Harnessing Java™ 7

A Comprehensive Approach to Learning Java™

Volume - 1

Kishori Sharan

SAMPLE CHAPTERS

Programming Concepts and Arrays

© 2011 Kishori Sharan

All rights reserved. No part of this book may be reproduced or transferred in any form or by any means, graphic, electronic, or mechanical, including photocopying, recording, taping, or by any information storage retrieval system, without the written permission of the author.

The author has taken great care in the preparation of this book. This book could include technical inaccuracies or typographical errors. The information in this book is distributed on an “as is” basis, without any kind of expressed or implied warranty. The author assumes no responsibility for errors or omissions in this book. The accuracy and completeness of information provided in this book are not guaranteed or warranted to produce any particular results. The author shall not be liable for any loss incurred as a consequence of the use and application, directly or indirectly, of any information presented in this book.

Trademarks

Trademarked names may appear in this book. Trademarks and product names used in this book are the property of their respective trademark holders. The author of this book is not affiliated with any of the entities holding those trademarks that may be used in this book.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Cover Design & Images By: Richard Castillo (http://www.digitizedchaos.com)

Printing History:

August 2011 First Print

ISBN-10: 1-46-376771-4

ISBN-13: 978-1-463-76771-6

Dedicated

To

Three of my college professors who have influenced my academic career the most:

Sri Subir Kumar Roy,

Late Sri Madan Sharma, and

Sri Subhash Sareen

Table of Contents

Preface

My first encounter with the Java programming language was during a one-week Java training session in 1997. I did not then 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 in the test by scoring ninety five 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 about the Java programming language. So, in 2001, 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 seven hundred to eight hundred pages.

As I progressed, I realized that a book covering most of the Java topics in detail could not be written in seven to eight hundred pages. One chapter alone that covered data types, operators, and statements spanned ninety 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 publish the book, because it could have compromise the quality 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 keeping in mind that this book 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, because a building is made up of these smaller things. The same logic applies to most of the things that we want to understand in our lives. It also applies to an understanding of the Java programming language. If you want to master the Java programming language, you must start with understanding its basic building blocks. I have used this approach throughout this book endeavoring to build each topic by describing the basics first. In this 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 over two hundred and sixty graphical representations in this book (spanning three volumes) to aid readers in understanding and visualizing its contents. Developers who have little or no programming experience have difficulty in putting things together to make it a complete program. Keeping those developers in mind, I have included over five hundred complete Java programs that are ready to be compiled and run.

As I finished a chapter, I distributed copies to Java students and developers to get their feedback. Their feedback included a common observation that the material in this book is simple yet detailed. That kept me motivated to write the succeeding chapters. One feedback received in the beginning was about the coverage of setting the classpath in this book. I have seen some Java developers with quite a bit programming experience struggle with setting the classpath for a Java application. Most of the developers start programming using a Java editor. Java editors make it easy for the developers to set the classpath. Most of the time, a developer is unaware of the classpath settings when he uses a Java editor. In reality, a Java developer has to debug issues that are related to classpath settings on a machine or an application server. Keeping the principle of describing the basics of a topic, I devoted a chapter on writing a very basic Java program (the chapter name is *Writing Java Programs*) that also describes setting the classpath in detail. I gave this chapter to

many Java students and some Java developers with over five years of experience. All of them reported, “Now, I know how to work with the classpath.”

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 of the topic. Finally, it was always fun to play with Java programs, sometimes for hours, to add them to the book.

I encountered many hurdles and pauses (some long ones) along the way of writing this book. I registered for a master degree program after I finished a few chapters. As I was working on my master program, I could not work on the book for over two years. Sometimes, the extra workload at work prevented me from doing any work on this book for months. It took me ten years (You read it right. Ten years is called a decade.) to finish this book. If I had devoted all my time on writing this book, I could say that it would have taken me about two years to finish it. I also had to keep adding new material to cover the newer versions of Java. I started writing this book using Java 1.2 and finished it using Java 1.7. Finally, it turned out to be almost a two thousand page book, which had to be split in three volumes, because of the restrictions on the number of pages a print-on-demand book can have. At this point, all I can say is, “All’s well that ends well.”

Kishori Sharan

Structure of the Book

This book contains thirty-four chapters and three appendixes spread across three volumes. The print-on-demand technology puts a restriction on the maximum number of pages in a book. This made me divide the book into three volumes. To get the most out of this book, a reader is suggested to read from the first chapter to the last. Each chapter builds upon the previous chapters. Volumes and chapters inside a volume have been arranged in a way that presents the most basic material about the Java programming language first. Sections in a chapter are arranged in an order of increasing complexity, the least complex section being the first. The following is the list of topics covered in the three volumes.

Volume - 1

- Programming Concepts
- Data Types
- Operators
- Statements
- Classes & Objects
- Object and Objects Classes
- AutoBoxing
- Exception Handling
- Assertions
- Strings & Dates
- Formatting Objects
- Regular Expressions
- Arrays
- Garbage Collection
- Inheritance

Volume - 2

- Interfaces
- Annotations
- Inner Classes
- Enum
- Reflection
- Generics
- Threads
- Input/Output
- Archive Files
- Collections

Volume - 3

- Swing
- Applets
- Network Programming
- JDBC API
- Remote Method Invocation
- Java Native Interface

Audience

This book is designed to be useful for anyone who wants to learn about 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 them later, when you gain more experience.

If you are a Java developer with 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. You may only want to read volumes of this book that cover the topics of your interest.

If you are reading this book to get a certification in the Java programming language, you need to read almost all 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 five hundred 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 in Volume 1, 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, as a Java student, you 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.

> *Be curious always, for knowledge will not acquire you; you must acquire it. - Sudie Back*
>
> *Knowledge comes by eyes always open and working hard, and there is no knowledge that is not power. - Jeremy Taylor*
>
> *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 website of Oracle Corporation at *http://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 - “Practice makes a person perfect”, which is also true in learning how to program in Java.

Java 7 New Features

Java 7 has added many new language level features. This book covers all Java 7 language level new features. A complete chapter in Volume - 2 has been devoted to discuss the NIO 2.0 in detail. The new features in Java 7 have been discussed in the related chapters. The following is the list of the new features of Java 7 covered in three volumes of this book.

Volume - 1

- Binary Numeric Literals
- Underscores in Numeric Literals
- Strings in a switch Statement
- try-with-resources Statement
- Catching Multiple Exception Types
- Re-throwing Exceptions with Improved Type Checking
- The java.util.Objects class

Volume - 2

- Generic Type Inference
- Heap Pollution
- Improved Compiler Warnings and Errors When Using Non-Reifiable Formal Parameters with Varargs Methods
- Improved File and Channel Closing Mechanism using a try-with-resources Statement
- New Input/Output 2.0 (NIO 2.0)
- Fork/Join Framework
- Phaser Synchronization Barrier
- TransferQueue Collection

Volume - 3

- JLayer Swing Component
- Translucent Window
- Shaped Window
- Asynchronous Socket IO
- Multicast DatagramChannel
- RowSetFactory

Acknowledgements

This book could not have been written without the encouragements, supports and contributions from many people.

First, I would like to thank the authors of all the books, articles, white papers and Java Specification Requests (JSRs) related to the Java programming language that I read and consulted to gain my own knowledge of the Java programming language.

My heartfelt thanks are due to my father-in-law Mr. Jim Baker for displaying extraordinary patience in proof reading the book. I am very grateful to him for spending so much of his valuable time teaching me quite a bit of English grammar that helped me in producing better material, and hence less work for him during his proof reading sessions. I would also like to thank my mother-in-law Ms. Kim Baker for providing him delicious food (including letting him eat ice cream) and regularly reminding him to finish proof reading this book.

My wife Ellen was always patient when I spent long hours at my computer desk working on this book. She would happily bring me snacks, fruit, and a glass of water every thirty minutes or so to sustain me during that period. I want to thank her for all her support in writing this book. She also deserves my sincere thanks for proofreading many of the chapters and providing valuable feedback.

I would like to thank my sister-in-law Patty Boyd for cooking delicious food for me while I worked on the book. Thanks also go to my brother-in-law Jeff Boyd and my nephew Christopher Estes for helping me in many ways to save my time, so that I can focus on the book.

I would like to thank my friend Kannan Somasekar for his support and hard work to get an appropriate subtitle for this book. I would also like to thank him for time spent in researching the possible publishing options. His research helped me choose the print-on-demand publishing option. I would like to thank Kannan's wife, Divya Somasekar, for taking care of their two lovely sons, Krish and Rishi, while Kannan spent time at his computer desk to help me finish this book.

My special thanks go to my friend and colleague, Richard Castillo, for proof reading this book very thoroughly. He deserves a big thank-you for designing the cover pages and suggesting the title of the book.

I would like to thank my friends and colleagues Christopher Coley, Tanu Mutreja, Rahul Jain, Raju Mudunuri, Willie Baptiste, Tejas Dholakia, Amita Dholakia, Lorenzo Braxter, and Mahbub Chowdhury, for providing valuable feedback. Willie Baptiste, Tejas Dholakia, Amita Dholakia, Lorenzo Braxter, and Mahbub Chowdhury deserve little extra thanks for giving me the opportunity to teach them Java, which gave me more insight on how to explain things in the book to make it easier for the readers to understand.

There is one good thing about members of anybody's family, including mine. Once you tell them that you are working on writing a book, they will remind you periodically about the status of your book, which keeps you always aware that you have one unfinished job at your hand and you must finish it sometime in your life! Finishing this book was not possible without the blessings of my parents, Ram Vinod Singh and Pratibha Devi, and my elder brothers, Janki Sharan and Dr. Sita Sharan. My most sincere and heartfelt thanks go to them for their support and encouragement during the entire period I was working on the book. I would also like to thank my sister Ratna Kumari and brother-in-law Abhay Kumar Singh, my nephew Navin Kumar and daughter-in-law Anjali Singh, my niece Vandana Kumari and son-in-law Prem Prakash, my nephew Neeraj Kumar and daughter-in-law Pallavi, and my nephews Gaurav Sharan and Saurav Kumar, for their interest.

I would like to thank Chitranjan Sharma, my nephew and a student of Bachelor of Computer Applications at Gaya College Gaya, for his diligent efforts to learn Java using this book and providing me valuable feedback.

I would like to thank the following colleagues for their support, encouragement and for being always supportive, while I worked with the Department of Children Service, State of Tennessee: John Jacobs, Reddy Matta, Donna Duncan, Katrina Hills, LaTondra Okeke, Prasanna Despande, Geshan Alvis, Buddy Rice, Jack Parker, Jags.Pinni, Bettye Clark, Charles O' Riley, Basir Kabir, Barbara Gentry, Paula Daugherty, Dinesh Sankala, Elaine Blaylock, Lisa Simmons, Deborah Hurd, Wallace Inman, Pravin Lokhande, Priyanka Sharma, Amitabh Sharma, and Wanda Jackson. I know that some of them did not know that I had been writing a book. But, I feel they deserve mention, because it helped me in the writing of this book at home, when they all were nice and supportive at work. After all, good attitude is contagious!

I would like to thank the following colleagues for their support and helping me learn the intricacies of Java while I worked for Kingsway America in Mobile, Alabama: Larry Brewster, Biju Nair, Jim Jacobs, Ram Atmakuri, Srinivas Kakerra, James Pham, Megan Bodiford, Udhaya Kumar, Matt Flowers, and Greg Langham.

I would like to thank the following colleagues for their support while I worked at ProAssurance in Birmingham, Alabama: LaRonda Lanier, Christy Mueller-Smith, Darren Jackson, Bob Waldrop, Tatiana Telioukova, Russell Thomas, Cameron Ellison, Brandon Russell, Devaraj Rajan, Frank Lay, Andriy Shlykov, Andy Purvis, Rob Ballard, Marty Heim, and Troy Dotson. As I have mentioned earlier, some of my colleagues may not even be aware that I have been writing a book. However,

their support in creating a cordial and helping working environment at work helped me carrying the good frame of mind to home in the evening to spend a few hours of my time on this book. So, all of you deserve my sincere thanks.

I would like to thank the following managers for their support while I worked in different projects: Robert Holloway, Connie Spradlin, Ed Bennett, Kieran Cloonan, and Donovan Fitzgerald at Department of Children Services, Nashville, TN; Leslie Zanders, Cheryl Lawrence, and Kelly Dumas at Kingsway America in Mobile, AL; Kirby Sims, Douglas Dyer, Brian Russell, Lael Boyd, and Vivi Gin at ProAssurance in Birmingham, AL; Heath Wade and Amy Gartman at Doozer Inc.

I would like to thank the following friends for their support and encouragement: Rahul Nagpal, Ravi Datla, Anil Kumar Singh, Balram Kumar, Dilip Kumar, Ramta Prasad Singh, Pratap Chandra, Sanjeev Choudhary, Pramod Kumar, Prakash Chandra, Dharmendhra Kumar Mishra, Rajeev Kumar Verma, Randy Lucas, Sanjay Pandey, Suman Kumar Singh, Amarjeet Kumar, Vijay Kumar Tarun, Raju Mishra, Jayshankar Prasad Singh, Mukesh Sinha, Rajesh Kumar, Vishwa Mohan, Ranjeet Ekka, Sanjay Singh, Kamal Singh, Pankaj Kumar, Ranjeet Kumar, Krishna Kumar, and Anuj Sinha.

Source Code and Errata

Source code and errata for this volume may be downloaded from *http://www.jdojo.com*.

Questions and Comments

Please direct all your questions and comments to *ksharan@jdojo.com*

.

Chapter 1. Programming Concepts

What is Programming?

