Apress & Java expert

Apress®

Contents at a Glance

Introduction

How This Book Came About

My first encounter with the Java programming language was during a one-week Java training session in 1997. I did not get a chance to use Java in a project until 1999. I read two Java books and took a Java 2 Programmer certification examination. I did very well on the test, scoring 95 percent. The three questions that I missed on the test made me realize that the books that I had read did not adequately cover details of all the topics necessary about Java. I made up my mind to write a book on the Java programming language. So, I formulated a plan to cover most of the topics that a Java developer needs to use the Java programming language effectively in a project, as well as to get a certification. I initially planned to cover all essential topics in Java in 700 to 800 pages.

As I progressed, I realized that a book covering most of the Java topics in detail could not be written in 700 to 800 hundred pages. One chapter alone that covered data types, operators, and statements spanned 90 pages. I was then faced with the question, "Should I shorten the content of the book or include all the details that I think a Java developer needs?" I opted for including all the details in the book, rather than shortening its content to keep the number of pages low. It has never been my intent to make lots of money from this book. I was never in a hurry to finish this book because that rush could have compromised the quality and the coverage of its contents. In short, I wrote this book to help the Java community understand and use the Java programming language effectively, without having to read many books on the same subject. I wrote this book with the plan that it would be a comprehensive one-stop reference for everyone who wants to learn and grasp the intricacies of the Java programming language.

One of my high school teachers used to tell us that if one wanted to understand a building, one must first understand the bricks, steel, and mortar that make up the building. The same logic applies to most of the things that we want to understand in our lives. It certainly applies to an understanding of the Java programming language. If you want to master the Java programming language, you must start by understanding its basic building blocks. I have used this approach throughout this book, endeavoring to build each topic by describing the basics first. In the book, you will rarely find a topic described without first learning its background. Wherever possible, I have tried to correlate the programming practices with activities in our daily life. Most of the books about the Java programming language available in the market either do not include any pictures at all or have only a few. I believe in the adage, "A picture is worth a thousand words." To a reader, a picture makes a topic easier to understand and remember. I have included plenty of illustrations in the book to aid readers in understanding and visualizing the contents. Developers who have little or no programming experience have difficulty in putting things together to make it a complete program. Keeping them in mind, the book contains over 240 complete Java programs that are ready to be compiled and run.

I spent countless hours doing research for writing this book. My main source of research was the Java Language Specification, white papers and articles on Java topics, and Java Specification Requests (JSRs). I also spent quite a bit of time reading the Java source code to learn more about some of the Java topics. Sometimes, it took a few months researching a topic before I could write the first sentence on the topic. Finally, it was always fun to play with Java programs, sometimes for hours, to add them to the book.

Structure of the Book

This book contains 18 chapters and three appendixes. The chapters contain fundamental topics of Java such as syntax, data types, operators, classes, objects, etc. The chapters are arranged in an order that aids learning the Java programming language faster. The first chapter, "Programming Concepts," explains basic concepts related to programming in general, without going into too much technical details; it introduces Java and its features. The second chapter, "Writing Java Programs," introduces the first program using Java; this chapter is especially written for those learning Java for the first time. Subsequent chapters introduce Java topics in an increasing order of complexity. The new features of Java 8 are included wherever they fit in the chapter. The new Date-Time API, which is one of the biggest addition in Java 8, has been discussed in great detail in over 80 pages in Chapter 12.

After finishing this book, to take your Java knowledge to the next level, two companion books are available by the author: *Beginning Java 8 Language Features* (ISBN 978-1-4302-6658-7) and *Beginning Java 8 APIs, Extensions, and Libraries* (ISBN 978-1-4302-6661-7).

Audience

This book is designed to be useful for anyone who wants to learn the Java programming language. If you are a beginner, with little or no programming background, you need to read from the first chapter to the last, in order. The book contains topics of various degrees of complexity. As a beginner, if you find yourself overwhelmed while reading a section in a chapter, you can skip to the next section or the next chapter, and revisit it later when you gain more experience.

If you are a Java developer with an intermediate or advanced level of experience, you can jump to a chapter or to a section in a chapter directly. If a section uses an unfamiliar topic, you need to visit that topic before continuing the current one.

If you are reading this book to get a certification in the Java programming language, you need to read almost all of the chapters, paying attention to all the detailed descriptions and rules. Most of the certification programs test your fundamental knowledge of the language, not the advanced knowledge. You need to read only those topics that are part of your certification test. Compiling and running over 240 complete Java programs will help you prepare for your certification.

If you are a student who is attending a class in the Java programming language, you need to read the first six chapters of this book thoroughly. These chapters cover the basics of the Java programming languages in detail. You cannot do well in a Java class unless you first master the basics. After covering the basics, you need to read only those chapters that are covered in your class syllabus. I am sure, you, as a Java student, do not need to read the entire book page-by-page.

How to Use This Book

This book is the beginning, not the end, for you to gain the knowledge of the Java programming language. If you are reading this book, it means you are heading in the right direction to learn the Java programming language that will enable you to excel in your academic and professional career. However, there is always a higher goal for you to achieve and you must constantly work harder to achieve it. The following quotations from some great thinkers may help you understand the importance of working hard and constantly looking for knowledge with both your eyes and mind open.

> *The learning and knowledge that we have, is, at the most, but little compared with that of which we are ignorant.*
>
> —Plato

> *True knowledge exists in knowing that you know nothing. And in knowing that you know nothing, that makes you the smartest of all.*
>
> —Socrates

Readers are advised to use the API documentation for the Java programming language, as much as possible, while using this book. The Java API documentation is the place where you will find a complete list of documentation for everything available in the Java class library. You can download (or view) the Java API documentation from the official web site of Oracle Corporation at `www.oracle.com`. While you read this book, you need to practice writing Java programs yourself. You can also practice by tweaking the programs provided in the book. It does not help much in your learning process if you just read this book and do not practice by writing your own programs. Remember that "practice makes perfect," which is also true in learning how to program in Java.

Source Code and Errata

Source code and errata for this book may be downloaded from `www.apress.com/source-code`.

Questions and Comments

Please direct all your questions and comments for the author to `ksharan@jdojo.com`.

CHAPTER 1

Programming Concepts

In this chapter, you will learn

- The general concept of programming
- Different components of programming
- Major programming paradigms
- The object-oriented paradigm and how it is used in Java

What Is Programming?

The term "programming" is used in many contexts. We will discuss its meaning in the context of human-to-computer interaction. In the simplest terms, programming is the way of writing a sequence of instructions to tell a computer to perform a specific task. The sequence of instructions for a computer is known as a program. A set of well-defined notations is used to write a program. The set of notations used to write a program is called a programming language. The person who writes a program is called a programmer. A programmer uses a programming language to write a program.

How does a person tell a computer to perform a task? Can a person tell a computer to perform any task or does a computer have a predefined set of tasks that it can perform? Before we look at human-to-computer communication, let's look at human-to-human communication. How does a human communicate with another human? You would say that human-to-human communication is accomplished using a spoken language, for example, English, German, Hindi, etc. However, spoken language is not the only means of communication between humans. We also communicate using written languages or using gestures without uttering any words. Some people can even communicate sitting miles away from each other without using any words or gestures; they can communicate at thought level.

To have a successful communication, it is not enough just to use a medium of communication like a spoken or written language. The main requirement for a successful communication between two parties is the ability of both parties to understand what is communicated from the other party. For example, suppose there are two people. One person knows how to speak English and the other one knows how to speak German. Can they communicate with each other? The answer is no, because they cannot understand each other's language. What happens if we add an English-German translator between them? We would agree that they would be able to communicate with the help of a translator even though they do not understand each other directly.

Computers understand instructions only in binary format, which is a sequence of 0s and 1s. The sequence of 0s and 1s, which all computers understand, is called machine language or machine code. A computer has a fixed set of basic instructions that it understands. Each computer has its own set of instructions. For example, 0010 may be an instruction to add two numbers on one computer and 0101 is an instruction to add two numbers on another computer. Therefore, programs written in machine language are machine-dependent. Sometimes machine code is referred to as native code as it is native to the machine for which it is written. Programs written in machine language are very difficult, if not impossible, to write, read, understand, and modify. Suppose you want to write a program that adds two numbers, 15 and 12. The program to add two numbers in machine language will look similar to the one shown below. You do not need to understand the sample code written in this section. It is only for the purpose of discussion and illustration.

```
0010010010   10010100000100110
0001000100   01010010001001010
```

The above instructions are to add two numbers. How difficult will it be to write a program in machine language to perform a complex task? Based on the above code, you may now realize that it is very difficult to write, read, and understand a program written in a machine language. But aren't computers supposed to make our jobs easier, not more difficult? We needed to represent the instructions for computers in some notations that were easier to write, read, and understand, so computer scientists came up with another language called an assembly language. An assembly language provides different notations to write instructions. It is little easier to write, read, and understand than its predecessor, machine language. An assembly language uses mnemonics to represent instructions as opposed to the binary (0s and 1s) used in machine language. A program written in an assembly language to add two numbers looks similar to the following:

```
li $t1, 15
add $t0, $t1, 12
```

If you compare the two programs written in the two different languages to perform the same task, you can see that assembly language is easier to write, read, and understand than machine code. There is one-to-one correspondence between an instruction in machine language and assembly language for a given computer architecture. Recall that a computer understands instructions only in machine language. The instructions that are written in an assembly language must be translated into machine language before the computer can execute them. A program that translates the instructions written in an assembly language into machine language is called an assembler. Figure 1-1 shows the relationship between assembly code, an assembler, and machine code.

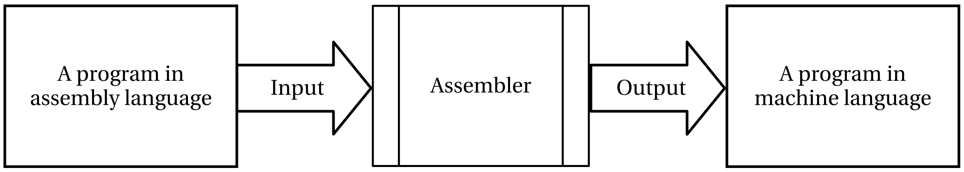


Figure 1-1. *The relationship between assembly code, assembler, and machine code*

Machine language and assembly language are also known as low-level languages. They are called low-level languages because a programmer must understand the low-level details of the computer to write a program using those languages. For example, if you were writing programs in machine and assembly languages, you would need to know what memory location you are writing to or reading from, which register to use to store a specific value, etc. Soon programmers realized a need for a higher-level programming language that could hide the low-level details of computers from them. The need gave rise to the development of high-level programming languages like COBOL, Pascal, FORTRAN, C, C++, Java, C#, etc. The high-level programming languages use English-like words, mathematical notation, and punctuation to write programs. A program written in a high-level programming language is also called source code. They are closer to the written languages that humans are familiar with. The instructions to add two numbers can be written in a high-level programming language, for example. Java looks similar to the following:

```
int x = 15 + 27;
```

You may notice that the programs written in a high-level language are easier and more intuitive to write, read, understand, and modify than the programs written in machine and assembly languages. You might have realized that computers do not understand programs written in high-level languages, as they understand only sequences of 0s and 1s. So there's a need for a way to translate a program written in a high-level language to machine language. The translation is accomplished by a compiler, an interpreter, or a combination of both. A compiler is a program that translates programs written in a high-level programming language into machine language. Compiling a program is an overloaded phrase. Typically, it means translating a program written in a high-level language into machine language. Sometimes it is used to mean translating a program written in a high-level programming language into a lower-level programming language, which is not necessarily the machine language. The code that is generated by a compiler is called compiled code. The compiled program is executed by the computer.

Another way to execute a program written in high-level programming language is to use an interpreter. An interpreter does not translate the whole program into machine language at once. Rather, it reads one instruction written in a high-level programming language at a time, translates it into machine language, and executes it. You can view an interpreter as a simulator. Sometimes a combination of a compiler and an interpreter may be used to compile and run a program written in a high-level language. For example, a program written in Java is compiled into an intermediate language called bytecode. An interpreter, specifically called a Java Virtual Machine (JVM) for the Java platform, is used to interpret the bytecode and execute it. An interpreted program runs slower than a compiled program. Most of the JVMs today use just-in-time compilers (JIT), which compile the entire Java program into machine language as needed. Sometimes another kind of compiler, which is called an ahead-of-time (AOT) compiler, is used to compile a program in an intermediate language (e.g. Java bytecode) to machine language. Figure 1-2 shows the relationship between the source code, a compiler, and the machine code.

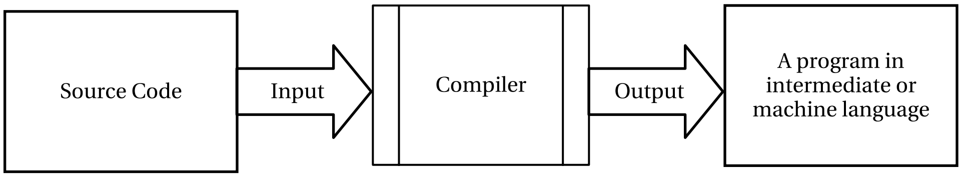


Figure 1-2. *The relationship between source code, a compiler, and machine code*

Components of a Programming Language

A programming language is a system of notations that are used to write instructions for computers. It can be described using three components:

- Syntax
- Semantics
- Pragmatics

The syntax part deals with forming valid programming constructs using available notations. The semantics part deals with the meaning of the programming constructs. The pragmatics part deals with the use of the programming language in practice.

Like a written language (e.g. English), a programming language has a vocabulary and grammar. The vocabulary of a programming language consists of a set of words, symbols, and punctuation marks. The grammar of a programming language defines rules on how to use the vocabulary of the language to form valid programming constructs. You can think of a valid programming construct in a programming language like a sentence in a written language. A sentence in a written language is formed using the vocabulary and grammar of the language. Similarly, a programming construct is formed using the vocabulary and the grammar of the programming language. The vocabulary and the rules to use that vocabulary to form valid programming constructs are known as the syntax of the programming language.

In a written language, you may form a grammatically correct sentence, which may not have any valid meaning. For example, “The stone is laughing.” is a grammatically correct sentence. However, it does not make any sense. In a written language, this kind of ambiguity is allowed. A programming language is meant to communicate instructions to computers, which have no room for any ambiguity. We cannot communicate with computers using ambiguous instructions. There is another component of a programming language, which is called semantics. The semantics of a programming language explain the meaning of the syntactically valid programming constructs. The semantics of a programming language answer the question, “What does this program do when it is run on a computer?” Note that a syntactically valid programming construct may not also be semantically valid. A program must be syntactically and semantically correct before it can be executed by a computer.

The pragmatics of a programming language describe its uses and its effects on the users. A program written in a programming language may be syntactically and semantically correct. However, it may not be easily understood by other programmers. This aspect is related to the pragmatics of the programming language. The pragmatics are concerned with the practical aspect of a programming language. It answers questions about a programming language like its ease of implementation, suitability for a particular application, efficiency, portability, support for programming methodologies, etc.

Programming Paradigms

The online Merriam-Webster’s Learner’s dictionary defines the word “paradigm” as follows:

> *“A paradigm is a theory or a group of ideas about how something should be done, made, or thought about.”*

In the beginning, it is a little hard to understand the word “paradigm” in a programming context. Programming is about providing a solution to a real-world problem using computational models supported by the programming language. The solution is called a program. Before we provide a solution to a problem in the form of a program, we always have a mental view of the problem and its solution. Before I discuss how to solve a real-world problem using a computational model, let’s take an example of a real-world social problem, one that has nothing to do with computers.

Suppose there is a place on Earth that has a shortage of food. People in that place do not have enough food to eat. The problem is “shortage of food.” Let’s ask three people to provide a solution to this problem. The three people are a politician, a philanthropist, and a monk. A politician will have a political view about the problem and its solution. He may think about it as an opportunity to serve his countrymen by enacting some laws to provide food to the hungry people. A philanthropist will offer some money/food to help those hungry people because he feels compassion for all humans and so for those hungry people. A monk will try to solve this problem using his spiritual views. He may preach to them to work and make livings for themselves; he may appeal to rich people to donate food to the hungry; or he may teach them yoga to conquer their hunger! Did you see how three people have different views about the same reality, which is “shortage of food"? The ways they look at the reality are their paradigms. You can think of a paradigm as a mindset with which a reality is viewed in a particular context. It is usual to have multiple paradigms, which let one view the same reality differently. For example, a person who is a philanthropist and politician will have his ability to view the “shortage of food” problem and its solution differently, once with his political mindset and once with his philanthropist mindset. Three people were given the same problem. All of them provided a solution to the problem. However, their perceptions about the problem and its solution were not the same. We can define the term paradigm as a set of concepts and ideas that constitutes a way of viewing a reality.