The term “programming” is used in many contexts. We will discuss the meaning of this term 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 notation 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 pre-defined set of tasks that it can perform? Before we look at human-to-computer communication, let us look at human-to-human communication. How does a human communicate with another human? You would say that a human-to-human communication is accomplished using a spoken language e.g. English, German, Hindi, etc. 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 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 other party. For example, suppose there are two persons. One person knows how to speak English and 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 - sequence of zeros and ones. 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 as 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 is very difficult, if not impossible, to write, read, understand and modify. Suppose we 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. They are only for the purpose of discussion and illustration.

```
0010010010  10010100000100110
0001000100  01010010001001010
```

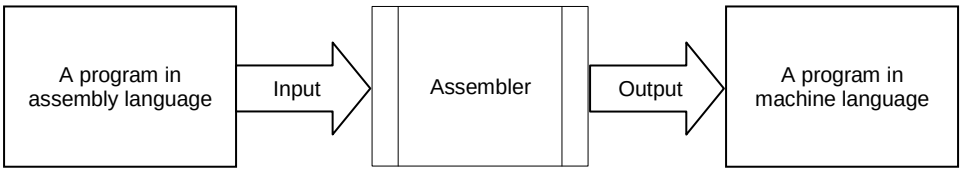
The above instructions are to add two numbers. How difficult will it be to write a program in machine language to perform a complex task? Now, you may realize that it is difficult to write, read, and understand a program written in a machine language. Computers are supposed to make our

jobs easier and not more difficult. We needed to represent the instructions for computers in some notations that were easier to write, read and understand. Computer scientists came up with another language called *assembly language*. Assembly language provides different notations to write instructions for computers. It is little easier to write, read and understand than its predecessor machine language. Assembly language uses mnemonics to represent instructions as opposed to binary (0s and 1s) used in machine language. A program written in assembly language to add two numbers look similar to those shown below.

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

If you compare the programs written in machine language and assembly language to perform the same task, you would realize that assembly language is easier to write, read and understand than the machine language. 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, which consists of 0s and 1s. The instructions that are written in an assembly language must be translated to the machine language before they can be executed by the computer. A program that translates the instructions written in an assembly language to a machine language is called an *assembler*. Figure 1-1 shows the relationship between assembly code, an assembler, and machine code.

Figure 1-1: Relationship between assembly code, assembler and machine code



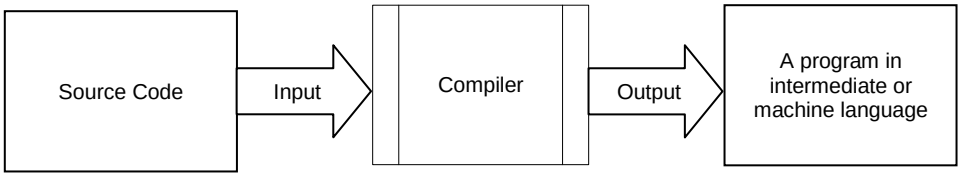
Machine language and assembly language are also known as *low-level languages*. They are called low-level languages, because a programmer must understand low-level details of the computer to write a program those languages. For example, if you were writing programs in machine and assembly languages, you 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 high-level programming language that will hide the low-level details of computers from them. The need to hide the low-level details of computers from programming 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 notations and punctuations to write a program. A program written in a high-level programming language is also called *source code*. They are closer to written languages that humans are familiar with. The instructions to add two numbers can be written in a high-level programming language e.g. Java, which looks similar to the following.

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

You may notice that the programs written in a high-level language is 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. We need a means to translate a program written in high-level language to machine language. The translation of the program written in a high-level programming language to machine language is accomplished by a compiler, an *interpreter* or combination of both. A compiler is a program that translates a program 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, which is called a Java Virtual Machine (JVM), 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. Sometimes, another kind of compiler, which is called ahead-of-time compiler (AOT), is used to compile a program in 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: Relationship between a source code, a compiler, and a machine code



Components of a Programming Language

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

- Syntax
- Semantics and
- 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 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 vocabulary of the language to form valid programming constructs. You can think of a valid programming construct in programming language like a sentence in a written language. A sentence in a written language is formed using vocabulary and grammar of the language. Similarly, a programming construct is formed using vocabulary and the grammar of the programming language. The vocabulary and rules to use vocabulary to form valid programming constructs is known as *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*. Semantics of a programming language explains the meaning of the syntactically valid programming constructs. The semantics of a programming language answers 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 compute.

The pragmatics of a programming language describes 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 is 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 little hard to understand the word "paradigm" in programming context. Programming is about providing a solution to a real-world problem using computational models supported by the programming language. The solution to the real-world problem that we provide in terms of computational models is called a program. Before we provide a solution to a problem in the form of a program, we always have a mental view about the problem and its solution. Before we discuss how we solve a real-world problem using computational model, let us take an example of a real-world social problem, which has nothing to do with computers. Suppose there is a place on the earth that has a food problem. People in that place do not have enough food to eat. The problem is, "shortage of food." Let us 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 the 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, philanthropist 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 and the best solution. 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 all kinds 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 is a reputed computer scientist. He gave this definition in his 1979 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 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. 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

Imperative paradigm is also known as algorithmic paradigm. In 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 are 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 imperative language to add 10 to 15 as follows. 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 code inside 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 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 algorithm is permanent.

Procedural Paradigm

Procedural paradigm is similar to imperative paradigm with one difference that 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 procedural language consists of data and a sequence of procedure calls that manipulate the data. The following piece of code is a 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`. We will 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 codes 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 imperative paradigm. In 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 effect. It modifies the data part of the program as it executes its logic.

Declarative Paradigm

In 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 computer's job to arrive at a solution when a problem is described to it. Contrast the declarative paradigm with imperative paradigm. In imperative paradigm, we are concerned about the "how" part of the problem. In 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. Functional paradigm and logic paradigm, which are described next, are examples of declarative paradigm. Writing a database query using structured query language (SQL) falls under programming based on declarative paradigm where you specify what data you want and the database engine figures out how to retrieve the `data for you`. Unlike imperative paradigm, the data is permanent and the algorithm is transient in declarative paradigm. In imperative paradigm, the data is modified as the algorithm is executed. In declarative paradigm, data is supplied to the algorithm as input and the input data remains unchanged as the algorithm is executed. Algorithm produces new data rather than modifying the input data. In other words, in declarative paradigm, execution of an algorithm does not produce side effects.

Functional Paradigm

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 input. Unlike a procedure used 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 assignment, which are used for modifying data. In imperative programming, a repeated task is performed using a loop construct e.g. 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. 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
```

Logic Paradigm

Unlike the imperative paradigm, 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. Algorithm is of less importance to the programmer. The primary task of the programmer is to describe the problem as closely as possible. In 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, which makes 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 *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`, as defined below.

```
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 below.

```
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 and it 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 `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 the persons 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 programming language that supports logic paradigm.

Object-Oriented Paradigm

In object-oriented (OO) paradigm, a program consists of interacting objects. An object encapsulates data and algorithms. Data define 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 object-oriented paradigm with imperative and functional paradigms. In imperative and functional paradigms, data and algorithms are separated, whereas in object-oriented paradigm, data and algorithms are not separate. Rather, they are combined in one entity, which is called object.

Classes are the basic units of programming in the object-oriented paradigm. Similar objects are grouped into one definition called *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 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 we want to represent real-world persons in our program. We will create a `Person` class and its instances will represent persons in our program. The `Person` class can be defined as listed in Listing 1-1. This example uses syntax of Java programming language. You do not need to understand the syntax used in the programs that we are writing at this point. We will discuss the syntax to define classes and create objects in subsequent chapters.

Listing 1-1: The definition of the Person class whose instances represent real-world persons in a program

```
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 object 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 real-world, a person possesses many properties, e.g., 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, which are relevant to the system being modeled. For our current demonstration purpose, we decided to 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 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 our `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 same as asking a question to a person – "What is your name?" and he responds by telling 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 name of `donna` object from `Donna Duncan` to `Donna Jacobs`.

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

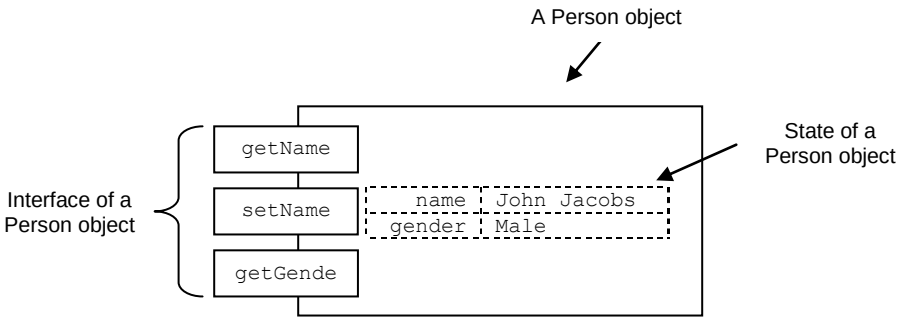
If you send the `getName` message to `donna` object at this point, it will return `Donna Jacobs` and not `Donna Duncan`.

You may notice that our `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..

In object-oriented paradigm, the state of an object is kept private to the object. That is, users of the object (or client code) cannot access/modify the state of the object directly. Users of the object use one of the interfaces provided by the object to access or modify the state of the object. The methods defined in the class that are accessible to the client code constitute the interface for objects. Figure 1-3 shows the state and interface of the `Person` object called `john`.

TIP
In object-oriented paradigm, an object consists of state (data) and interface (behavior). The users of the object interact with it using the interface. The state of the object cannot be accessed directly from outside the object. Interfaces are used to access/modify the state of the object.

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



The object-oriented paradigm is a very powerful paradigm to model real-world phenomena in computational model. We are used to working with objects all around us in our daily-life. It is taught that object-oriented paradigm is natural and intuitive as it lets you think in terms of objects. However, it does not give you ability to think in terms of objects correctly. Sometimes, solution to a problem does not fall into the domain of object-oriented paradigm. In such cases, you need to use the paradigm that suits the problem domain the most. 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 object-oriented paradigm. You must understand and be able to use these features to take the full advantage of the object-oriented paradigm. We will discuss these features of Object-oriented paradigm in the sections to follow. In subsequent chapters, we will discuss these features and how to implement them in a program in detail.

To name a few, C++, Java and C# (pronounced as CSharp)) are programming languages that support 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 object-oriented paradigm as well as procedural paradigm. You often read a phrase like – “Java is an object-oriented programming language”. What is meant is that Java language has features that support 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 procedural paradigm. You can write a program in Java, which is a 100% procedural program without an iota of object-orientedness in it,

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

Soon after its release in 1995, Java became a very popular programming language. One of the most important features for its popularity was its “Write Once, Run Everywhere” (WORE) feature. This feature lets you write a Java program once, which can be run on any platform. For example, you can write and compile a Java program on UNIX and run it on Microsoft Windows, Macintosh or

UNIX without any modifications to the source code. WORE is achieved by compiling a Java program into an intermediate language called *bytecode*. The format of bytecode is platform-independent. A virtual machine, which is called 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 “virtual” machine. The job of 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.

Java became a very popular programming language in a very brief period after its first release in 1995. Following are a few characteristics of Java behind its popularity and acceptance in 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 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 C++ experience you had. Java retained most of the syntax of C/C++, which helps a C/C++ programmer learn the new language. 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 were included in C++. If you are learning Java as your first programming language, whether it is still a simple language to learn may not be true for you. This is the reason, we stated in the beginning of this section that simplicity of Java or any programming language is very subjective.

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 HyperText Markup Language (HTML), and it 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 HTML page 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 application. A concurrent application has multiple interacting threads of execution running in parallel. We will discuss of these features of Java language in details 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. 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 compile-time error does not compile into bytecode until the errors are corrected. Missing a semi-colon at the end of a statement, assigning a decimal value, say 10.23, to a variable of integer type, etc. are the examples of compile-time errors.

Runtime errors occur when a Java program is run. This kind of error is not detected by compiler because a compiler does not have all 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, 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 de-allocation. The feature of automatic memory de-allocation 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 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, few logic errors sneak into production environment and they go unnoticed even after the application is decommissioned.

An error in a program is known a bug. The process of finding and fixing bugs in a program is known as debugging. All modern Integrated Development Environments (IDE) such as NetBeans, Eclipse, JDeveloper, JBuilder, etc, provide programmers with a tool called 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 debuggers that comes with the development tools that you use to develop your Java programs.

Object-Oriented Paradigm and Java

Abstraction

A program provides solutions to a real-world problem using computational model. A program size may range from a few lines to a few millions of lines. A program may be written as a monolithic structure running from the first line to 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 maintainability, a big monolithic program must be decomposed into smaller sub-programs. The sub-programs are then assembled together to solve the original problem. Care must be taken when a program is being decomposed into sub-programs. All sub-programs must be simple and small enough to be understood by themselves. When they are assembled together, they must solve the original problem.

Let us consider the following requirement for a device.

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

One way to design such a device is to provide a keyboard that has keys for all possible combinations of all alphabets, 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 an alphabet, a digit or a symbol one at a time, which represents the smaller part of the original problem. If you can type all alphabets, 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 vertical line. A user can use the two keys combinations

to type in `E`, `T`, `I`, `F`, `H`, and `L` because these alphabets consist of only horizontal and vertical lines. With this solution, a user can type six alphabets using the combination of just two keys. However, with your experience using keyboard you may realize that decomposing the keys so that a key can be used to type in only part of an alphabet is not a reasonable solution although it is a valid solution.

Why is providing two keys to type six alphabets 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. Our reasoning behind calling it "not reasonable" is that it is not easily understood by users. It exposes more details to its 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 alphabet, he does not have to deal with these details. It is important that the sub-programs, which 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 sub-program 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 sub-programs, a modification in a sub-program should not affect other sub-programs. Sub-programs can also be further decomposed by focusing on different level of details and ignoring other details. A good decomposition of a program aims at providing the following characteristics.

- Simplicity
- Isolation
- Maintainability

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

Each sub-program 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 which 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 not about removing or hiding details from a problem. It 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 of some details of a problem. Abstraction is concerned about what details of a thing that should be considered and what 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 ignoring the irrelevant details. As an entity, it is a particular view of a problem, which considers some relevant details and ignores the irrelevant details.

Let us discuss application of abstraction in a real-world programming. Suppose we want to write a program that will compute the sum of all integers between two integers. Suppose we want to compute the sum of all integers between 10 and 20. We 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);
```

The above snippet of code will add `10 + 11 + 12 + … + 20` and print `165`. Suppose we 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);
```

The above 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 same in both. However, the lower limit and the upper limit of the range are different. If we can ignore the differences that exist between the two snippets of code, we will be able to avoid the duplicating of logic at two places. Let us 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, we did not use any actual values for lower and upper limits of any range. Rather, we have used `lowerLimit` and `upperLimit` placeholders that are not known at the time the code is written. By using `lowerLimit` and `upperLimit` placeholders in our code, we are hiding the

identity of the lower and upper limits of the range. In other words, we are ignoring their actual values when writing the above piece of code. We 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 sub-program) called 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 our 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 our case, which indicates that the result of 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 our 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 we want to execute the code for a procedure, we must pass the actual values for its formal parameters. We can compute and print the sum of all integers between `10` and `20` as follows.

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

The above snippet of code will print `165`. To compute the sum all integers between `40` and `60`, we can execute the following snippet of code.