Why do we need to bother about a paradigm anyway? Does it matter if a person used his political, philanthropical, or spiritual paradigm to arrive at the solution? Eventually we get a solution to our problem. Don’t we?

It is not enough just to have a solution to a problem. The solution must be practical and effective. Since the solution to a problem is always related to the way the problem and the solution are thought about, the paradigm becomes paramount. You can see that the solution provided by the monk may kill the hungry people before they can get any help. The philanthropist's solution may be a good short-term solution. The politician's solution seems to be a long term solution and the best one. It is always important to use the right paradigm to solve a problem to arrive at a practical and the most effective solution. Note that one paradigm cannot be the right paradigm to solve every kind of problem. For example, if a person is seeking eternal happiness, he needs to consult a monk and not a politician or a philanthropist.

Here is a definition of the term "programming paradigm" by Robert W. Floyd, who was a prominent computer scientist. He gave this definition in his 1978 ACM Turing Award lecture titled "The Paradigms of Programming."

"A programming paradigm is a way of conceptualizing what it means to perform computation, and how tasks that are to be carried out on a computer should be structured and organized."

You can observe that the word "paradigm" in a programming context has a similar meaning to that used in the context of daily life. Programming is used to solve a real-world problem using computational models provided by a computer. The programming paradigm is the way you think and conceptualize about the real-world problem and its solution in the underlying computational models. The programming paradigm comes into the picture well before you start writing a program using a programming language. It is in the analysis phase when you use a particular paradigm to analyze a problem and its solution in a particular way. A programming language provides a means to implement a particular programming paradigm suitably. A programming language may provide features that make it suitable for programming using one programming paradigm and not the other.

A program has two components, data and algorithm. Data is used to represent pieces of information. An algorithm is a set of steps that operates on data to arrive at a solution to a problem. Different programming paradigms involve viewing the solution to a problem by combining data and algorithms in different ways. Many paradigms are used in programming. The following are some commonly used programming paradigms:

- Imperative paradigm
- Procedural paradigm
- Declarative paradigm
- Functional paradigm
- Logic paradigm
- Object-oriented paradigm

Imperative Paradigm

The imperative paradigm is also known as an algorithmic paradigm. In the imperative paradigm, a program consists of data and an algorithm (sequence of commands) that manipulates the data. The data at a particular point in time defines the state of the program. The state of the program changes as the commands are executed in a specific sequence. The data is stored in memory. Imperative programming languages provide variables to refer to the memory locations, an assignment operation to change the value of a variable, and other constructs to control the flow of a program. In imperative programming, you need to specify the steps to solve a problem. Suppose you have an integer, say 15, and you want to add 10 to it. Your approach would be to add 1 to 15 ten times and you get the result, 25. You can write a program using an imperative language to add 10 to 15 as follows. Note that you do not need to understand the syntax of the following code; just try to get the feeling of it.

```
int num = 15;    // num holds 15 at this point
int counter = 0; // counter holds 0 at this point

while(counter < 10) {
        num = num + 1;         // Modifying data in num
        counter = counter + 1; // Modifying data in counter
}
// num holds 25 at this point
```

The first two lines are variable declarations that represent the data part of the program. The `while` loop represents the algorithm part of the program that operates on the data. The code inside the `while` loop is executed 10 times. The loop increments the data stored in the `num` variable by 1 in its each iteration. When the loop ends, it has incremented the value of `num` by 10. Note that data in imperative programming is transient and the algorithm is permanent. FORTRAN, COBOL, and C are a few examples of programming languages that support the imperative paradigm.

Procedural Paradigm

The procedural paradigm is similar to the imperative paradigm with one difference: it combines multiple commands in a unit called a procedure. A procedure is executed as a unit. Executing the commands contained in a procedure is known as calling or invoking the procedure. A program in a procedural language consists of data and a sequence of procedure calls that manipulate the data. The following piece of code is typical code for a procedure named `addTen`:

```
void addTen(int num) {
        int counter = 0;
        while(counter < 10) {
                num = num + 1;          // Modifying data in num
                counter = counter + 1;  // Modifying data in counter
        }
        // num has been incremented by 10
}
```

The `addTen` procedure uses a placeholder (also known as parameter) `num`, which is supplied at the time of its execution. The code ignores the actual value of `num`. It simply adds 10 to the current value of `num`. Let's use the following piece of code to add 10 to 15. Note that the code for `addTen` procedure and the following code are not written using any specific programming language. They are provided here only for the purpose of illustration.

```
int x = 15; // x holds 15 at this point
addTen(x);  // Call addTen procedure that will increment x by 10
// x holds 25 at this point
```

You may observe that the code in imperative paradigm and procedural paradigm are similar in structure. Using procedures results in modular code and increases reusability of algorithms. Some people ignore this difference and treat the two paradigms, imperative and procedural, as the same. Note that even if they are different, a procedural paradigm always involves the imperative paradigm. In the procedural paradigm, the unit of programming is not a sequence of commands. Rather, you abstract a sequence of commands into a procedure and your program consists of a sequence of procedures instead. A procedure has side effects. It modifies the data part of the program as it executes its logic. C, C++, Java, and COBOL are a few examples of programming languages that support the procedural paradigm.

Declarative Paradigm

In the declarative paradigm, a program consists of the description of a problem and the computer finds the solution. The program does not specify how to arrive at the solution to the problem. It is the computer's job to arrive at a solution when a problem is described to it. Contrast the declarative paradigm with the imperative paradigm. In the imperative paradigm, we are concerned about the "how" part of the problem. In the declarative paradigm, we are concerned about the "what" part of the problem. We are concerned about what the problem is, rather than how to solve it. The functional paradigm and the logic paradigm, which are described next, are subtypes of the declarative paradigm.

Writing a database query using a structured query language (SQL) falls under programming based on the declarative paradigm where you specify what data you want and the database engine figures out how to retrieve the data for you. Unlike the imperative paradigm, the data is permanent and the algorithm is transient in the declarative paradigm. In the imperative paradigm, the data is modified as the algorithm is executed. In the declarative paradigm, data is supplied to the algorithm as input and the input data remains unchanged as the algorithm is executed. The algorithm produces new data rather than modifying the input data. In other words, in the declarative paradigm, execution of an algorithm does not produce side effects.

Functional Paradigm

The functional paradigm is based on the concept of mathematical functions. You can think of a function as an algorithm that computes a value from some given inputs. Unlike a procedure in procedural programming, a function does not have a side effect. In functional programming, values are immutable. A new value is derived by applying a function to the input value. The input value does not change. Functional programming languages do not use variables and assignments, which are used for modifying data. In imperative programming, a repeated task is performed using a loop construct, for example, a `while` loop. In functional programming, a repeated task is performed using recursion, which is a way in which a function is defined in terms of itself. In other words, it does some work, then calls itself.

A function always produces the same output when it is applied on the same input. A function, say `add`, that can be applied to an integer `x` to add an integer `n` to it may be defined as follows:

```
int add(x, n) {
        if (n == 0) {
                return x;
        }
        else {
                return 1 + add(x, n-1); // Apply add function recursively
        }
}
```

Note that the `add` function does not use any variable and does not modify any data. It uses recursion. You can call the `add` function to add 10 to 15 as follows:

```
add(15, 10); // Results in 25
```

Haskell, Erlang, and Scala are a few examples of programming languages that support the functional paradigm.

■ **Tip** Java 8 added a new language construct called a lambda expression, which can be used to perform functional programming in Java.

Logic Paradigm

Unlike the imperative paradigm, the logic paradigm focuses on the "what" part of the problem rather than how to solve it. All you need to specify is what needs to be solved. The program will figure out the algorithm to solve it. The algorithm is of less importance to the programmer. The primary task of the programmer is to describe the problem as closely as possible. In the logic paradigm, a program consists of a set of axioms and a goal statement. The set of axioms is the collection of facts and inference rules that make up a theory. The goal statement is a theorem. The program uses deductions to prove the theorem within the theory. Logic programming uses a mathematical concept called a relation from set theory. A relation in set theory is defined as a subset of the Cartesian product of two or more sets. Suppose there are two sets, `Persons` and `Nationality`, which are defined as follows:

```
Person = {John, Li, Ravi}
Nationality = {American, Chinese, Indian}
```

The Cartesian product of the two sets, denoted as `Person x Nationality`, would be another set, as shown:

```
Person x Nationality = {{John, American}, {John, Chinese},
                        {John, Indian}, {Li, American}, {Li, Chinese},
                        {Li, Indian}, {Ravi, American}, {Ravi, Chinese},
                        {Ravi, Indian}}
```

Every subset of `Person x Nationality` is another set that defines a mathematical relation. Each element of a relation is called a tuple. Let `PersonNationality` be a relation defined as follows:

```
PersonNationality = {{John, American}, {Li, Chinese}, {Ravi, Indian}}
```

In logic programming, you can use the `PersonNationality` relation as the collection of facts that are known to be true. You can state the goal statement (or the problem) like

```
PersonNationality(?, Chinese)
```

which means "give me all names of people who are Chinese." The program will search through the `PersonNationality` relation and extract the matching tuples, which will be the answer (or the solution) to your problem. In this case, the answer will be `Li`.

Prolog is an example of a programming language that supports the logic paradigm.

Object-Oriented Paradigm

In the object-oriented (OO) paradigm, a program consists of interacting objects. An object encapsulates data and algorithms. Data defines the state of an object. Algorithms define the behavior of an object. An object communicates with other objects by sending messages to them. When an object receives a message, it responds by executing one of its algorithms, which may modify its state. Contrast the object-oriented paradigm with the imperative and functional paradigms. In the imperative and functional paradigms, data and algorithms are separated, whereas in the object-oriented paradigm, data and algorithms are not separate; they are combined in one entity, which is called an object.

Classes are the basic units of programming in the object-oriented paradigm. Similar objects are grouped into one definition called a class. A class' definition is used to create an object. An object is also known as an instance of the class. A class consists of instance variables and methods. The values of instance variables of an object define the state of the object. Different objects of a class maintain their states separately. That is, each object of a class has its own copy of the instance variables. The state of an object is kept private to that object. That is, the state of an object cannot be accessed or modified directly from outside the object. Methods in a class define the behavior of its objects. A method is like a procedure (or subroutine) in the procedural paradigm. Methods can access/modify the state of the object. A message is sent to an object by invoking one of its methods.

Suppose you want to represent real-world people in your program. You will create a Person class and its instances will represent people in your program. The Person class can be defined as shown in Listing 1-1. This example uses the syntax of the Java programming language. You do not need to understand the syntax used in the programs that you are writing at this point; I will discuss the syntax to define classes and create objects in subsequent chapters.

Listing 1-1. The Definition of a Person Class Whose Instances Represent Real-World Persons in a Program

```
package com.jdojo.concepts;

public class Person {
        private String name;
        private String gender;

        public Person(String initialName, String initialGender) {
                name = initialName;
                gender = initialGender;
        }

        public String getName() {
                return name;
        }

        public void setName(String newName) {
                name = newName;
        }

        public String getGender() {
                return gender;
        }
}
```

The Person class includes three things:

- Two instance variables: name and gender.
- One constructor: Person(String initialName, String initialGender)
- Three methods: getName(), setName(String newName), and getGender()

Instance variables store internal data for an object. The value of each instance variable represents the value of a corresponding property of the object. Each instance of the Person class will have a copy of name and gender data. The values of all properties of an object at a point in time (stored in instance variables) collectively define the state of the object at that time. In the real world, a person possesses many properties, for example, name, gender, height, weight, hair color, addresses, phone numbers, etc. However, when you model the real-world person using a class, you include only those properties of the person that are relevant to the system being modeled. For this current demonstration, let's model only two properties, name and gender, of a real-world person as two instance variables in the Person class.

A class contains the definition (or blueprint) of objects. There needs to be a way to construct (to create or to instantiate) objects of a class. An object also needs to have the initial values for its properties that will determine its initial state at the time of its creation. A constructor of a class is used to create an object of that class. A class can have many constructors to facilitate the creation of its objects with different initial states. The Person class provides one constructor, which lets you create its object by specifying the initial values for name and gender. The following snippet of code creates two objects of the Person class:

```
Person john = new Person("John Jacobs", "Male");
Person donna = new Person("Donna Duncan", "Female");
```

The first object is called john with John Jacobs and Male as the initial values for its name and gender properties, respectively. The second object is called donna with Donna Duncan and Female as the initial values for its name and gender properties, respectively.

Methods of a class represent behaviors of its objects. For example, in the real world, a person has a name and his ability to respond when he is asked for his name is one of his behaviors. Objects of the Person class have abilities to respond to three different messages: getName, setName, and getGender. The ability of an object to respond to a message is implemented using methods. You can send a message, say getName, to a Person object and it will respond by returning its name. It is the same as asking "What is your name?" and having the person respond by telling you his name.

```
String johnName = john.getName();   // Send getName message to john
String donnaName = donna.getName(); // Send getName message to donna
```

The setName message to the Person object asks him to change his current name to a new name. The following snippet of code changes the name of the donna object from Donna Duncan to Donna Jacobs:

```
donna.setName("Donna Jacobs");
```

If you send the getName message to donna object at this point, it will return Donna Jacobs and not Donna Duncan.

You may notice that your Person objects do not have the ability to respond to a message such as - setGender. The gender of Person object is set when the object is created and it cannot be changed afterwards. However, you can query the gender of a Person object by sending getGender message to it. What messages an object may (or may not) respond to is decided at design-time based on the need of the system being modeled. In the case of the Person objects, we decided that they would not have the ability to respond to the setGender message by not including a setGender(String newGender) method in the Person class.

Figure 1-3 shows the state and interface of the Person object called john.

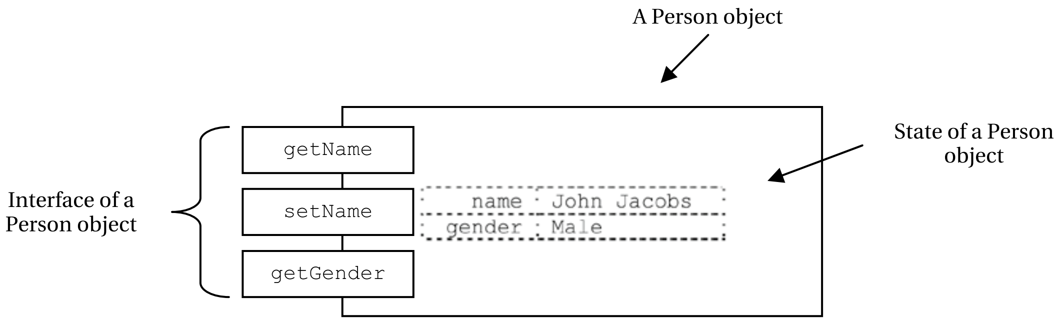


Figure 1-3. *The state and the interface for a Person object*

The object-oriented paradigm is a very powerful paradigm for modeling real-world phenomena in a computational model. We are used to working with objects all around us in our daily life. The object-oriented paradigm is natural and intuitive as it lets you think in terms of objects. However, it does not give you the ability to think in terms of objects correctly. Sometimes the solution to a problem does not fall into the domain of an object-oriented paradigm. In such cases, you need to use the paradigm that suits the problem domain the most. The object-oriented paradigm has a learning curve. It is much more than just creating and using objects in your program. Abstraction, encapsulation, polymorphism, and inheritance are some of the important features of the object-oriented paradigm. You must understand and be able to use these features to take full advantage of the object-oriented paradigm. I will discuss these features of the object-oriented paradigm in the sections to follow. In subsequent chapters, I will discuss these features and how to implement them in a program in detail.

To name a few, C++, Java and C# (pronounced as C sharp) are programming languages that support the object-oriented paradigm. Note that a programming language itself is not object-oriented. It is the paradigm that is object-oriented. A programming language may or may not have features to support the object-oriented paradigm.

What Is Java?

Java is a general purpose programming language. It has features to support programming based on the object-oriented, procedural, and functional paradigms. You often read a phrase like "Java is an object-oriented programming language." What is meant is that the Java language has features that support the object-oriented paradigm. A programming language is not object-oriented. It is the paradigm that is object-oriented, and a programming language may have features that make it easy to implement the object-oriented paradigm. Sometimes programmers have misconceptions that all programs written in Java are always object-oriented. Java also has features that support the procedural and functional paradigms. You can write a program in Java that is a 100% procedural program without an iota of object-orientedness in it.

The initial version of the Java platform was released by Sun Microsystems (part of Oracle Corporation since January 2010) in 1995. Development of the Java programming language was started in 1991. Initially, the language was called Oak and it was meant to be used in set-top boxes for televisions.