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

The above snippet of code will print `1050`, which is exactly the same result we had achieved before.

The abstraction method that we used in defining `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 `getRangeSum` procedure hide the identity of the upper and lower limits of the range of integers. You have seen the first concrete example of abstraction. Abstraction is a vast topic. We will cover some more basics about abstraction in this section.

Suppose a programmer writes the code for `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 and the second one is the user of the procedure. What pieces of information does the user of the `getRangeSum` procedure needs know in order to use it? Before we answer this question, let us consider a real-world example of designing and using a DVD player (**D**igital **V**ersatile **D**isc player). A DVD player is designed and developed by Electronic engineers. How do you use a DVD player? You do not open a DVD player to study all the details about its parts that are based on Electronics engineering theories before you use it. When you buy a DVD player, 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 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 the DVD disc, 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 about how a DVD player works internally. The manual also describes some conditions to operate the DVD player. For example, you must plug the power cord to a power outlet and switch on the power before you can use the DVD player.

A program is designed, developed and used in the same way as a DVD player. The user of the program shown in Listing 1-2 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 – interface to use it, and conditions that must be met before and after using it. In other words, we 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, specification for a program is also known as *documentation* or *comments*. The specification for a program 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 the users of the program.

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 greater 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;
}
```

Listing 1-3 shows the same `getRangeSum` procedure code with its specification. It uses Javadoc standards to write specification for a Java program that can be processed by Javadoc tool to generate HTML pages. In Java, specification for a program element is placed between `/**` and `*/`. The specification is meant for the users of the `getRangeSum` procedure. Javadoc tool (refer to *Appendix - B* for details) will generate the following specification (partially shown) for `getRangeSum` procedure.

getRangeSum
`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)

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 "*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, we 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 sub-program (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 1-4 depicts the user's view of `getRangeSum` procedure. Note that a user does not see or need not see the body of the procedure that has the details for computing the sum of integers between two integers. The details for computing the sum of integers between two integers are relevant only for the writer of the program and not its users.

Figure 1-4: User's view of the getRangeSum procedure as a black box using abstraction

What advantages did we achieve by applying the abstraction to define `getRangeSum` procedure? One of the most important advantages is *isolation*. It is isolated from other programs. If we 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`, we used 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 `getRangeSum` procedure executes `n` times where `n` is equal to (`upperLimit – lowerLimit + 1`). The number of instructions that need 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 a formula as follows.

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

If we 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. We 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 getRangeSum procedure with 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 1-3 and Listing 1-4. However, the users of `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 `getRangeSum` procedure as shown in Listing 1-4, 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 future without affecting other parts of the application.

We will consider two types abstraction in this section.

- Procedural abstraction
- Data abstraction

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

Object-oriented programming is based on data abstraction. We need to discuss about data type briefly, before we discuss about 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 pre-defined 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 `int` data type.
- A value of `int` data type is represented in 32-bit memory in 2's compliment form.

All three components of `int` data type are pre-defined by Java language. You cannot extend or re-define the definition of `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, we can associate integer `26` to the name `n1` using an assignment statement as:

```
n1 = 26;
```

You may ask a question at this stage, "Where in memory the value `26`, which is associated with the name `n1`, is stored?" We know from definition of `int` data type that `n1` will take 32-bit memory. However, we do not know, cannot know, and need not 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 is 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 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.

Object-oriented programming languages, e.g., Java, let you create new data types using an abstraction mechanism called *data abstraction*. The new data types are known as **A**bstract **D**ata **T**ypes (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 programmer create a new data type called 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 because 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 ADT does not mean absence of data representation. Data representation is always present in ADT. It only means hiding of the data representation from its users.

Java language has two constructs, class and interface, that let you define new ADTs. When you use a class to define a new ADT in Java, you need to be careful to hide the data representation so that 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 implementation of operations that operates on the data representation. An interface in Java lets you create 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`, we 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. We are using three terms, `type`, `class` and `interface`, interchangeably because they mean one and 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 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 the above snippet of code has been written only in terms of the operations provided by `Person` type. It has never referred (and could not refer) to the name and gender instance variables directly. Let us see how we can change the data representation of `Person` type without affecting the above snippet of code. Listing 1-5 shows the code for newer version for the `Person` class. Compare the code in Listing 1-1 and Listing 1-5. This time we have replaced the two instance variables (name and gender), which were the data representation for `Person` type in Listing 1-1 by a `String` array of two elements. Since operations (or methods) in a class operate on the data representation, we had to change the implementations for all four operations in `Person` type. The above client code was written in terms of the specifications of the four operations and not their implementation. Since we have not changed the specification of any of the operation, we do not need to change the above snippet of code that uses the `Person` class. It is still valid with the newer definition of `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. We have not shown the specification for the methods here, which would be Javadoc comments.

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

```
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];
     }
}
```

We 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. What new data types you create depends on the application domain. For example, for a banking system, `Person`, `Currency` and `Account` may be a good choice for new data types, whereas for an auto insurance application, Person, Vehicle and Claim may be good choice of new data types. What operations are included in a new data type depends 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 class.
- It lets you bundle one or more logically related classes in an entity called 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 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 about how an item is hidden. Encapsulation, abstraction and information hiding are three separate concepts. They are very closely related though. One concept facilitates workings of 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, we can state that the `Person` class is an encapsulation of data elements - `name` and `gender`, and methods – `getName()`, `setName()` and `getGender()`. The same `Person` class uses information hiding by hiding the data elements – `name` and `gender`, 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 code for the `Person2` class. The code in Listing 1-1 and Listing 1-6 are 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 exactly the same way as 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.

Listing 1-6: The definition of Person2 class in which data elements are not hidden by declaring them public

```
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;
     }
}
```

TIP
Encapsulation and information hiding are two distinct concepts of object-oriented programming. Existence of one does not imply the existence of other.

Inheritance

Inheritance is another important concept in object-oriented programming. It lets you use abstraction in a new way. We 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. 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 subtype is derived (or inherited) from supertype; supertype is a generalization of subtype; and subtype is a specialization of 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.

Inheritance allows you to use varying degrees of abstraction at different levels of hierarchy. In Figure 1-5, the `Person` class is at the top (highest level) of the inheritance hierarchy. `Employee` and `Customer` classes are at the second level of inheritance hierarchy. As we move up the inheritance level, we focus on more important pieces information. In other words, at higher level of

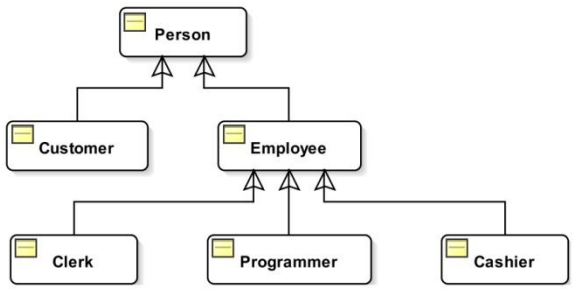
inheritance, we are concerned about the bigger picture; and at lower levels of inheritance, we are concerned about more and more details. There is another way to look at inheritance hierarchy from abstraction point of view. At `Person` level in Figure 1-5, we focus on the common characteristics of `Employee` and `Customer`, and we ignore the difference between them. At `Employee` level, we focus on common characteristics of `Clerk`, `Programmer` and `Cashier`, and we ignore the differences between them.

In inheritance hierarchy, a supertype and its subtype represent an "is-a" relationship. That is, an `Employee` is a `Person`; a `Programmer` is an `Employee`, etc. Since the lower level of inheritance means more pieces of information, a subtype always includes what its supertype has and maybe some more. This characteristic of inheritance leads to another feature in object-oriented programming, which is known as *principle of substitutivity*. It means that a supertype can always be substituted with its subtype. For example, we have considered only `name` and `gender` information for a person in our `Person` abstraction. If we inherit `Employee` from `Person`, `Employee` includes `name` and `gender` information, which it inherits from `Person`. `Employee` may include some more pieces of information such as employee id, hire date, salary, etc. If a `Person` is expected in a context, it implies that only `name` and `gender` information are relevant in that context. We can always replace a `Person` in this context with an `Employee`, a `Customer`, a `Clerk`, or a `Programmer` because being a subtype (direct or indirect) of the `Person` these abstractions guarantee that they have the ability to deal with at least `name` and `gender` information.

At programming level, inheritance provides a code reuse mechanism. The code written in supertype may be reused by its subtype. A subtype may extend the functionality of its supertype by adding more functionality or by redefining existing functionalities of its supertype.

Inheritance is a vast topic. This book devotes a complete chapter to inheritance. We will discuss how Java allows us to use inheritance mechanisms in *Chapter 9 - Inheritance and Reusability*

Figure 1-5: Inheritance hierarchy for the Person class



Polymorphism

The word "Polymorphism" has its root in two Greek words – "poly" (means many) and "morphos" (means form). In programming, polymorphism is the ability of an entity (e.g. variable, class, method, object, code, parameter, etc) to take on different meanings in different contexts. The entity that takes on different meanings is known as polymorphic entity. Various types of polymorphism exist. Each type of polymorphism has a name that usually indicates how that type of polymorphism is achieved in practice. The proper use of polymorphism results in generic and reusable codes. The purpose of polymorphism is writing reusable and maintainable code by writing codes in terms of a generic type that works for many types (or ideally all types).

Polymorphism can be categorized in the following two categories.

- Ad hoc Polymorphism
- Universal Polymorphism

If types for which a piece of code works is finite and all those types must be known when code is written, it is known as ad hoc polymorphism. Ad hoc polymorphism is also known as apparent polymorphism because it is not a polymorphism in a true sense. Some computer science purists do not consider ad hoc polymorphism as polymorphism at all. Ad hoc polymorphism is divided into two types – overloading polymorphism and coercion polymorphism.

If a piece of code is written in such a way that it works for infinite number of types (will also work for new types not known at the time the code is written), it is called universal polymorphism. In universal polymorphism, the same code works on many types, whereas in ad hoc polymorphism different implementations of code are provided for different types giving an apparent impression of polymorphism. Universal polymorphism is divided into two types – inclusion polymorphism and parametric polymorphism.

Overloading Polymorphism

Overloading is an ad hoc polymorphism. Overloading results when a method (called method in Java and function in other languages) or an operator has at least two definitions that work on different types. In such cases, the same method or operator name is used for different definitions of the method or the operator. That is, the same name exhibits many behaviors and hence the polymorphism. Such methods and operators are called overloaded methods and overloaded operators. Java lets you define overloaded methods. Java has some overloaded operators. Java does not let you overload an operator in your code. You cannot provide a new definition for an operator in Java.

Listing 1-7: An example of an overloaded method in Java

```
public class MathUtil {
     public static int max(int n1, int n2) {
          // Code to determine the maximum of two integers goes here
     }

     public static float max(double n1, double n2) {
          // Code to determine the maximum of two floating-point numbers
          // goes here
     }

     public static int max(int[] num) {
          // Code to determine the maximum an array of integers goes here
     }
}
```

Listing 1-7 shows code for a class named `MathUtil`. The `max()` method of `MathUtil` class is overloaded. It has three definitions and each of its definitions performs the same task of computing maximum, but on different types. The first definition computes maximum of two numbers of `int` data type; second one computes maximum of two floating-point numbers of `double` data type and third one computes maximum of an array of numbers of `int` data type. The following snippet of code makes use of all three definitions of the overloaded `max()` method.

```
int max1 = MathUtil.max(10, 23);       // Uses max(int, int)
```

```
int max2 = MathUtil.max(10.34, 2.89); // Uses use max(float, float)
int max3 = MathUtil.max(new int[]{1, 89, 8, 3}); // Uses max(int[])
```

Note that method overloading gives you only sharing of the method name. It does not result in the sharing of definitions. In Listing 1-7, the method name `max` is shared by all three methods, but they all have their own definition of computing maximum of different types. In method overloading, the definitions of methods do not have to be related at all. They may perform entirely different things and share the same name.

The following code snippet shows an example of operator overloading in Java. The operator is `+`. In the following three statements, it performs three different things.

```
int n1 = 10 + 20;              // Adds two integers
double n2 = 10.20 + 2.18;      // Adds two floating-point numbers
String str = "Hi " + "there"; // Concatenates two strings
```

In the first statement, the + operator performs addition on two integers - `10` and `20` and returns 30. In the second statement, it performs addition on two floating-point numbers – `10`.20 and `2.18` and returns `12.38`. In the third statement, it performs concatenation of two strings and returns `"Hi there"`.

In overloading, the types of actual method's parameters (types of operands in case of operators) are used to determine which definition of the code to use. Method overloading provides only the reuse of the method name. You can remove method overloading by simply supplying a unique name to all versions of an overloaded method. For example, you could rename three versions of `max()` method as `max2Int()`, `max2Double()` and `maxNInt()`. Note that all versions of an overloaded method or operator do not have to perform related or similar tasks. In Java, the only requirement to overload a method name is that all versions of the method must differ in number and/or type of their formal parameters.

Coercion Polymorphism

Coercion is an ad hoc polymorphism. Coercion occurs when a type is implicitly converted (coerced) to another type automatically even if it was not intended explicitly. Consider the following statements in Java.

```
int num = 707;
double d1 = (double)num;  // Explicit conversion of int to double
double d2 = num;  // Implicit conversion of int to double (coercion)
```

The variable `num` has been declared to be of `int` data type, and it has been assigned a value of `707`. The second statement uses cast, `(double)`, to convert the `int` value stored in `num` to `double`, and assigns the converted value to `d1`. This is the case of explicit conversion from `int` to `double`. In this case, the programmer makes his intention explicit by using the cast. The third statement has exactly the same effect as the second one. However, it relies on implicit conversion (called widening conversion in Java) provided by Java language that converts an `int` to `double` automatically when needed. The third statement is an example of coercion. A programming language (including Java) provides different coercions in different contexts – assignment (shown above), method parameters, etc.

Let us consider the following snippet of code that shows definition of a `square()` method, which accepts a parameter of `double` data type.