Soon after its release, Java became a very popular programming language. One of the most important features for its popularity was its "write once, run anywhere" (WORA) feature. This feature lets you write a Java program once and run it on any platform. For example, you can write and compile a Java program on UNIX and run it on Microsoft Windows, Macintosh, or UNIX machine without any modifications to the source code. WORA is achieved by compiling a Java program into an intermediate language called bytecode. The format of bytecode is platform-independent. A virtual machine, called the Java Virtual Machine (JVM), is used to run the bytecode on each platform. Note that JVM is a program implemented in software. It is not a physical machine and this is the reason it is called a "virtual" machine. The job of a JVM is to transform the bytecode into executable code according to the platform it is running on. This feature makes Java programs platform-independent. That is, the same Java program can be run on multiple platforms without any modifications.

The following are a few characteristics behind Java's popularity and acceptance in the software industry:

- Simplicity
- Wide variety of usage environments
- Robustness

Simplicity may be a subjective word in this context. C++ was the popular and powerful programming language widely used in the software industry at the time Java was released. If you were a C++ programmer, Java would provide simplicity for you in its learning and use over the C++ experience you had. Java retained most of the syntax of C/C++, which was helpful for C/C++ programmers trying to learn this new language. Even better, it excluded some of the most confusing and hard-to-use-correctly features (though powerful) of C++. For example, Java does not have pointers and multiple inheritance, which are present in C++.

If you are learning Java as your first programming language, whether it is a simple language to learn may not be true for you. This is the reason why I said that the simplicity of Java or any programming language is very subjective. The Java language and its libraries (a set of packages containing Java classes) have been growing ever since its first release. You will need to put in some serious effort in order to become a serious Java developer.

Java can be used to develop programs that can be used in different environments. You can write programs in Java that can be used in a client-server environment. The most popular use of Java programs in its early days was to develop applets. An applet is a Java program that is embedded in a web page, which uses the HyperText Markup Language (HTML), and is displayed in a web browser such as Microsoft Internet Explorer, Google Chrome, etc. An applet's code is stored on a web server, downloaded to the client machine when the HTML page containing the reference to the applet is loaded by the browser, and run on the client machine. Java includes features that make it easy to develop distributed applications. A distributed application consists of programs running on different machines connected through a network. Java has features that make it easy to develop concurrent applications. A concurrent application has multiple interacting threads of execution running in parallel. I will discuss these features of the Java platform in detail in subsequent chapters in this book.

Robustness of a program refers to its ability to handle unexpected situations reasonably. The unexpected situation in a program is also known as an error. Java provides robustness by providing many features for error checking at different points during a program's lifetime. The following are three different types of errors that may occur in a Java program:

- Compile-time error
- Runtime error
- Logic error

Compile-time errors are also known as syntax errors. They are caused by incorrect use of the Java language syntax. Compile-time errors are detected by the Java compiler. A program with a compile-time error does not compile into bytecode until the errors are corrected. Missing a semicolon at the end of a statement, assigning a decimal value such as 10.23 to a variable of integer type, etc. are examples of compile-time errors.

Runtime errors occur when a Java program is run. This kind of error is not detected by the compiler because a compiler does not have all of the runtime information available to it. Java is a strongly typed languages and it has a robust type checking at compile-time as well as runtime. Java provides a neat exception handling mechanism to handle runtime errors. When a runtime error occurs in a Java program, the JVM throws an exception, which the program may catch and deal with. For example, dividing an integer by zero (e.g. `17/0`) generates a runtime error. Java avoids critical runtime errors, such as memory overrun and memory leaks, by providing a built-in mechanism for automatic memory allocation and deallocation. The feature of automatic memory deallocation is known as garbage collection.

Logic errors are the most critical errors in a program, and they are hard to find. They are introduced by the programmer by implementing the functional requirement incorrectly. This kind of error cannot be detected by a Java compiler or Java runtime. They are detected by application testers or users when they compare the actual behavior of a program with its expected behavior. Sometimes a few logic errors can sneak into the production environment and they go unnoticed even after the application is decommissioned.

An error in a program is known as a bug. The process of finding and fixing bugs in a program is known as debugging. All modern integrated development environments (IDEs) such as NetBeans, Eclipse, JDeveloper, JBuilder, etc, provide programmers with a tool called a debugger, which lets them run the program step-by-step and inspect the program's state at every step to detect the bug. Debugging is a reality of programmer's daily activities. If you want to be a good programmer, you must learn and be good at using the debuggers that come with the development tools that you use to develop your Java programs.

The Object-Oriented Paradigm and Java

The object-oriented paradigm supports four major principles: abstraction, encapsulation, inheritance, and polymorphism. They are also known as four pillars of the object-oriented paradigm. Abstraction is the process of exposing the essential details of an entity, while ignoring the irrelevant details, to reduce the complexity for the users. Encapsulation is the process of bundling data and operations on the data together in an entity. Inheritance is used to derive a new type from an existing type, thereby establishing a parent-child relationship. Polymorphism lets an entity take on different meanings in different contexts. The four principles are discussed in detail in the sections to follow.

Abstraction

A program provides solutions to a real-world problem. The size of the program may range from a few lines to a few million lines. It may be written as a monolithic structure running from the first line to the millionth line in one place. A monolithic program becomes harder to write, understand, and maintain if its size is over 25 to 50 lines. For easier maintenance, a big monolithic program must be decomposed into smaller subprograms. The subprograms are then assembled together to solve the original problem. Care must be taken when a program is being decomposed. All subprograms must be simple and small enough to be understood by themselves, and when assembled together, they must solve the original problem.

Let's consider the following requirement for a device:

> *Design and develop a device that will let its user type text using all English letters, digits, and symbols.*

One way to design such a device is to provide a keyboard that has keys for all possible combinations of all letters, digits, and symbols. This solution is not reasonable as the size of the device will be huge. You may realize that we are talking about designing a keyboard. Look at your keyboard and see how it has been designed. It has broken down the problem of typing text into typing a letter, a digit, or a symbol one at a time, which represents the smaller part of the original problem. If you can type all letters, all digits, and all symbols one at a time, you can type text of any length.

Another decomposition of the original problem may include two keys: one to type a horizontal line and another to type a vertical line, which a user can use to type in `E`, `T`, `I`, `F`, `H`, and `L` because these letters consist of only horizontal and vertical lines. With this solution, a user can type six letters using the combination of just two keys. However, with your experience using keyboards, you may realize that decomposing the keys so that a key can be used to type in only part of a letter is not a reasonable solution, although it is a solution.

Why is providing two keys to type six letters not a reasonable solution? Aren't we saving space and number of keys on the keyboard? The use of the phrase "reasonable" is relative in this context. From a purist point of view, it may be a reasonable solution. My reasoning behind calling it "not reasonable" is that it is not easily understood by users. It exposes more details to the users than needed. A user would have to remember that the horizontal line is placed at the top for `T` and at bottom for `L`. When a user gets a separate key for each letter, he does not have to deal with these details. It is important that the subprograms that provide solutions to parts of the original problem must be simplified to have the same level of detail to work together seamlessly. At the same time, a subprogram should not expose details that are not necessary for someone to know in order to use it.

Finally, all keys are mounted on a keyboard and they can be replaced separately. If a key is broken, it can be replaced without worrying about other keys. Similarly, when a program is decomposed into subprograms, a modification in a subprogram should not affect other subprograms. Subprograms can also be further decomposed by focusing on a different level of detail and ignoring other details. A good decomposition of a program aims at providing the following characteristics:

- Simplicity
- Isolation
- Maintainability

Each subprogram should be simple enough to be understood by itself. Simplicity is achieved by focusing on the relevant pieces of information and ignoring the irrelevant ones. What pieces of information are relevant and what are irrelevant depends on the context.

Each subprogram should be isolated from other subprograms so that any changes in a subprogram should have localized effects. A change in one subprogram should not affect any other subprograms. A subprogram defines an interface to interact with other subprograms. The inner details about the subprogram are hidden from the outside world. As long as the interface for a subprogram remains unchanged, the changes in its inner details should not affect the other subprograms that interact with it.

Each subprogram should be small enough to be written, understood, and maintained easily.

All of the above characteristics are achieved during decomposition of a problem (or program that solves a problem) using a process called abstraction. What is abstraction? Abstraction is a way to perform decomposition of a problem by focusing on relevant details and ignoring the irrelevant details about it in a particular context. Note that no details about a problem are irrelevant. In other words, every detail about a problem is relevant. However, some details may be relevant in one context and some in another. It is important to note that it is the "context" that decides what details are relevant and what are irrelevant. For example, consider the problem of designing and developing a keyboard. For a user's perspective, a keyboard consists of keys that can be pressed and released to type text. Number, type, size, and position of keys are the only details that are relevant to the users of a keyboard. However, keys are not the only details about a keyboard. A keyboard has an electronic circuit and it is connected to a computer. A lot of things occur inside the keyboard and the computer when a user presses a key. The internal workings of a keyboard are relevant for keyboard designers and manufactures. However, they are irrelevant to the users of a keyboard. You can say that different users have different views of the same thing in different contexts. What details about the thing are relevant and what are irrelevant depends on the user and the context.

Abstraction is about considering details that are necessary to view the problem in the way that is appropriate in a particular context and ignoring (hiding or suppressing or forgetting) the details that are unnecessary. Terms like "hiding" and "suppressing" in the context of abstraction may be misleading. These terms may mean hiding some details of a problem. Abstraction is concerned with which details of a thing should be considered and which should not for a particular purpose. It does imply hiding of the details. How things are hidden is another concept called information hiding, which is discussed in the following section.

The term "abstraction" is used to mean one of the two things: a process or an entity. As a process, it is a technique to extract relevant details about a problem and ignore the irrelevant details. As an entity, it is a particular view of a problem that considers some relevant details and ignores the irrelevant details.

Abstraction for Hiding Complexities

Let's discuss the application of abstraction in real-world programming. Suppose you want to write a program that will compute the sum of all integers between two integers. Suppose you want to compute the sum of all integers between 10 and 20. You can write the program as follows. Do not worry if you do not understand the syntax used in programs in this section; just try to grasp the big picture of how abstraction is used to decompose a program.

```
int sum = 0;
int counter = 10;
while(counter <= 20) {
        sum = sum + counter;
        counter = counter + 1;
}
System.out.println(sum);
```

This snippet of code will add `10 + 11 + 12 + ... + 20` and print `165`.

Suppose you want to compute sum of all integers between `40` and `60`. Here is the program to achieve just that:

```
int sum = 0;
int counter = 40;
while(counter <= 60) {
        sum = sum + counter;
        counter = counter + 1;
}
System.out.println(sum);
```

This snippet of code will perform the sum of all integers between 40 and 60, and it will print 1050. Note the similarities and differences between the two snippets of code. The logic is the same in both. However, the lower limit and the upper limit of the range are different. If you can ignore the differences that exist between the two snippets of code, you will be able to avoid the duplicating of logic in two places.

Let's consider the following snippet of code:

```
int sum = 0;
int counter = lowerLimit;
while(counter <= upperLimit) {
        sum = sum + counter;
        counter = counter + 1;
}
System.out.println(sum);
```

This time, you did not use any actual values for the lower and upper limits of any range. Rather, you used `lowerLimit` and `upperLimit` placeholders that are not known at the time the code is written. By using `lowerLimit` and `upperLimit` placeholders in your code, you are hiding the identity of the lower and upper limits of the range. In other words, you are ignoring their actual values when writing the above piece of code. You have applied the process of abstraction in the above code by ignoring the actual values of the lower and upper limits of the range.

When the above piece of code is executed, the actual values must be substituted for `lowerLimit` and `upperLimit` placeholders. This is achieved in a programming language by packaging the above snippet of code inside a module (subroutine or subprogram) called a procedure. The placeholders are defined as formal parameters of that procedure. Listing 1-2 has the code for such a procedure.

Listing 1-2. A Procedure Named getRangeSum to Compute the Sum of All Integers Between Two Integers

```
int getRangeSum(int lowerLimit, int upperLimit) {
        int sum = 0;
        int counter = lowerLimit;
        while(counter <= upperLimit) {
                sum = sum + counter;
                counter = counter + 1;
        }
        return sum;
}
```

A procedure has a name, which is `getRangeSum` in this case. A procedure has a return type, which is specified just before its name. The return type indicates the type of value that it will return to its caller. The return type is `int` in this case, which indicates that the result of the computation will be an integer. A procedure has formal parameters (possibly zero), which are specified within parentheses following its name. A formal parameter consists of data type and a name. In this case, the formal parameters are named as `lowerLimit` and `upperLimit`, and both are of the data type `int`. It has a body, which is placed within braces. The body of the procedure contains the logic.

When you want to execute the code for a procedure, you must pass the actual values for its formal parameters. You can compute and print the sum of all integers between 10 and 20 as follows:

```
int s1 = getRangeSum(10, 20);
System.out.println(s1);
```

This snippet of code will print 165.

To compute the sum all integers between 40 and 60, you can execute the following snippet of code:

```
int s2 = getRangeSum(40, 60);
System.out.println(s2);
```

This snippet of code will print 1050, which is exactly the same result you had achieved before.

The abstraction method that you used in defining the getRangeSum procedure is called abstraction by parameterization. The formal parameters in a procedure are used to hide the identity of the actual data on which the procedure's body operates. The two parameters in the getRangeSum procedure hide the identity of the upper and lower limits of the range of integers. Now you have seen the first concrete example of abstraction. Abstraction is a vast topic. I will cover some more basics about abstraction in this section.

Suppose a programmer writes the code for the getRangeSum procedure as shown in Listing 1-2 and another programmer wants to use it. The first programmer is the designer and writer of the procedure; the second one is the user of the procedure. What pieces of information does the user of the getRangeSum procedure need to know in order to use it?

Before you answer this question, let's consider a real-world example of designing and using a DVD player (Digital Versatile Disc player). A DVD player is designed and developed by electronic engineers. How do you use a DVD player? Before you use a DVD player, you do not open it to study all the details about its parts that are based on electronics engineering theories. When you buy it, it comes with a manual on how to use it. A DVD player is wrapped in a box. The box hides the details of the player inside. At the same time, the box exposes some of the details about the player in the form of an interface to the outside world. The interface for a DVD player consists of the following items:

- Input and output connection ports to connect to a power outlet, a TV set, etc.
- A panel to insert a DVD
- A set of buttons to perform operations such as eject, play, pause, fast forward, etc.

The manual that comes with the DVD player describes the usage of the player's interface meant for its users. A DVD user need not worry about the details of how it works internally. The manual also describes some conditions to operate it. For example, you must plug the power cord to a power outlet and switch on the power before you can use it.

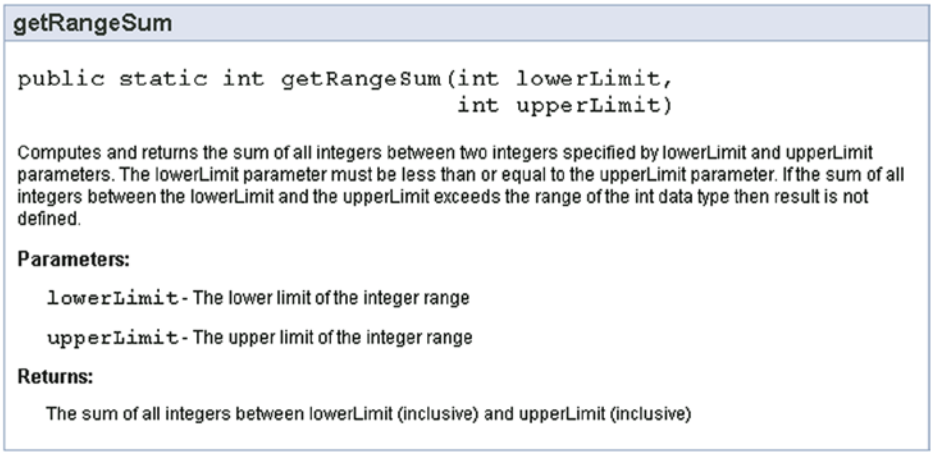
A program is designed, developed, and used in the same way as a DVD player. The user of the program, shown in Listing 1-1, need not worry about the internal logic that is used to implement the program. A user of the program needs to know only its usage, which includes the interface to use it, and conditions that must be met before and after using it. In other words, you need to provide a manual for the getRangeSum procedure that will describe its usage. The user of the getRangeSum procedure will need to read its manual to use it. The "manual" for a program is known as its specification. Sometimes it is also known as documentation or comments. It provides another method of abstraction, which is called abstraction by specification. It describes (or exposes or focuses) the "what" part of the program and hides (or ignores or suppresses) the "how" part of the program from its users.

Listing 1-3 shows the same getRangeSum procedure code with its specification.