```
double square(double num) {
     return num * num;
}
```

The `square()` method can be called with actual parameter of `double` data type as:

```
double d1 = 20.23;
double result = square(d1);
```

The same `square()` method may also be called with actual parameter of `int` data type as:

```
int k = 20;
double result = square(k);
```

We have just seen that the `square()` method works on `double` data type as well as `int` data type although we have defined it only once in terms of a formal parameter of `double` data type. This is exactly what polymorphism means. In this case, `square()` method is called polymorphic method with respect to `double` and `int` data type. In this case, `square()` method is exhibiting polymorphic behavior even though the programmer who wrote code did not intend it. The `square()` method is polymorphic because of implicit type conversion (coercion from `int` to `double`) provided by Java language. Here is a more formal definition of a polymorphic method.

> *Suppose* m *is a method that declares a formal parameter of type* T. If S *is a type that can be implicitly converted to* T, the method m *is said to be polymorphic with respect to* S *and* T.

Inclusion Polymorphism

Inclusion is a universal polymorphism. It is also known as subtype (or subclass) polymorphism because it is achieved using subtyping or subclassing. This is the most common type of polymorphism supported by object-oriented programming languages. Java supports it. Inclusion polymorphism occurs when a piece of code that is written using a type works for all its subtypes. This type of polymorphism is possible based on the subtyping rule that a value that belongs to a subtype also belongs to the supertype. Suppose `T` is a type and `S1`, `S2`, `S3...` are subtypes of `T`. A value that belongs to `S1`, `S2`, `S3...` also belongs to `T`. This subtyping rule makes us write code as follows.

```
T t;
S1 s1;
S2 s2;
...
t = s1; // A value of type s1 can be assigned to variable of type T
t = s2; // A value of type s2 can be assigned to variable of type T
```

Java supports inclusion polymorphism using inheritance, which is a subclassing mechanism. You can define a method in Java using a formal parameter of a type, e.g., `Person`, and that method can be called on all its subtypes, e.g., `Employee`, `Student`, `Customer`, etc. Suppose we have a method `processDetails()` as follows.

```
void processDetails(Person p) {
    /* Write code using the formal parameter p, which is
       of type Person. The same code will work if an object
       of any of the subclass of Person is passed to this method
    */
```

```
}
```

The `processDetails()` method declares a formal parameter of Person type. You can define any number of classes that are subclasses of the `Person` class. The `processDetails()` method will work for all subclasses of the `Person` class. Assume that `Employee` and `Customer` are subclasses of the `Person` class. We can write code like:

```
Person p1 = create a Person object;
Employee e1 = create an Employee object;
Customer c1 = create a Customer object;
processDetails(p1); // Use Person type
processDetails(e1); // Use Employee type, which is a subclass of Person
processDetails(c1); // Use Customer type, which is a subclass of Person
```

The effect of the subtyping rule is that the supertype includes (hence the name inclusion) all values that belong to its subtypes. A piece of code is called universally polymorphic only if it works on infinite number of types. In the case of inclusion polymorphism, the number of types for which the code works is constrained but infinite. The constraint is that all types must be the subtype of the type in whose term the code is written. If there is no restriction on how many subtypes a type can have, the number of subtypes is infinite (at least in theory). Note that inclusion polymorphism not only lets you write reusable code, it also lets you write extensible and flexible code. The `processDetails()` method works on all subclasses of the `Person` class. It will keep working for all subclasses of the `Person` class, which will be defined in future, without any modifications. Java uses other mechanisms, like method overriding and dynamic dispatch (also called late binding), along with subclassing rules to make the inclusion polymorphism more effective and useful.

Parametric Polymorphism

Parametric is a universal polymorphism. It is also called “true” polymorphism, because it lets you write true generic code that works for any types (related or unrelated). Sometimes, it is also referred to as generics. In parametric polymorphism, a piece of code is written in such a way that it works on any type. Contrast parametric polymorphism with inclusion polymorphism. In inclusion polymorphism, code is written for one type and it works for all of its subtypes. It means all types for which the code works in inclusion polymorphism are related by supertype-subtype relationship. However, in parametric polymorphism, the same code works for all types, which are not necessarily related. Parametric polymorphism is achieved by using a type variable while writing the code rather than using any specific type. The type variable assumes a type for which the code needs to be executed. Java supports parametric polymorphism since Java 5 through generics. Java supports polymorphic entity (e.g. parameterized classes) as well as polymorphic method (parameterized methods) that use parametric polymorphism.

All collection classes in Java 5 have been retrofitted to use generics (parametric polymorphism is achieved in Java using generics). You can write code using generics as shown below. It uses a List object as a list of String type and Integer type. Using generics, you can treat a List object as a list of any type in Java. Note the use of <XXX> (angle brackets) in code to specify the type for which you want to instantiate the List object.

```
// Use List for String type
List<String> sList = new ArrayList<String>();
sList.add("string 1");
sList.add("string 2");
String s2 = sList.get(1);

// Use List for Integer type
```

```
List<Integer> iList = new ArrayList<Integer>();
iList.add(10);
iList.add(20);
int i2 = iList.get(1)
```

Chapter 2. Arrays

What is an Array?

An array is a fixed-length data structure that is used to hold more than one value of the same type. Let us consider an example, which will explain why we need an array. Suppose you have been asked to declare variables to hold employee ids of three employees. It has been stated that the employee ids will be integers. Your variable declarations to hold three integer values will look like:

```
int empId1, empId2, empId3;
```

What do you do if the number of employees increases to five? You may modify your variable declarations to:

```
int empId1, empId2, empId3, empId4, empId5;
```

What do you do if the number of employees increases to one thousand? Definitely, you would not want to declare one thousand `int` variables like `empId1`, `empId2...empId1000`. Even if you go for 1000 variables declarations, the resulting code would be unmanageable and clumsy. Arrays come to your rescue in such situations. Using an array, you can declare a variable of a type, which can hold as many values of that type as you want. In fact, Java has a restriction on the number of values an array can hold. An array can hold a maximum of 2147483647 values, which is the maximum value of the `int` data type. What makes a variable an array? Placing `[]` (empty brackets) after the data type or after the variable name in a variable declaration makes the variable an array. For example,

```
int empId;
```

is a simple variable declaration. Here, `int` is the data type and `empId` is the variable name. This declaration means that the `empId` variable can hold one integer value. If we place `[]` after the data type in the above declaration:

```
int[] empId;
```

then `empId` is an array variable. The above declaration is read as – "*empId is an array of int*". You can also make the `empId` variable an array as:

```
int empId[];
```

Both of the above declarations of `empId` as an array of `int` are valid. This book uses the first convention to declare an array (e.g. `int[] empId;`). We started our discussion with an example of variable declaration to hold three employee ids. Until now, we have prepared the ground to hold more than one value in one variable. That is, our `empId` variable declared as an array of `int` is capable of holding more than one `int` value. How many values can our `empId` array variable hold? The answer is we do not know yet. You cannot specify the number of values an array can hold at the time you declare the array. The subsequent sections explain how to specify the number

of values an array can hold. You can declare an array to hold multiple values of a data type – primitive or reference. More examples of arrays declarations are shown below.

```
//salary can hold multiple float values
float[] salary;

//name can hold multiple references to String objects
String[] name;

//emp can hold multiple references to Employee objects
Employee[] emp;
```

An Array is an Object

All arrays in Java are objects. Every object belongs to a class, so does every array object. You can create an array object using the `new` operator. We have used the `new` operator with a constructor to create an object of a class. The name of a constructor is the same as the name of the class. What is the name of the class of an array object? The answer to this question is not so obvious. We will answer this question later in this chapter. For now, we will concentrate on how to create an array object of a particular type. The array object creation expression starts with the `new` operator, followed by the data type of the values you want to store in the array, followed by an integer enclosed in `[]` (brackets), which is the number of values you want to store in the array. The general syntax for array creation expression is shown below.

```
new ArrayDataType[ArrayLength];  // Creates an array object of type
                                 // ArrayDataType of ArrayLength length
```

Continuing with our discussion, we can declare an array to store five employee ids as follows.

```
new int[5];
```

In the above expression, 5 is the *length* of the array (also called the *dimension* of the array). The word *dimension* is also used in another context. You can have an array of dimension one, two, three, or more. An array with more than one dimension is called a multi-dimensional array. We will cover the multi-dimensional array later in this chapter. In this book, we will refer to 5 in the above expression as the length and not as the dimension of the array.

Note that the above expression creates an array object in memory, which allocates memory to store 5 integers. The `new` operator returns the reference of the new object in memory. If we want to use this object later in our code, we must store that reference in an object reference variable. The reference variable type must match the type of object reference returned by the `new` operator. In the above case, the `new` operator will return an object reference of `int` array type. We have already seen how to declare a reference variable of `int` array type. It is declared as:

```
int[] empId;
```

To store the array object reference in `empId`, we can write:

```
empId = new int[5];
```

We can also combine the declaration of an array and its creation in one statement as:

```
int[] empId = new int[5];
```

How would you create an array to store 252 employee ids? You can do this as:

```
int[] empId = new int[252];
```

You can also use an expression to specify the length of an array while creating it. For example,

```
int total = 23;
int[] array1 = new int[total];       // array1 has 23 elements
int[] array2 = new int[total * 3];   // array2 has 69 elements
```

Since all arrays in Java are objects, their references can be assigned to a reference variable of `Object` type. For example,

```
int[] empId = new int[5]; // Create an array object
Object obj = empId;       // Valid assignment
```

However, if you have the reference of an array in a reference variable of `Object` type, you need to cast it to the appropriate array type before you can assign it to an array reference variable. Remember that every array in Java is an object. However, not every object in Java is necessarily an array. If you want to assign an `int` array reference stored in an `obj` reference variable to a variable of `int` array type, in the above code, you will do it as:

```
// Assume that obj has a reference of an int array. If obj does not
// have a reference to an int array, the following assignment will
// generate runtime error, but will compile fine
int[] tempIds = (int[])obj;
```

Referring to Elements of an Array

Once you create an array object using the `new` operator, you can refer to each individual element of the array using an element's index enclosed in brackets. The index for the first element of an array is 0 (zero). The index for the second element of an array is 1 and so on. The index for the last element of an array is equal to the length of the array minus 1. If you have an array of length 5, the indexes you can use to refer to array elements would be 0, 1, 2, 3 and 4. Consider the following statement:

```
int[] empId = new int[5];
```

The length of the `empId` array is 5; its elements can be referred to as `empId[0]`, `empId[1]`, `empId[2]`, `empId[3]`, and `empId[4]`. It is a runtime error to refer to a non-existing element of an array. For example, using `empId[5]` in your code will throw an exception, because `empId` has a length of 5 and `empId[5]` refers to the 6th element, which is non-existent. You can assign values to elements of an array as follows.

```
empId[0] = 10; // Assign 10 to the 1st element of empId
empId[1] = 20; // Assign 20 to the 2nd element of empId
empId[2] = 30; // Assign 30 to the 3rd element of empId
empId[3] = 40; // Assign 40 to the 4th element of empId
empId[4] = 50; // Assign 50 to the 5th element of empId
```

Table 2-1 has array's elements index, elements in memory with values and the element's syntactic notation to refer to them in program after above statements are executed.

Table 2-1: Array elements in memory for the empId array

Element's Index →	0	1	2	3	4
Element's values →	10	20	30	40	50
Element's reference →	empId[0]	empId[1]	empId[2]	empId[3]	empId[4]

If you want to assign the value of the 3rd element of the `empId` array to an `int` variable `temp`, you can do so as:

```
int temp = empId[2];    // Assign the value stored in 3rd element of
                        // empId array to variable temp
```

Length of an Array

An array object has a `public final` instance variable named `length`. You can use the `length` instance variable to get the number of elements in the array as:

```
int[] empId = new int[5];    // Create an array of length 5
int len = empId.length ;     // 5 will be assigned to len
```

Note that `length` is the property of the array object you create. Unless you create the array object, you cannot use its `length` property. Following code fragment illustrates this situation.

```
int[] salary; // salary is just a reference variable, which can
              // refer to an array of int. At this point,
              // salary is not referring to any array object

int len = salary.length;     // Runtime error. salary is not referring
                             // to any array object yet

salary = new int[1000];      // Create an int array of length 1000
                             // and assign its reference to salary

int len2 = salary.length;    // Correct. len2 has value 1000
```

Typically, when you work with arrays, you also work with loops. If you want to do any processing with all elements of an array, you execute a loop starting from index 0 (zero) to length minus 1. For example, to assign the values 10, 20, 30, 40, and 50 to the elements of the `empId` array of length 5, you would execute a for-loop as:

```
for (int i = 0 ; i < empId.length; i++) {
     empId[i] = (i + 1) * 10;
}
```

It is important to note that while executing the loop, the loop condition must check for array index/subscript for being less than the length of array as in "`i < empId.length`", because the array index starts with 0 (zero) and not 1. Another common mistake made by programmers, while

processing an array using a for-loop, is to start the loop with a loop counter of 1 as opposed to 0 (zero). What will happen if we change the initialization part of the for-loop in the above code from `int i = 0` to `int i = 1`? It would not give us any errors. However, our first element `empId[0]` would not be processed and would not be assigned the value of 10 as intended.

You cannot change the length of an array after it is created. You may be tempted to modify its length property as:

```
int[] roll = new int[5]; // Create an array of 5 elements

roll.length = 10;  // Compiler error. Length property of an array is
                   // declared final. You cannot modify it.
```

You can have a zero-length array. Such an array is called an empty array. You can create an array of `int` of zero length as:

```
int[] emptyArray = new int[0];    // Create an array of length zero
int len = emptyArray.length;      // Will assign zero to len
```

TIP
In Java, array indexes are zero based. That is, the first element of an array has an index of zero. Arrays are created dynamically at runtime and their length cannot be modified after it has been created. If you feel the need to alter the length of an array, you must create a new array and copy the elements from the old array to the new array. It is valid to create an array of length zero.

Initializing Elements of an Array

Recall from the chapter on *Classes and Objects* that unlike class member variables (instance and static variables), local variables are not initialized by default. That is, you cannot access a local variable in your code unless you have assigned a value to it. The same rule applies to blank final variables. The Java compiler uses *Rules of Definite Assignment* to make sure that all variables have been initialized before their values are used in a program.

Array elements are always initialized irrespective of the scope in which the array is created. Array elements of primitive data type are initialized to the default value for their data types. For example, the elements of arrays of numeric types are initialized to zero; the elements of `boolean` arrays are initialized to `false`; the elements of a reference type arrays are initialized to `null`. The following snippet of code illustrates the array initialization. Listing 2-1 illustrates the array initialization for an instance variable and some local variables.

```
// All three elements have value of zero. intArray[0], intArray[1]
// and intArray[2] are initialized to zero by default
int[] intArray = new int[3];

// bArray[0] and bArray[1] are initialized to false
boolean[] bArray = new boolean[2];

// Example of reference type array strArray[0] and strArray[1]
// are initialized to null
String[] strArray = new String[2]
```

```
// Another example of reference type array. All 100 elements of
// person array are initialized to null
Person[] person = new Person[100];
```

Listing 2-1: Array Initialization

```
// ArrayInit.java
package com.jdojo.chapter14;

public class ArrayInit {
     private boolean[] bArray = new boolean[3];  // An instance variable

     public ArrayInit() {
          // Display the initial value for elements of
          // instance variable bArray
          for (int i = 0; i < bArray.length; i++) {
               System.out.println("bArray[" + i + "]:" + bArray[i]);
          }
     }

     public static void main(String[] args) {
          int[] empId = new int[3];  // A local array variable
          System.out.println("int array initialization:");
          for (int i = 0; i < empId.length; i++) {
               System.out.println("empId[" + i + "]:" + empId[i]);
          }

          System.out.println("\nboolean array initialization:");
          new ArrayInit();  // Note that we are displaying initial
                            // value for bArray inside constructor

          String[] name = new String[3];   // A array local variable
          System.out.println("\nReference type array initialization:");

          for (int i = 0; i < name.length; i++) {
               System.out.println("name[" + i + "]:" + name[i]);
          }
     }
}
```

Output:

```
int array initialization:
empId[0]:0
empId[1]:0
empId[2]:0

boolean array initialization:
bArray[0]:false
bArray[1]:false
bArray[2]:false

Reference type array initialization:
name[0]:null
name[1]:null
name[2]:null
```

Be Careful with Reference Type Arrays

Array elements of a primitive type contain values of that primitive type, whereas array elements of a reference type contain the reference of objects. For example, if we have an array `empId` as:

```
int[] empId = new int[5];
```

Then, `empId[0]`,`empId[1]...empId[4]` contain an `int` value. However, if we have an array `name` as:

```
String[] name = new String[5];
```

Then, `name[0], name[1]...name[4]` may contain reference of a `String` object. Note that the `String` objects, the elements of the `name` array, have not been created yet. As discussed in the previous section, all elements of the `name` array are referring to `null` at this stage. You need to create the `String` objects and assign those object references to the elements of the array one by one as:

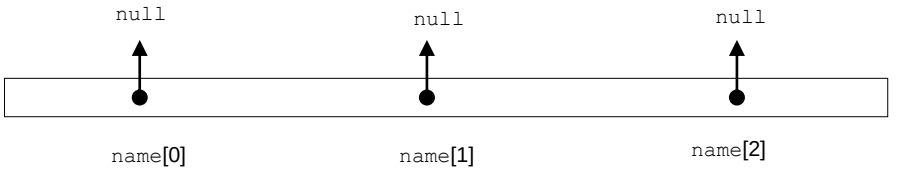
```
name[0] = new String("name1");
name[1] = new String("name2");
name[2] = new String("name3");
name[3] = new String("name4");
name[4] = new String("name5");
```

It is a common mistake to refer to the elements of an array of reference type just after creating the array and before assigning a valid object reference to each element. The following code illustrates this common mistake.

```
String[] name = new String[5];  // Create an array of string

/* Error! Code is trying to get the length of first string stored in
   name array. It will generate runtime error because name[0] is not
   referring to any string object yet. name[0] is null at this point
*/
int len = name[0].length();

// Assign a valid string object to all elements of name array
for (int i = 0; i < name.length; i++){
     name[i] = "name" + ( i + 1 );
}

// Now you can get the length of the first element as
int len2 = name[0].length();     // Correct. len2 has value 5
```

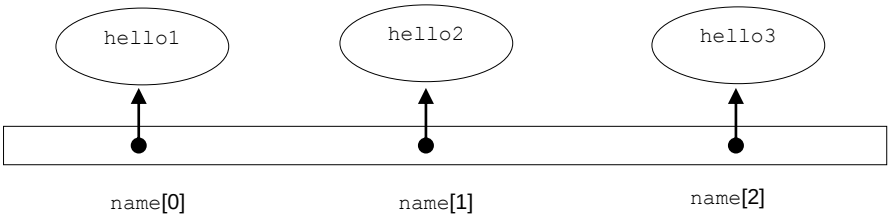
The concept of initialization of the `String` reference type array has been depicted in Figure 2-1. This concept applies to all reference types.

Figure 2-1: Reference type array initialization



Memory state after the following statement is executed.

```
String[] name = new String[3];
```



Memory state after the following statement is executed.

```
For (int i = 0; i < name.length; i++){
        name[i] = "hello" + ( i + 1 );
}
```

Explicit Array Initialization

You may want to initialize array elements with values other than the default values. You can initialize elements of an array explicitly either when you declare the array or when you create the array object using the `new` operator. The initial values for elements are separated by a comma and enclosed in braces (`{}`). For example,

```
int[] empId = {1, 2, 3, 4, 5};
```

The above statement creates an array of `int` of length 5, and initializes its elements to `1`, `2`, `3`, `4`, and `5`. Note that you do not specify the length of an array when you specify the array initialization list at the time of the array declaration. The length of the array is the same as the number of values specified in the array initialization list. Here, length of `empId` array will be 5, because we passed 5 values in the initialization list {`1`, `2`, `3`, `4`, `5`}.

A comma may follow the last value in initialization list as:

```
int[] empId = {1, 2, 3, 4, 5,}; // A comma after last value 5 is valid
```

Alternatively, you can initialize the elements of an array as:

```
int[] empId = new int[]{1, 2, 3, 4, 5};
```

Note that you cannot specify the length of an array if you specify the array initialization list. The length of the array is the same as the number of values specified in the initialization list.

It is valid to create an empty array by using an empty initialization list as:

```
int[] emptyNumList = { };
```

For a reference type array, you can specify the list of objects in the initialization list. The following code fragment illustrates array initialization for `String` and `Account` types. Assume that `Account` class exists and it has a constructor, which takes account number as an argument.

```
// Create name array with two elements with text "Sara" and "Truman"
String[] name = {new String("Sara"), new String("Truman")};

// Create ac array with two object of Account type
Account[] ac = new Account[]{new Account(1), new Account(2)};
```

TIP
You can initialize the array elements at the time of declaration or at the time of creation using an array initialization list. When you use an initialization list to initialize the elements of an array, you cannot specify the length of an array. The length of the array is set to the number of values in the array initialization list.

Limitations of Using Arrays

An array in Java cannot be expanded or shrunk after it is created. Suppose you have an array of 100 elements and at some point of time, you need to keep only 15 elements. You cannot get rid of the remaining (and redundant for you) 85 elements of the array. If you need 135 elements and you have created an array of only 100 elements, you cannot append 35 more elements to it. You can deal with the first limitation (memory cannot be freed for unused array elements), if you have enough memory available to your application. However, there is no way out if you need to add more elements to an existing array. The only solution you have is to create another array of the desired length, and copy the array elements from the original array to the new array. You can copy array elements from one array to another in two ways:

- Using a loop
- Using the `arraycopy() static` method of the `java.lang.System` class

If we have an `int` array of `originalLength` length and we want to modify its length to `newLength`, we can apply the first method of copying arrays as shown in the following snippet of code.

```
int originalLength = 100;
int newLength = 15;
int[] ids = new int[originalLength];

// Do some processing here...

// Create a temporary array of new length
int[] tempIds = new int[newLength];
```

```
// While copying array elements we have to check if the new length
// is less than or greater than original length
int elementsToCopy = 0;

if (originalLength > newLength) {
     elementsToCopy = newLength;
}
else {
     elementsToCopy = originalLength;
}

// Copy the elements from original array to new array
for (int i = 0; i < elementsToCopy; i++){
     tempIds[i] = ids[i];
}

// Finally assign the reference of new array to ids
ids = tempIds;     // Now ids array has newLength length
                   // If newLength is greater than originalLength,
                   // new elements will have default values
```

Another way to copy elements of an array to another array is by using the `arraycopy()` method of the `System` class. The signature of the `arraycopy()` method is shown below.

```
public static void arraycopy(Object sourceArray,
                             int sourceStartPosition,
                             Object destinationArray,
                             int destinationStartPosition,
                             int lengthToBeCopied)
```

Here,
`sourceArray` is the reference to the source array,
`sourceStartPosition` is the starting index in source array from where the copying of elements will start,
`destinationArray` is the reference to the destination array,
`destinationStartPosition` is the start index in destination array from where new elements from source array will be copied,
`lengthToBeCopied` is the number of elements to be copied from source array to destination array.

We can replace the last for-loop, which copies array elements from ids to `tempIds`, in the above snippet of code with the code shown below:

```
// Now copy array elements using arraycopy() method
System.arraycopy (ids, 0, tempIds, 0, elementsToCopy);
```

You may observe that using the `arraycopy()` method offers more flexibility over using a for-loop. Refer to Java2 Platform online documentation at *http://www.oracle.com* for the complete documentation of the `arraycopy()` method. The objects of the two classes, `java.util.ArrayList` and `java.util.Vector`, can be used in place of an arrays, where the length of the array needs to be modified. You can think of the objects of these two classes as variable length arrays. The next section discusses these two classes in detail.

Listing 2-2 demonstrates how to copy an array using a for-loop and the `System.arraycopy()` method. An `Arrays` class is in the `java.util` package. It has many convenience methods for dealing with arrays, e.g., methods for converting elements of an array to a string format, sorting an array, etc. Listing 2-2 uses the `Arrays.toString() static` method to get the contents of an array in the string format. The `Arrays.toString()` method is overloaded and you can use it to get the contents of an array of any type in string format. In this example, we have used a for-loop and the S`ystem.arraycopy()` method to copy arrays. However, note that using the `arraycopy()` method is much more powerful than using a for-loop. For example, the `arraycopy()` method is designed to handle copying of the elements of an array from one region to another region in the same array. It takes care of any overlap in the source and the destination regions within the array. You can use the `arraycopy()` method to copy any type of array. Our for-loop implementation to copy an array is a trivial example and it can be used only to copy `int` arrays.

Listing 2-2: Copying an array using a for-loop and the System.arraycopy() method

```
// ArrayCopyTest.java
package com.jdojo.chapter14;

import java.util.Arrays;

public class ArrayCopyTest {
     public static void main(String[] args) {
          // Have an array with 5 elements
          int[] data = {1, 2, 3, 4, 5 };

          // Expand the data array to 7 elements
          int[] eData = expandArray(data, 7);

          // Truncate the data array to 3 elements
          int[] tData = expandArray(data, 3);

          System.out.println("Using for-loop...");
          System.out.println("Original Array: " +
                             Arrays.toString(data));
          System.out.println("Expanded Array: " +
                          Arrays.toString(eData));
          System.out.println("Truncated Array: " +
                          Arrays.toString(tData));

          // Copy data array to new arrays
          eData = new int[9];
          tData = new int[2];
          System.arraycopy(data, 0, eData, 0, 5);
          System.arraycopy(data, 0, tData, 0, 2);

          System.out.println("\nUsing System.arraycopy() method...");
          System.out.println("Original Array: " +
                             Arrays.toString(data));
          System.out.println("Expanded Array: " +
                             Arrays.toString(eData));
          System.out.println("Truncated Array: " +
                             Arrays.toString(tData));
     }

     // Uses a for-loop to copy an array
```

```
    public static int[] expandArray(int[] oldArray, int newLength) {
        int originalLength = oldArray.length;
        int[] newArray = new int[newLength];
        int elementsToCopy = 0;

        if (originalLength > newLength) {
            elementsToCopy = newLength;
        }
        else {
            elementsToCopy = originalLength;
        }

        for (int i = 0; i < elementsToCopy; i++) {
            newArray[i] = oldArray[i];
        }
        return newArray;
    }
}
```

```
Output:

Using for-loop...
Original Array: [1, 2, 3, 4, 5]
Expanded Array: [1, 2, 3, 4, 5, 0, 0]
Truncated Array: [1, 2, 3]

Using System.arraycopy() method...
Original Array: [1, 2, 3, 4, 5]
Expanded Array: [1, 2, 3, 4, 5, 0, 0, 0, 0]
Truncated Array: [1, 2]
```

ArrayList and Vector

`ArrayList` and `Vector` classes work the same way (roughly speaking). The only major difference between them is that the methods in the `Vector` class are synchronized, whereas methods in `ArrayList` are not synchronized. If your object list is accessed and modified by multiple threads simultaneously, you should use the `Vector` class, which will be slower but thread safe. Otherwise, you should use the `ArrayList` class. The big difference between arrays and `ArrayList`/`Vector` classes is that these two classes work with only objects and not with primitive data types. For example, if you need to store an `int` in an `ArrayList`/`Vector`, you must wrap it in an `Integer` object, before you could store it. All methods for these two classes accepts arguments of `Object` type whenever an element is expected in the argument and return an object of `Object` type whenever an element is returned. Therefore, most of the time, you will have to cast the element returned by methods of these classes to appropriate type.

TIP
Java 5 introduced a feature called *Autoboxing*, which relieves you of the pain of wrapping and unwrapping primitive values to objects and vice-versa when you work with collections, e.g., `ArrayList` or `Vector`. Please refer to the chapter on *AutoBoxing* for more details.

The following code fragment illustrates the use of the `ArrayList` class. Note that you need to import `java.util.ArrayList` and `java.util.Vector` classes in you programs to use the simple names `ArrayList` and `Vector` respectively.