Listing 1-3. The getRangeSum Procedure with its Specification for Javadoc Tool

```
/**
 * Computes and returns the sum of all integers between two
 * integers specified by lowerLimit and upperLimit parameters.
 *
 * The lowerLimit parameter must be less than or equal to the
 * upperLimit parameter. If the sum of all integers between the
 * lowerLimit and the upperLimit exceeds the range of the int data
 * type then result is not defined.
 *
 * @param lowerLimit The lower limit of the integer range
 * @param upperLimit The upper limit of the integer range
 * @return The sum of all integers between lowerLimit (inclusive)
 *        and upperLimit (inclusive)
 */
public static int getRangeSum(int lowerLimit, int upperLimit) {
        int sum = 0;
        int counter = lowerLimit;
        while(counter <= upperLimit) {
                sum = sum + counter;
                counter = counter + 1;
        }
        return sum;
}
```

It uses Javadoc standards to write a specification for a Java program that can be processed by the Javadoc tool to generate HTML pages. In Java, the specification for a program element is placed between /** and */ immediately before the element. The specification is meant for the users of the `getRangeSum` procedure. The Javadoc tool will generate the specification for the `getRangeSum` procedure, as shown in Figure 1-4.



getRangeSum

```
public static int getRangeSum(int lowerLimit,
                              int upperLimit)
```

Computes and returns the sum of all integers between two integers specified by lowerLimit and upperLimit parameters. The lowerLimit parameter must be less than or equal to the upperLimit parameter. If the sum of all integers between the lowerLimit and the upperLimit exceeds the range of the int data type then result is not defined.

Parameters:

`lowerLimit` - The lower limit of the integer range

`upperLimit` - The upper limit of the integer range

Returns:

The sum of all integers between lowerLimit (inclusive) and upperLimit (inclusive)

Figure 1-4. *The specification for the getRangeSum procedure*

The above specification provides the description (the "what" part) of the getRangeSum procedure. It also specifies two conditions, known as pre-conditions, that must be true when the procedure is called. The first pre-condition is that the lower limit must be less than or equal to the upper limit. The second pre-condition is that the value for lower and upper limits must be small enough so that the sum of all integers between them fits in the size of the `int` data type. It specifies another condition that is called post-condition, which is specified in the "Returns" clause. The post-condition holds as long as pre-conditions hold. The pre-conditions and post-conditions are like a contract (or an agreement) between the program and its user. It states that as long as the user of the program makes sure that the pre-condition holds true, the program guarantees that the post-condition will hold true. Note that the specification never tells the user about how the program fulfils (implementation details) the post-condition. It only tells "what" it is going to do rather than "how" it is going to do it. The user of the `getRangeSum` program, who has the specification, need not look at the body of the `getRangeSum` procedure to figure out the logic that it uses. In other words, you have hidden the details of implementation of getRangeSum procedure from its users by providing the above specification to them. That is, users of the `getRangeSum` procedure can ignore its implementation details for the purpose of using it. This is another concrete example of abstraction. The method of hiding implementation details of a subprogram (the "how" part) and exposing its usage (the "what" part) by using specification is called abstraction by specification.

Abstraction by parameterization and abstraction by specification let the users of a program view the program as a black box, where they are concerned only about the effects that program produces rather than how the program produces those effects. Figure 1-5 depicts the user's view of the `getRangeSum` procedure. Note that a user does not see (need not see) the body of the procedure that has the details. The details are relevant only for the writer of the program, not its users.

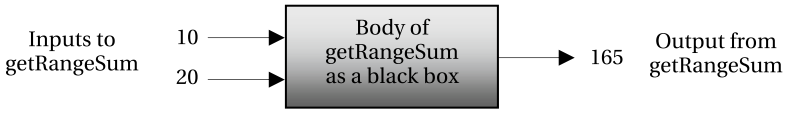


Figure 1-5. *User's view of the getRangeSum procedure as a black box using abstraction*

What advantages did you achieve by applying the abstraction to define the `getRangeSum` procedure? One of the most important advantages is isolation. It is isolated from other programs. If you modify the logic inside its body, other programs, including the ones that are using it, need not be modified at all. To print the sum of integers between `10` and `20`, you use the following program:

```
int s1 = getRangeSum(10, 20);
System.out.println(s1);
```

The body of the procedure uses a `while` loop, which is executed as many times as the number of integers between lower and upper limits. The `while` loop inside the `getRangeSum` procedure executes `n` times where `n` is equal to `(upperLimit - lowerLimit + 1)`. The number of instructions that needs to be executed depends on the input values. There is a better way to compute the sum of all integers between two integers, `lowerLimit` and `upperLimit`, using the following formula:

```
n = upperLimit - lowerLimit + 1;
sum = n * (2 * lowerLimit + (n-1))/2;
```

If you use the above formula, the number of instructions that are executed to compute the sum of all integers between two integers is always the same. You can rewrite the body of the `getRangeSum` procedure as shown in Listing 1-4. The specification of `getRangeSum` procedure is not shown here.

Listing 1-4. Another Version of the getRangeSum Procedure with the Logic Changed Inside its Body

```
public int getRangeSum(int lowerLimit, int upperLimit) {
        int n = upperLimit - lowerLimit + 1;
        int sum = n * (2 * lowerLimit + (n-1))/2;
        return sum;
}
```

Note that the body (implementation or the "how" part) of the getRangeSum procedure has changed between Listing 1-2 and Listing 1-3. However, the users of the getRangeSum procedure are not affected by this change at all because the details of the implementation of this procedure were kept hidden from its users by using abstraction. If you want to compute the sum of all integers between 10 and 20 using the version of the getRangeSum procedure as shown in Listing 1-3, your old code (shown below) is still valid.

```
int s1 = getRangeSum(10, 20);
System.out.println(s1);
```

You have just seen one of the greatest benefits of abstraction, in which the implementation details of a program (in this case, a procedure) can be changed without warranting any changes in the code that uses the program. This benefit also gives you a chance to rewrite your program logic to improve performance in the future without affecting other parts of the application.

I will consider two types of abstraction in this section:

- Procedural abstraction
- Data abstraction

Procedural Abstraction

Procedural abstraction lets you define a procedure, for example, getRangeSum, that you can use as an action or a task. So far, in this section, I have been discussing procedural abstraction. Abstraction by parameterization and abstraction by specification are two methods to achieve procedural abstraction as well as data abstraction.

Object-oriented programming is based on data abstraction. However, I need to discuss data type briefly before I discuss data abstraction. A data type (or simply a type) is defined in terms of three components:

- A set of values (or data objects)
- A set of operations that can be applied to all values in the set
- A data representation, which determines how the values are stored

Programming languages provide some predefined data types, which are known as built-in data types. They also let programmers define their own data types, which are known as user-defined data types. A data type that consists of an atomic and indivisible value, and that is defined without the help of any other data types, is known as a primitive data type. For example, Java has built-in primitive data types such as int, float, boolean, char, etc. Three components that define the int primitive data type in Java are as follows:

- An int data type consists of a set of all integers between -2147483648 and 2147483647.
- Operations such as addition, subtraction, multiplication, division, comparison, and many more are defined for the int data type.
- A value of int data type is represented in 32-bit memory in 2's compliment form.

All three components of the int data type are predefined by Java language. You cannot extend or redefine the definition of the int data type as a programmer. You can give a name to a value of the int data type as

```
int n1;
```

The above statement states that n1 is a name (technically called an identifier) that can be associated with one value from the set of values that defines values for int data type. For example, you can associate integer 26 to the name n1 using an assignment statement as

```
n1 = 26;
```

At this stage, you may be asking, “Where is the value 26, which is associated with the name n1, stored in memory?” You know from the definition of int data type that n1 will take 32-bit memory. However, you do not know, cannot know, and do not need to know where in the memory that 32-bit is allocated for n1. Do you see an example of abstraction here? If you see an example of abstraction in this case, you are right. This is an example of abstraction, which is built into the Java language. In this instance, the pieces of information about the data representation of the data value for int data type are hidden from the users (programmers) of the data type. In other words, a programmer ignores the memory location of n1 and focuses on its value and the operations that can be performed on it.
A programmer does not care if the memory for n1 is allocated in a register, RAM, or the hard disk.

Data Abstraction

Object-oriented programming languages such as Java let you create new data types using an abstraction mechanism called data abstraction. The new data types are known as abstract data types (ADT). The data objects in ADT may consist of a combination of primitive data types and other ADTs. An ADT defines a set of operations that can be applied to all its data objects. The data representation is always hidden in ADT. For users of an ADT, it consists of operations only. Its data elements may only be accessed and manipulated using its operations. The advantage of using data abstraction is that its data representation can be changed without affecting any code that uses the ADT.