```
// Create an ArrayList object
ArrayList ids = new ArrayList();

// Get the size of array list
int total = ids.size(); // total will be zero at this point

// Print the details of array list
System.out.println("Array List size is " + total);
System.out.println("Array List elements are " + ids);

// Add three ids 10, 20, 30 in array list. Note that we cannot add
// int values directly to array list. We must wrap them in
// primitive wrapper class Integer
ids.add(new Integer(10));
ids.add(new Integer(20));
ids.add(new Integer(30));

// Get the size of the array list
total = ids.size();     // total will be 3

// Print the details of array list
System.out.println("Array List size is " + total);
System.out.println("Array List elements are " + ids);

// Clear all elements from array list
ids.clear();

// Get the size of the array list
total = ids.size();     // total will be 0

// Print the details of array list
System.out.println("Array List size is " + total);
System.out.println("Array List elements are " + ids);
```

```
Output:

Array List size is 0
Array List elements are []
Array List size is 3
Array List elements are [10, 20, 30]
Array List size is 0
Array List elements are []
```

You can make one important observation from the above output. You can print the list of all elements in an `ArrayList` just by passing its reference to the `System.out.println()` method. The `toString()` method of the `ArrayList` class returns a string that is a comma separated string representation of its elements enclosed in brackets (`[]`).

TIP
You can get the comma-separated list of `ArrayList` elements enclosed in brackets (`[]`) by using its `toString()` method. However, the `toString()` method of an array does not return the list of elements contained in the array.

Like arrays, `ArrayList` and `Vector` use zero based indexes. That is, the first element of `ArrayList` and `Vector` will have an index of zero. You can get the element stored at any index by using the `get()` method as:

```
// Get the element at the index 1, that is, the second element
Integer secondId = (Integer)ids.get(1);   // Must cast to Integer
                                          // because the return type of
                                          // the get() method is Object

// Get the integer value
int secondIntValue = secondId.intValue();

// Add three objects to the arraylist
ids.add(new Integer(10));
ids.add(new Integer(20));
ids.add(new Integer(30));
```

You can check if an object is one of the elements in an array list by using its `contains()` method as:

```
Integer id20 = new Integer(20);
Integer id50 = new Integer(50);

// Check if the array list contains id20 and id50
boolean found20 = ids.contains(id20);     // found20 will be true
boolean found50 = ids.contains(id50);     // found50 will be false
```

You can iterate through the elements of an `ArrayList` in one of the two ways – using a loop or using an iterator. In this chapter, we will discuss how to iterate through elements of an `ArrayList` using a for-loop. Please refer to the chapter on *Collections* to learn how to iterate through elements of an `ArrayList` (or any type of collection, e.g. a `Set`) using an iterator. The following snippet of code shows how to use a for-loop to iterate through the elements of an `ArrayList`.

```
// Get the size of the ArrayList
int total = ids.size();

Integer temp = null;

// Iterate through all elements
for (int i = 0; i < total; i++) {
     temp = (Integer)ids.get(i);  // Get element at index I
     // Do some processing …
}
```

Listing 2-3 illustrates the use of a for-loop to iterate through elements of an `ArrayList`. It also shows you how to remove an element from an `ArrayList` using its `remove()` method.

Listing 2-3: Iterating through elements of an ArrayList

```
// NameIterator.java
package com.jdojo.chapter14;

import java.util.ArrayList;

public class NameIterator {
     public static void main(String[] args) {
          ArrayList nameList = new ArrayList();

          //Add some names to it
          nameList.add("Christopher");
          nameList.add("Kathleen");
          nameList.add("Ann");

          // Get the count of names in the list
          int count = nameList.size();

          // Let us print the name list
          System.out.println("List of names...");
          for(int i = 0; i < count; i++) {
               String name = (String)nameList.get(i);
               System.out.println(name);
          }

          // Let us remove Kathleen from the list
          nameList.remove("Kathleen");

          // Get the count of names in the list again
          count = nameList.size();

          // Let us print the name list again
          System.out.println("\nAfter removing Kathleen...");
          for(int i = 0; i < count; i++) {
               String name = (String)nameList.get(i);
               System.out.println(name);
          }
     }
}
```

Output

```
List of names...
Christopher
Kathleen
Ann

After removing Kathleen...
Christopher
Ann
```

Passing an Array as a Parameter

You can pass an array as a parameter to a method or a constructor. The type of array you pass to the method must be assignment compatible to the formal parameter type. The syntax for array type parameter declaration for a method is the same as for the other data types. That is, parameter declaration should start with the array type, followed by one or more whitespaces and the argument name as:

```
modifiers returnType methodName(ArrayType argumentName, ...)
```

Some examples of method declarations with array arguments are:

```
// processSalary() has two parameters:
// 1. id is an array of int
// 2. salary is an array of double
public static void processSalary(int[] id, double[] salary) {
     // Code goes here...
}

// setAKA() has two parameters:
// 1. id is int (It is simply int type and not array of int)
// 2. aka is an array of String
public static void setAKA(int id, String[] aka) {
     // Code goes here...
}

// printStates() has one parameter:
// 1. stateNames is array of String
public static void printStates(String[] stateNames) {
     // Code goes here...
}
```

TIP
The parameter declaration for an array type in a method is specified the same way as we do an array type variable declaration. The parameter declaration of an array type must not specify the length of the array. The length of the array type parameter is determined at runtime using the actual parameter.

The following code fragment for a method mimics the `toString()` method of `ArrayList`. It accepts an `int` array and returns the comma-separated values enclosed in brackets (`[]`).

```
public static String arrayToString (int[] source){
     if (source == null) {
          return null;
     }

     // Use StringBuffer instead of String to improve
     // performance because we are doing string manipulations
     StringBuffer result = new StringBuffer("[");

     for (int i = 0; i < source.length; i++){
          if (i == source.length - 1) {
               result.append(source[i]);
```

```
            }
            else {
                result.append(source[i] + ",");
            }
        }

        result.append("]");
        return result.toString() ;
}
```

The above method may be called as:

```
int[] ids = {10, 15, 19};
String str = arrayToString(ids); // Pass ids int array to
                                 // arrayToString() method
```

Since an array is an object, the array reference is passed to the method. The method, which receives an array parameter, can modify the elements of the array. Listing 2-4 illustrates how a method can change the elements of its array parameter. This example also shows how to implement the `swap()` method to swap two integers using an array.

Listing 2-4: Passing an array as a method parameter

```
// Swap.java
package com.jdojo.chapter14;

public class Swap {
    public static void main(String[] args) {
        int[] num = {17, 80};

        System.out.println("Before swap");
        System.out.println("#1: " + num[0]);
        System.out.println("#2: " + num[1]);

        swap(num);

        System.out.println("\nAfter swap");
        System.out.println("#1: " + num[0]);
        System.out.println("#2: " + num[1]);
    }

    // swap method accepts an int array as argument and swaps the
    // values if array contains two values
    public static void swap (int[] source) {
        if (source != null && source.length == 2) {
            // Swap the first and the second elements
            int temp = source[0];
            source[0] = source[1] ;
            source[1] = temp ;
        }
    }
}
```

Output

```
Before swap
#1: 17
#2: 80

After swap
#1: 80
#2: 17
```

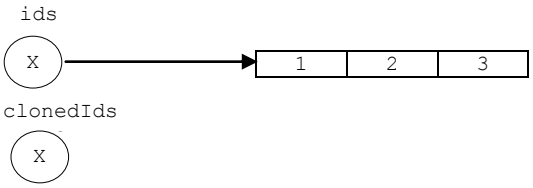
Recall that we were not able to implement a method for swapping two integers using method's parameters of primitive types. It was so, because the actual parameters' values are copied to the formal parameters for primitive data type parameters. Here, we were able to swap two integers inside the `swap()` method, because we used an array as the parameter. The array's reference is passed to the method and not the copy of the elements of the array.

There is a risk involved when an array is passed to a method. The method that receives an array as its parameter may modify the array elements, which may not be desired in some cases. In such a case, you should pass a copy of the array to the method and not the original array. The method may modify its array argument, which is a copy of original array, without affecting your original array. You can make a quick copy of your array using array's `clone()` method. The phrase *"quick copy"* warrants special attention. For primitive arrays, the cloned array will have a true copy of the original array. A new array of the same length is created and the value of each element in the original array is copied to the corresponding element of the cloned array. However, for reference type arrays, the reference of the object (not the object) stored in each element of the original array is copied to the corresponding element of the cloned array. This is known as a *shallow copy*, whereas the former type, where the object (or the value) is copied, is known as a *deep copy*. In case of a shallow copy, elements of both arrays, the original and the cloned, refer to the same object in memory. You can modify the objects using their reference stored in the original array as well as using the reference stored in the cloned array. In this case, even if you pass a copy of the original array to a method, your original array can be modified inside that method. The solution to this problem is to make a deep copy of your original array and pass the copy to the method. The following snippet of code illustrates the cloning of an `int` array and a `String` array. Note that the return type of the `clone()` method is `Object` and you need to cast the returned value to an appropriate array type.

```
// Create an array of 3 integers - 1,2,3
int[] ids = {1, 2, 3};

// Declare an array of int named clonedIds
int[] clonedIds;

// clonedIds array has the same values as ids array i.e. 1,2,3
clonedIds = (int[])ids.clone()

// Create an array of 3 strings
String[] names  = {"Lisa", "Pat", "Kathy"};

// Declare an array of String named clonedNames
String[] clonedNames ;

// clonedNames array has the reference of the same three strings as
// the names array has
clonedNames = (String[])names.clone();
```

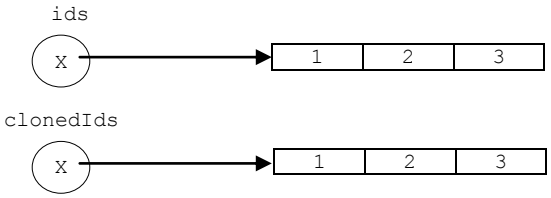
The cloning processes for primitive array (i.e. `ids`) and reference array (e.g. `names`) have been depicted in Figure 2-2.

Figure 2-2: Difference in primitive arrays and reference arrays cloning



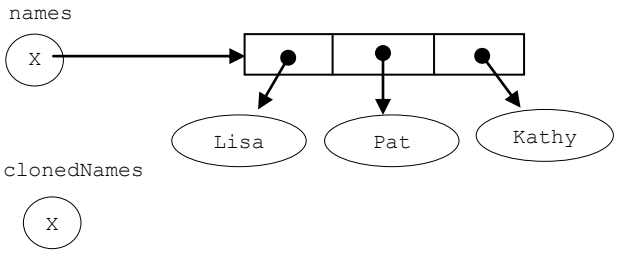
Memory state after the following statements are executed.

```
int[] ids = {1,2,3};
int[] clonedIds;
```



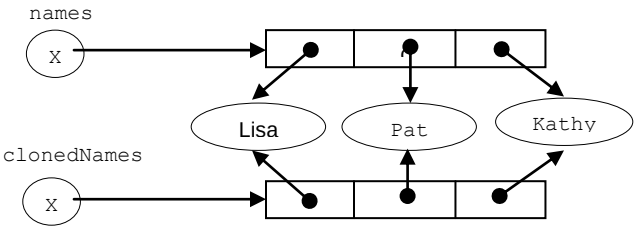
Memory state after the following statement is executed.

```
clonedIds = (int[])ids.clone();
```



Memory state after the following statements are executed.

```
String names = {"Lisa", "Pat", "Kathy"};
String[] clonedNames;
```




```
Memory state after the following statement is executed.
clonedNames = (String[])names.clone();
```

You may observe that when the `names` array is cloned, the `clonedNames` array elements also refer to the same `String` objects in memory. When we mention a method of modifying an array parameter passed to it, we may mean one or all of the following three things.

- Array Parameter Reference
- Elements of the Array Parameter
- The Object Referred by the Array Parameter Elements

In subsequent discussions, assume that we have a `getElements()` method that returns a `String` containing the elements of the passed array enclosed in `[]` and separated by a comma.

Array Parameter Reference

Since an array is an object in Java, a copy of its reference is passed to a method. If the method changes the array parameter, the actual parameter is not affected. For example, consider a method `tryArrayChange()` as shown below.

```
void tryArrayChange(int[] num) {
     System.out.println("Inside method-1:" + getElements(num));

     // Create and store a new int array in num
     num = new int[] {10, 20};

     System.out.println("Inside method-2:" + getElements(num));
}
```

The following snippet of code calls the method `tryArrayChange()`:

```
int[] origNum = {101, 307, 78};
System.out.println("Before method call:" + getElements(origNum));
tryArrayChange(origNum);
System.out.println("After method call:" + getElements(origNum))
```

```
Output:

Before method call:[101,307,78]
Inside method-1:[101, 307, 78]
Inside method-2:[10, 20]
After method call:[101,307,78]
```

You may observe that we assign a new array reference to the num method's parameter inside the method. However, it did not affect the actual parameter `origNum`.

If you do not want your method to change the array reference inside the method body, you must declare the method parameter as `final` as shown below.

```
void tryArrayChange(final int[] num) {
     // Error. num is final and cannot be changed
     num = new int[]{10, 20};
}
```

Elements of the Array Parameter

The values stored in the elements of an array parameter can always be changed inside a method. Consider the following snippet of code, which defines two methods.

```
void tryElementChange(int[] num) {
     // If array has at least one element, store 1116
     // in its first element
     if (num != null && num.length > 0) {
          num[0] = 1116;
     }
}

void tryElementChange(String[] names) {
     // If array has at least one element, store "Twinkle"
     // in its first element
     if (names != null && names.length > 0) {
          names[0] = "Twinkle";
     }
}
```

We use the following snippet of code to test these two methods.

```
Int[] origNum = { 10, 89 , 7};
String[] origNames  = {"Mike", "John"};

System.out.println("Before method call, origNum:" +
                    getElements(origNum));

System.out.println("Before method call, origNames:" +
                    getElements(origNames));

tryElementChange(origNum);
tryElementChange(origNames);

System.out.println("After method call, origNum:" +
                    getElements(origNum));

System.out.println("After method call, origNames:" +
                    getElements(origNames));
```

```
Output:

Before method call, origNum:[10, 89, 7]
Before method call, origNames:[Mike, John]
After method call, origNum:[1116, 89, 7]
After method call, origNames:[Twinkle, John]
```

You may observe that the arrays' first element has changed after the method calls. You can change the elements of an array parameter inside a method, even if the array parameter is declared `final`.

The Object Referred by the Array Parameter Elements

This section applies to array parameters of only the reference type. If the array's reference type is mutable, you can change the state of the object stored in the array elements. Note that in case 2 discussed above, we discussed replacing the reference stored in the array element by a new object

reference. This section discusses changing the state of the object referred to by the elements of the array. Consider a class `Item` as follows.

```
public class Item {
     private double price;
     private String name;

     public Item (String name, double initialPrice) {
          this.name = name;
          this.price = initialPrice;
     }

     public double getPrice() {
          return this.price;
     }

     public void setPrice(double newPrice ) {
          this.price = newPrice;
     }

     public String toString() {
          return "[" + this.name + ", " + this.price + "]";
     }
}
```

Let us consider the following snippet of code that defines a method, which changes the object state.

```
void tryStateChange(Item[] allItems) {
     if (allItems != null && allItems.length > 0 ) {
          // Change the price of first item to 10.38
          allItems[0].setPrice(10.38);
     }
}
```

The following piece of code is used to test the above method.

```
Item[] myItems = {new Item("Pen",25.11),new Item("Pencil",0.10)};

System.out.println("Before method call #1:" + myItems[0]);
System.out.println("Before method call #2:" + myItems[1]);

tryStateChange(myItems);

System.out.println("After method call #1:" + myItems[0]);
System.out.println("After method call #2:" + myItems[1]);
```

```
Output:

Before method call #1:[Pen, 25.11]
Before method call #2:[Pencil, 0.1]
After method call #1:[Pen, 10.38]
After method call #2:[Pencil, 0.1]
```

Note that the price of the first item has been changed inside the method.

TIP
The `clone()` method can be used to make a clone of an array. For a reference array, the `clone()` method performs a shallow copy. An array should be passed to a method and returned from a method with caution. If a method may modify its array parameter and you do not want your actual array parameter to get affected by that method call, you must pass a deep copy of your array to that method.

If you store the state of an object in an array instance variable, you should think carefully before returning the reference of that array from any method. The caller of that method will get the handle of the array instance variable and will be able to change the state of the objects of that class outside the class. This situation is illustrated in following example.

```
public class MagicNumber {
     // Magic numbers are not supposed to be changed
     // It can be look up though
     private int[] magicNumbers = {5, 11, 21, 51, 101};

     // Other codes go here...

     public int[] getMagicNumbers ( ) {
          /* Never do the following. If you do this, caller of
             this method will be able to change the magic numbers
          */
          //return this.magicNumbers;

         /* Do the following instead. In case of reference array,
            make a deep copy, and return that copy. For
            primitive array you can use clone() method
         */
          return (int[])magicNumbers.clone();
     }
}
```

You can also create an array and pass it to a method without storing the array reference in a variable. Suppose there is a method named `setNumbers(int[] nums)`, which accepts an `int` array as a parameter. You can call this method as shown below.

```
setNumbers(new int[]{10, 20, 30});
```

Command-Line Arguments

A Java application can be launched from a command prompt, e.g., a DOS prompt in windows and a shell prompt in UNIX. It can also be launched from within a Java development environment tool, e.g., NetBeans. A java application is run at the command line as:

```
java <<optionslists>> <<classname>>
```

`<<optionslists>>` is optional. You can also pass command-line arguments to a Java application by specifying arguments after the class name as:

```
java classname <<List of Command-line Arguments>>
```

Each argument in the argument list is separated by a space. For example, the following command runs the `com.jdojo.chapter14.Test` class and passes three animal names as the command line arguments.

```
java com.jdojo.chapter14.Test Cat Dog Rat
```

What happens to these three command-line arguments when the `Test` class is run? The operating system passes the list of the arguments to the JVM. Sometimes, the operating system may expand the list of argument by interpreting its meaning and may pass a modified arguments list to the JVM. The JVM parses the argument lists. The separator used for parsing is a space. It creates an array of `String`, whose length is the same as the number of arguments in the list. It populates the `String` array with the items in the arguments list sequentially. Finally, the JVM passes this `String` array to the `main()` method of the `Test` class that you are running. This is the time when we use the `String` array argument passed to the `main()` method. If there is no command-line argument, the JVM creates a `String` array of length zero and passes it to the `main()` method. If you want to pass space-separated words as one argument, you can enclose them in double quotes. You can also avoid the operating system interpretation of special characters by enclosing them in double quotes. Let us create a class called `CommandLine` as listed in Listing 2-1.

Listing 2-5: Processing Command-line arguments inside the main() method

```
// CommandLine.java
package com.jdojo.chapter14;

public class CommandLine {
     public static void main(String[] args) {
          // args contains all command-line arguments
          System.out.println("Total Arguments:" + args.length);

          // Display all arguments
          for(int i = 0 ; i < args.length; i++) {
               System.out.println("Argument #" + (i+1) + ":" + args[i]);
          }
     }
}
```

Table 2-2 shows the command to run the `com.jdojo.chapter14.CommandLine` class and its output.

Table 2-2: Output of command-line argument program running on Windows 2000 DOS prompt

Command	Output
`java com.jdojo.chapter14.CommandLine`	`Total Arguments:0`

`java com.jdojo.chapter14.CommandLine Cat Dog Rat`	`Total Arguments:3`
	`Argument #1:Cat`
	`Argument #2:Dog`
	`Argument #3:Rat`

`java com.jdojo.chapter14.CommandLine "Cat Dog Rat"`	`Total Arguments:1` `Argument #1:Cat Dog Rat`

`java com.jdojo.chapter14.CommandLine 29 Dogs`	`Total Arguments:2`
	`Argument #1:29`
	`Argument #2:Dogs`

What is the use of command-line arguments? It lets you change the behavior of your program without re-compiling it. For example, you may want to sort the contents of a file in ascending or descending order. You may pass command-line arguments, which will specify sorting order. If there is no sorting order specified on the command line, you may assume ascending order by default. If you call the sorting class `com.jdojo.chapter14.SortFile`, you may run it as:

To sort `employee.txt` file in ascending order

```
java com.jdojo.chapter14.SortFile employee.txt asc
```

To sort `department.txt` file in descending order

```
java com.jdojo.chapter14.SortFile department.txt desc
```

To sort `salary.txt` in ascending order

```
java com.jdojo.chapter14.SortFile salary.txt
```

Depending on the second element, if any, of the `String` array passed to the `main()` method of the `SortFile` class, you may sort the file differently.

Note that all command-line arguments are passed to the `main()` method as `Strings`. If you pass a numeric argument, you need to convert the string argument to a number inside the `main()` method. To illustrate this numeric argument conversion let us develop a mini calculator class, which takes an expression as command-line argument and prints the result. Our min calculator supports only four basic operations: add, subtract, multiply and divide. The program is listed in Listing 2-6.

Listing 2-6: A mini command-line calculator

```
// Calc.java
package com.jdojo.chapter14;

public class Calc {
     public static void main(String[] args) {
          // Make sure we have three arguments
          // second argument has only one character to
          // indicate operation
          System.out.println(java.util.Arrays.toString(args));

          if (!(args.length == 3 && args[1].length() == 1)) {
               printUsage();
               return;   // Stop the program here
          }

          // Parse the two number operands. Place the parsing code
          // inside try-catch so that we will handle the error in
          // case both operands are not numbers
          double n1 = 0.0;
```

```
        double n2 = 0.0;
        try {
            n1 = Double.parseDouble(args[0]);
            n2 = Double.parseDouble(args[2]);
        }
        catch (NumberFormatException e) {
            System.out.println("Both operands must be number");
            printUsage();
            return;   // Stop the program here
        }

        // Convert the operation string to char so that we can
        // use switch case statement
        char operation = args[1].charAt(0);

        double result = compute(n1, n2, operation);

        // Print the result
        System.out.println(args[0] + args[1] + args[2] +
                           "=" + result);
    }

    public static double compute(double n1, double n2, char operation)
     {
        // Initialize the result with not-a-number
        double result = Double.NaN;

        switch (operation) {
            case '+':
                result = n1 + n2;
                break;
            case '-':
                result = n1 - n2;
                break;
            case '*':
                result = n1 * n2;
                break;
            case '/':
                result = n1 / n2;
                break;
            default:
                System.out.println("Invalid operation:" + operation);
        }

        return result;
     }

     public static void printUsage() {
        System.out.println("Usage: " +
                           " java com.jdojo.chapter14.Calc expr");
        System.out.println("Where expr could be:");
        System.out.println("n1 + n1");
        System.out.println("n1 - n2");
        System.out.println("n1 * n2");
        System.out.println("n1 / n2");
        System.out.println("n1, n2 are two numbers");
     }
```

```
}
```

You may use the `Calc` class as:

```
java com.jdojo.chapter14.Calc 3 + 7
java com.jdojo.chapter14.Calc 78.9 * 98.5
```

You may get an error when you try to use * (asterisk) to multiply two numbers. The operating system may interpret it as all files names in the current directory. To avoid such errors, you can enclose the operator in double quotes or the escape character provided by your operating system as:

```
java com.jdojo.chapter14.Calc 7 "*" 8
```

TIP
If you use command-line arguments in your Java program, your program is not 100% Java. It is so, because your program does not fit in the category of "Write once, run everywhere". Some operating systems do not have a command prompt and hence you may not be able to use the command-line argument feature. Additionally, an operating system may interpret the meta-characters used in the command-line arguments differently.

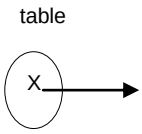
Multi-Dimensional Arrays

If a data element in a list is identified using more than one dimension, you can use a multi-dimensional array to represent the list in your program. For example, a data element in a table is identified by two dimensions, row and column. You can store a tabular data in your Java program in a two dimensional array. You can declare a multi-dimensional array by using a pair of brackets ([]) for each dimension in the array declaration. For example, you can declare a two dimensional array of `int` as:

```
int[][] table;     // table can refer to any two-dimensional array
```

Here, `table` is a reference variable that can hold a reference to a two-dimensional array of `int`. Memory is allocated only for the reference variable `table` and not for any array elements at the time of declaration. The memory state after the above fragment of code is executed is depicted in Figure 2-3.

Figure 2-3: Memory state after the declaration of a two-dimensional array

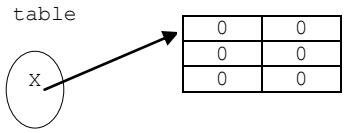


A two-dimensional array of `int` with 3 rows and 2 columns can be created as:

```
table = new int[3][2];
```

The memory state after execution of the above snippet of code has been depicted in Figure 2-4. All elements have been shown to have a value of zero, because all elements of a numeric array are initialized to zero by default. The rules for default initialization of array elements of a multi-dimensional array are the same as that of a single dimensional array as discussed previously in this chapter.

Figure 2-4: Memory state after the creation of a two-dimensional array



The index of each dimension in a multi-dimensional array is zero-based. Each element of the `table` array can be accessed as `table[rownumber][columnNumber]`.The row number and the column number always starts at zero. For example, we can assign a value to the first row and the second column in `table` array as:

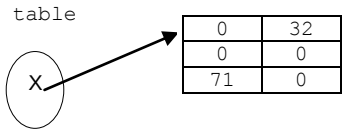
```
table[0][1] = 32;
```

We can assign a value 71 to the third row and the first column as:

```
table[2][0] = 71;
```

The memory state after the above two assignments has been depicted in Figure 2-5.

Figure 2-5: Memory state after two assignments to the two-dimensional array elements

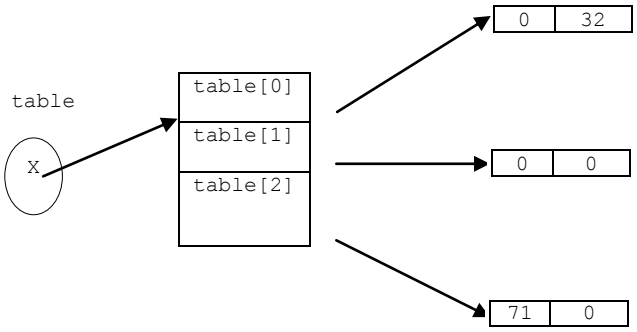


Java does not support multi-dimensional arrays in a true sense. Rather, it supports array of arrays. Using an array of arrays, you can implement the same functionality as provided by multi-dimensional arrays. When you create a two-dimensional array, the elements of the first array are of an array type, which can refer to a single dimensional array. The size of each single-dimensional array need not be the same. Considering the array of arrays concept for the `table` two-dimensional array, we can depict the memory state after array creation and assignments of two values as shown in Figure 2-6. The name of the two-dimensional array, `table`, refers to an array of three elements. Each element of the array to which `table` refers is a one-dimensional array of `int`. The data type of `table[0]`, `table[1]` and `table[2]` is an `int` array. The length of `table[0]`, `table[1]` and `table[2]` is 2.

You must specify the dimension of at least the first level array at the time you create a multi-dimensional array. For example, when you create a two-dimensional array you must specify at least the first dimension, which is the number of rows. We can achieve the same results as above code fragment as follows.

```
table = new int[3][];   // You must specify at least first dimension
                        // Number of columns is left
```

Figure 2-6: An array of arrays



The above statement creates only first level of array. Only `table[0]`, `table[1]` and `table[2]` exist at this time. They are referring to `null`. At this time, `table.length` has a value of `3`. Since `table[0]`, `table[1]` and `table[2]` are referring to `null`, you cannot use `length` attribute on them. That is, we have created three rows in a table, but we do not know how many columns each row will contain. Since `table[0]`, `table[1]` and `table[2]` are arrays of `int`, we can assign them values as:

```
table[0] = new int[2];  // Create two columns for row 1
table[1] = new int[2];  // Create two columns for row 2
table[2] = new int[2];  // Create two columns for row 3
```

We have completed the creation of the two-dimensional array, which has the three rows and each row has two columns. We can assign the values to some cells as:

```
table[0][1] = 32
table[2][0] = 71
```

It is also possible to create a two-dimensional array with different number of columns for each row. Such an array is called a *ragged* array. Listing 2-7 illustrates working with a ragged array.

Listing 2-7: An example of a ragged array

```
// RaggedArray.java
package com.jdojo.chapter14;

public class RaggedArray {
     public static void main(String[] args) {
          // Create a two-dimensional array of 3 rows
          int[][] raggedArr = new int[3][];

          // Add 2 columns to first row
          raggedArr[0] = new int[2];

          // Add 1 column to second row
          raggedArr[1] = new int[1];

          // Add 3 columns to third row
          raggedArr[2] = new int[3];

          // Assign values to all elements of raggedArr
```

```
        raggedArr[0][0] = 1;
        raggedArr[0][1] = 2;
        raggedArr[1][0] = 3;
        raggedArr[2][0] = 4;
        raggedArr[2][1] = 5;
        raggedArr[2][2] = 6;

        // Print all elements. One row at one line
        System.out.println(raggedArr[0][0] + "\t" + raggedArr[0][1]);
        System.out.println(raggedArr[1][0]);
        System.out.println(raggedArr[2][0] + "\t" + raggedArr[2][1]
                           + "\t" + raggedArr[2][2]);
    }
}
```

```
Output

1    2
3
4    5    6
```

TIP
Java supports array of arrays, which can be used to achieve functionalities provided by multi-dimensional arrays. Multi-dimensional arrays are widely used in scientific and engineering applications. If you are using arrays in your business application program that have more than two dimensions, you may need to reconsider the choice of multi-dimensional arrays as your choice of data structure.

Accessing Elements of a Multi-Dimensional Array

Typically, a multi-dimensional array is populated using nested for loops. The number of for- loops used to populate a multi-dimensional array equals the number of dimensions in the array. For example, two for-loops are used to populate a two-dimensional array. Typically, a loop is used to access the elements' values of a multi-dimensional array. Listing 2-8 illustrates how to populate and access elements of a two-dimensional array.

Listing 2-8: Accessing elements of a multi-dimensional array