■ **Tip** Data abstraction lets programmers create a new data type called an abstract data type, where the storage representation of the data objects is hidden from the users of the data type. In other words, ADT is defined solely in terms of operations that can be applied to the data objects of its type without knowing the internal representation of the data. The reason this kind of data type is called abstract is that users of ADT never see the representation of the data values. Users view the data objects of an ADT in an abstract way by applying operations on them without knowing the details about representation of the data objects. Note that an ADT does not mean absence of data representation. Data representation is always present in an ADT. It only means hiding of the data representation from its users.

Java language has constructs, for example, class, interface, and enum, that let you define new ADTs. When you use a class to define a new ADT, you need to be careful to hide the data representation, so your new data type is really abstract. If the data representation in a Java class is not hidden, that class creates a new data type, but not an ADT. A class in Java gives you features that you can use to expose the data representation or hide it. In Java, the set of values of a class data type are called objects. Operations on the objects are called methods. Instance variables (also known as fields) of objects are the data representation for the class type.

A class in Java also lets you provide an implementation of operations that operates on the data representation. An interface in Java lets you create a pure ADT. An interface lets you provide only the specification for operations that can be applied to the data objects of its type. No implementation for operations or data representation can be mentioned in an interface. Listing 1-1 shows the definition of the Person class using Java language syntax. By defining a class named Person, you have created a new ADT. Its internal data representation for name and gender uses String data type (String is built-in ADT provided by Java class library). Note that the definition of the Person class uses the private keyword in the name and gender declarations to hide it from the outside world. Users of the Person class cannot access the name and gender data elements. It provides four operations: a constructor and three methods (getName, setName, and getGender).

A constructor operation is used to initialize a newly constructed data object of Person type. The getName and setName operations are used to access and modify the name data element, respectively. The getGender operation is used to access the value of the gender data element.

Users of the Person class must use only these four operations to work with data objects of Person type. Users of the Person type are oblivious to the type of data storage being used to store name and gender data elements. I am using three terms, "type," "class," and "interface," interchangeably because they mean the same thing in the context of a data type. It gives the developer of the Person type freedom to change the data representation for the name and gender data elements without affecting any users of Person type. Suppose one of the users of Person type has the following snippet of code:

```
Person john = new Person("John Jacobs", "Male");
String intialName = john.getName();
john.setName("Wally Jacobs");
String changedName = john.getName();
```

Note that this snippet of code has been written only in terms of the operations provided by the Person type. It does not (and could not) refer to the name and gender instance variables directly. Let's see how to change the data representation of the Person type without affecting the above snippet of code. Listing 1-5 shows the code for a newer version for the Person class.

Listing 1-5. Another Version of the Person Class That Uses a String Array of Two Elements to Store Name and Gender Values as Opposed to Two String Variables

```
package com.jdojo.concepts;

public class Person {
        private String[] data = new String[2];

        public Person(String initialName, String initialGender) {
                data[0] = initialName;
                data[1] = initialGender;
        }

        public String getName() {
                return data[0];
        }

        public void setName(String newName) {
                data[0] = newName;
        }

        public String getGender() {
                return data[1];
        }
}
```

Compare the code in Listing 1-1 and Listing 1-5. This time you have replaced the two instance variables (name and gender), which were the data representation for the Person type in Listing 1-1, with a String array of two elements. Since operations (or methods) in a class operate on the data representation, you had to change the implementations for all four operations in the Person type. The client code in Listing 1-5 was written in terms of the specifications of the four operations and not their implementation. Since you have not changed the specification of any of the operations, you do not need to change the snippet of code that uses the Person class; it is still valid with the newer definition of the Person type as shown in Listing 1-5. Some methods in the Person class use the abstraction by parameterization and all of them use the abstraction by specification. I have not shown the specification for the methods here, which would be Javadoc comments.

You have seen two major benefits of data abstraction in this section.

- It lets you extend the programming language by letting you define new data types. The new data types you create depend on the application domain. For example, for a banking system, `Person`, `Currency`, and `Account` may be good choices for new data types whereas for an auto insurance application, `Person`, `Vehicle`, and `Claim` may be good choices. The operations included in a new data type depend on the need of the application.
- The data type created using data abstraction may change the representation of the data without affecting the client code using the data type.

Encapsulation and Information Hiding

The term encapsulation is used to mean two different things: a process or an entity. As a process, it is an act of bundling one or more items into a container. The container could be physical or logical. As an entity, it is a container that holds one or more items.

Programming languages support encapsulations in many ways. A procedure is an encapsulation of steps to perform a task; an array is an encapsulation of several elements of the same type, etc. In object-oriented programming, encapsulation is bundling of data and operations on the data into an entity called a class.

Java supports encapsulation in various ways.

- It lets you bundle data and methods that operate on the data in an entity called a class.
- It lets you bundle one or more logically related classes in an entity called a package. A package in Java is a logical collection of one or more related classes. A package creates a new naming scope in which all classes must have unique names. Two classes may have the same name in Java as long as they are bundled (or encapsulated) in two different packages.
- It lets you bundle one or more related classes in an entity called a compilation unit. All classes in a compilation unit can be compiled separately from other compilation units.

While discussing the concepts of object-oriented programming, the two terms, encapsulation and information hiding, are often used interchangeably. However, they are different concepts in object-oriented programming, and they should not be used interchangeably as such. Encapsulation is simply the bundling of items together into one entity. Information hiding is the process of hiding implementation details that are likely to change. Encapsulation is not concerned with whether the items that are bundled in an entity are hidden from other modules in the application or not. What should be hidden (or ignored) and what should not be hidden is the concern of abstraction. Abstraction is only concerned about which item should be hidden. Abstraction is not concerned about how the item should be hidden. Information hiding is concerned with how an item is hidden. Encapsulation, abstraction, and information hiding are three separate concepts. They are very closely related, though. One concept facilitates the workings of the others. It is important to understand the subtle differences in roles they play in object-oriented programming.

It is possible to use encapsulation with or without hiding any information. For example, the `Person` class in Listing 1-1 shows an example of encapsulation and information hiding. The data elements (`name` and `gender`) and methods (`getName()`, `setName()`, and `getGender()`) are bundled together in a class called `Person`. This is encapsulation. In other words, the `Person` class is an encapsulation of the data elements `name` and `gender`, plus the methods `getName()`, `setName()`, and `getGender()`. The same `Person` class uses information hiding by hiding the data elements from the outside world. Note that `name` and `gender` data elements use the Java keyword `private`, which essentially hides them from the outside world. Listing 1-6 shows the code for a `Person2` class.

Listing 1-6. The Definition of the Person2 Class in Which Data Elements Are Not Hidden by Declaring Them Public

```
package com.jdojo.concepts;

public class Person2 {
        public String name;   // Not hidden from its users
        public String gender; // Not hidden from its users

        public Person2(String initialName, String initialGender) {
                name = initialName;
                gender = initialGender;
        }

        public String getName() {
                return name;
        }

        public void setName(String newName) {
                name = newName;
        }

        public String getGender() {
                return gender;
        }
}
```

The code in Listing 1-1 and Listing 1-6 is essentially the same except for two small differences. The `Person2` class uses the keyword `public` to declare the `name` and the `gender` data elements. The `Person2` class uses encapsulation the same way the `Person` class uses. However, data elements `name` and `gender` are not hidden. That is, the `Person2` class does not use data hiding (Data hiding is an example of information hiding). If you look at the constructor and methods of `Person` and `Person2` classes, their bodies use information hiding because the logic written inside their bodies is hidden from their users.

■ **Tip** Encapsulation and information hiding are two distinct concepts of object-oriented programming. The existence of one does not imply the existence of the other.

Inheritance

Inheritance is another important concept in object-oriented programming. It lets you use abstraction in a new way. You have seen how a class represents an abstraction in previous sections. The `Person` class shown in Listing 1-1 represents an abstraction for a real-world person. The inheritance mechanism lets you define a new abstraction by extending an existing abstraction. The existing abstraction is called a supertype, a superclass, a parent class, or a base class. The new abstraction is called a subtype, a subclass, a child class, or a derived class. It is said that a subtype is derived (or inherited) from a supertype; a supertype is a generalization of a subtype; and a subtype is a specialization of a supertype. The inheritance can be used to define new abstractions at more than one level. A subtype can be used as a supertype to define another subtype and so on. Inheritance gives rise to a family of types arranged in a hierarchical form.