```
// MDAccess.java
package com.jdojo.chapter14;

public class MDAccess {
    public static void main(String[] args){
        int[][] ra = new int[3][];

        ra[0] = new int[2];
        ra[1] = new int[1];
        ra[2] = new int[3];

        // Populate the ragged array using for loops
        for(int i = 0; i < ra.length; i++) {
            for(int j = 0; j < ra[i].length; j++){
```

```
                    ra[i][j] = i + j;
               }
          }

          // Print the array using for loops
          for(int i = 0; i < ra.length; i++) {
               for (int j = 0; j < ra[i].length; j++){
                    System.out.print(ra[i][j] + "\t");
               }

               // Add a new line after each row is printed
               System.out.println();
          }
     }
}
```

```
Output

0    1
1
2    3    4
```

Initializing Multi-Dimensional Arrays

You may initialize the elements of a multi-dimensional array by supplying the list of values at the time of its declaration or at the time of its creation. You cannot specify the length of any dimension if you initialize the array with a list of values. The number of initial values for each dimension will determine the length of each dimension in the array. Since many dimensions are involved in a multi-dimensional array, the list of values for a level is enclosed in braces. For a two-dimensional array, the list of values for each row is enclosed in a pair of braces as:

```
int[][] arr = {{10, 20, 30}, {11, 22}, {222, 333, 444, 555}};
```

The above statement created a two-dimensional array with 3 rows. The first row contains 3 columns with values 10, 20, and 30. The second row contains two columns with values 11 and 22. The third row contains 4 columns with values 222, 333, 444, and 555. A zero-row and zero-column two-dimensional array can be created as:

```
int[][] empty2D = { };
```

Initialization of a multi-dimensional array of reference type follows the same rule. We can initialize a two-dimensional reference type arrays as:

```
String[][] acronymList = {{"JMF", "Java Media Framework"},
                          {"JSP", "Java Server Pages"},
                          {"JMS", "Java Message Service"}};
```

You can initialize the elements of a multi-dimensional array at the time you create it as:

```
int[][] arr = new int[][]{{1, 2}, {3,4,5}};
```

Enhanced for-loop for Arrays

Java 5 introduced an enhanced for-loop that lets you loop through elements of an array in a cleaner way. The enhanced for–loop is also known as for-each loop. The syntax of the for-each loop is as follows.

```
for(DataType e : array) {
     // Loop body goes here...

     // e contains one element of the array at a time
}
```

The for-each loop uses the same `for` keyword used by the basic for-loop. Its body is executed as many times as the number of elements in the `array`. The "`DataType e`" is a variable declaration, where `e` is the variable name and `DataType` is its data type. The data type of the variable `e` should be assignment-compatible with the type of the `array`. The variable declaration is followed by a colon (`:`), which is followed by the reference of the array that you want to loop through. The for-each loop assigns the value of an element of the array to the variable `e`, which you can use inside the body of the for-each loop. The following snippet of code uses a for-each loop to print all elements of an `int` array.

```
int[] numList = {1, 2, 3, 4};

for(int num : numList) {
     System.out.println(num);
}
```

```
Output:

1
2
3
4
```

We can accomplish the same thing using the basic for-loop as follows.

```
int[] numList = {1, 2, 3, 4};

for(int i = 0; i < numList.length; i++ ) {
     int num = numList[i];
     System.out.println(num);
}
```

```
Output:

1
2
3
4
```

Note that the for-each loop provides a way to loop through elements of an array, which is cleaner than the basic for-loop. However, the for-each loop is not a replacement for the basic for-loop,

because you cannot use it in all circumstances. For example, if you need to get to the index of the element inside the body of the loop, the for-each loop does not provide that. If you want to modify the value of the element of the array inside the for-each loop, you cannot do that.

TIP
The for-each loop also lets you iterate over elements of any collection object that implements `java.lang.Iterable` interface. We will revisit for-each loop in the chapter on *Collections*.

Array Declaration Syntax

You can declare an array by placing brackets (`[]`) after the data type of the array or after the name of the array reference variable. For example, the following declaration

```
int[] empIds;
int[][] points2D;
int[][][] points3D;
Person[] persons;
```

are equivalent to:

```
int empIds[];
int points2D[][];
int points3D[][][];
Person persons[];
```

Java also allows you to mix two syntaxes. In the same array declaration, you can place some brackets after the data type and some after the variable name. For example, you can declare a two-dimensional array of `int` as:

```
int[] points2D[];
```

You can declare a two-dimensional and a three-dimensional array of `int` in one declaration statement as:

```
int[] points2D[], points3D[][];
```

or

```
int[][] points2D, points3D[];
```

Runtime Array Bounds Checks

Java checks array bounds for every access to an array element runtime. If the array bound is exceeded, `java.lang.ArrayIndexOutOfBoundsException` exception is thrown. The only requirement for array index values at compile time is that they must be integers. The Java compiler does not check if the value of an array index is less than zero or beyond its length. This check must be performed at runtime, before every access to an array element is allowed. Runtime array

bounds checks slow down the program execution for two reasons. The first reason is the cost of bound checks itself. To check the array bounds, the length of array must be loaded in memory and two comparisons (one for less than zero and one for greater than or equal to its length) must be performed. The second reason is that an exception must be thrown when the array bounds are exceeded, Java must do some housekeeping and get ready to throw an exception, if the array bounds are exceeded. Listing 2-9 illustrates the exception thrown if array bounds are exceeded. The program creates an array of int named test. Since the `test` array has a length of 3, the program cannot access the 4th element, i.e., `test[3]`. An attempt to access the fourth element will result in an `ArrayIndexOutOfBoundsException` exception.

Listing 2-9: Array bounds checks

```
// ArrayBounds.java
package  com.jdojo.chapter14;

public class ArrayBounds {
     public static void main(String[] args) {
          int[] test = new int[3];

          System.out.println("Assigning 12 to the first element");
          test[0] = 12; // index 0 is between 0 and 2. Ok

          System.out.println("Assigning 79 to the fourth element");
          test[3] = 12; // index 3 is not between 0 and 2.
                        // At runtime exception will be thrown

          System.out.println("We will not get here");
     }
}
```

```
Output:

Assigning 12 to the first element
Assigning 79 to the fourth element
java.lang.ArrayIndexOutOfBoundsException: 3
     at com.jdojo.chapter14.ArrayBounds.main(ArrayBounds.java:12)
Exception in thread "main"
```

It is a good practice to check for array length before accessing an array element. The fact that array bounds violation throws an exception may be misused as shown in the following snippet of code, which prints values stored in an array. The code fragment uses an infinite while–loop to print values of the elements of an array and relies on exception throwing mechanism to check for array bounds. The right way is to use a for-loop and check for array index value using `length` property of the array

```
// Create an array
int[] arr = new int[10];

//Populate the array here...

// Print the array. Wrong way
try {
     // Start an infinite loop. When we are done with all elements
     // an exception will be thrown and we will be in catch block
     // and hence out of loop
```

```
     int counter = 0;
     while (true) {
          System.out.println(arr[counter++]);
     }
}
catch (ArrayIndexOutOfBoundsException e) {
     // We are done with printing array elements
}

// Do some processing here...
```

What is the Class of an Array Object?

Arrays in Java are objects. Since every object has a class, we must have a class for every array. All methods of `java.lang.Object` can be used on arrays of all types. Since the `getClass()` method of the `java.lang.Object` class gives the reference of the class for any object in Java, we will use this method to get the class name for all arrays. Listing 2-10 illustrates how to get the class name of an array.

Listing 2-10: Knowing the class of an array

```
// ArrayClass.java
package com.jdojo.chapter14;

public class ArrayClass {
     public static void main (String[] args){
          int[] iArr = new int[2];
          int[][] iiArr = new int[2][2];
          int[][][] iiiArr = new int[2][2][2];

          String[] sArr  = {"A", "B"} ;
          String[][] ssArr = {{"AA"}, {"BB"}} ;
          String[][][] sssArr = {} ; // 3D empty array of string

          // Print the class name for all arrays
          System.out.println("int[]:" + getClassName(iArr));
          System.out.println("int[][]:" + getClassName(iiArr));
          System.out.println("int[][][]:" + getClassName(iiiArr));
          System.out.println("String[]:" + getClassName(sArr));
          System.out.println("String[][]:" + getClassName(ssArr));
          System.out.println("String[][][]:" + getClassName(sssArr));
     }

     /* Any java object can be passed to getClassName() method.
        Since every array is also an object, we can also pass
        an array to this method.
     */
     public static String getClassName(Object obj) {
          // Get the reference of its class
          Class c = obj.getClass();

          // Get the name of the class
          String className = c.getName();
          return className;
```

```
    }
}
```

```
Output

int[]:[I
int[][]:[[I
int[][][]:[[[I
String[]:[Ljava.lang.String;
String[][]:[[Ljava.lang.String;
String[][][]:[[[Ljava.lang.String;
```

The class name of an array starts with left bracket(s) `([)`. The number of left brackets is equal to the dimension of the array. For `int` array, the left bracket(s) is followed by a character `I`. For reference type array, the left bracket(s) is followed by a character `L`, followed by the name of the class name, which is followed by a semi-colon. The class names for one-dimensional primitive array and reference type have been shown in Table 2-3.

Table 2-3: Class name of arrays

Array Type	Class Name
`byte[]`	`[B`
`short[]`	`[S`
`int[]`	`[I`
`long[]`	`[J`
`char[]`	`[C`
`float[]`	`[F`
`double[]`	`[D`
`boolean[]`	`[Z`
`com.jdojo.Person[]`	`[Lcom.jdojo.Person;`

The class names of arrays are not available at compile time for declaring or creating arrays. You must use the syntax described in this chapter to create an array. That is, you cannot write the following to create an `int` array.

```
[I myIntArray;
```

Rather, you must write to create an `int` array.

```
int[] myIntArray;
```

Array Assignment Compatibility

Data type of each element of an array is the same as the data type of the array. For example, each element of `int[]` array is of type `int`; each element of `String[]` array is of type `String`. The value assigned to an element of an array must be assignment compatible to its data type. For example, it is allowed to assign a `byte` value to an element of an `int` array, because `byte` is

assignment compatible to `int`. However, it is not allowed to assign a `float` value to an element of an `int` array, because `float` is not assignment compatible to `int`.

```
byte bValue = 10;
float fValue = 10.5f;
int[] sequence = new int[10];

sequence[0] = bValue;        // Ok
sequence[1] = fValue;        // Compiler error
```

The same rule must be followed while dealing with a reference type array. If there is a reference type array of type `T`, its elements can be assigned an object reference of type `S`, if and only if, `S` is assignment compatible to `T`. The subclass object reference is always assignment compatible to the superclass. Since the `java.lang.Object` class is the superclass of all classes in Java, you can use an array of `Object` class to store objects of any class. For example,

```
Object[] genericArray = new Object[4];

genericArray[0] = new String("Hello");     // Ok

genericArray[1] = new Person("Daniel");    // Ok. Assuming Person class
                                           // exists

genericArray[2] = new Account(189);        // Ok. Assuming Account class
                                           // exists

genericArray[3] = null;                    // Ok. null can be assigned
                                           // to any reference type
```

You need to perform a cast at the time you read back the object from array as:

```
/* Compiler will flag an error for the following statement.
   genericArray is of Object type and an Object reference cannot be
   assigned to a String reference variable. Even though genericArray[0]
   contains a String object reference, we need to cast it to String as
   we do in next statement.
*/
String s = genericArray[0];          // Compiler error

String str = (String)genericArray[0];
Person p = (Person)genericArray[1];
Account a = (Account)genericArray[2];
```

If you try to cast the array element to a type, whose actual type is not assignment compatible to the new type, the `java.lang.ClassCastException` is thrown. For example, the following statement will throw the `ClassCastException` at runtime.

```
String str = (String)genericArray[1]; // Person can't be cast to String
```

The vice-versa is not true. That is, you cannot store an object reference of the superclass in an array of the subclass. The following snippet of code illustrates this.

```
String[] names = new String[3];
names[0] = new Object(); // Error. Object is superclass of String
```

```
names[1] = new Person();     // Error, Person is not subclass of String
names[2] = null;             // Ok.
```

Finally, an array reference can be assigned to another array reference of another type if the former type is assignment compatible to the latter type. For example,

```
Object[] obj = new Object[3];
String[] str = new String[2];
Account[] a = new Account[5];

obj = str;              // Ok
str = (String[]) obj;   // Ok. Because obj has String array reference

obj = a;
str = (String[]) obj;   // ClassCastException error. Because obj has
                        // reference of Account array and Account can't
                        // be converted to String

a = (Account[]) obj;    // Ok
```

Converting an ArrayList/Vector to an Array

An `ArrayList` can be used when the number of elements in the list is not precisely known. Once the number of elements in the list is fixed, you may want to convert an `ArrayList` to an array. You may do this for one of the following reasons.

- The program semantics may require you to use an array and not an `ArrayList`. For example, you may need to pass an array to a method, but you have data stored in an `ArrayList`.
- You may want to store user inputs in an array. However, you do not know the number of values the user will input. In such a case, you can store values in an `ArrayList` while accepting input from the user. At the end, you can convert the `ArrayList` to an array.
- Accessing array elements is faster than accessing `ArrayList` elements. If you have an `ArrayList` and you want to access the elements multiple times, you may want to convert the `ArrayList` to an array for better performance.

The `ArrayList` class has an overloaded method named `toArray()` as:

```
public Object[] toArray()
public Object[] toArray(Object[]  a)
```

The first method returns the elements of `ArrayList` as an array of `Object`. The second method takes an array of `Object` (you can pass an array of any class though) as argument. All `ArrayList` elements are copied to that array if there is enough space and the same array is returned. If there is not enough space in the array that is passed to copy all `ArrayList` elements, a new array is created. The type of new array is the same as the passed in array. The length of the new array is equal to the size of `ArrayList`.

The above discussions and the example described in Listing 2-11 also apply to converting a `Vector` to an array.

Listing 2-11: An ArrayList to an array conversion

```
// ArrayListToArray.java
package com.jdojo.chapter14;

import java.util.ArrayList;

public class ArrayListToArray {
     public static void main(String[] args){
          ArrayList al = new ArrayList();
          al.add("cat");
          al.add("dog");
          al.add("rat");

          // Print the content of arraylist
          System.out.println("ArrayList:" + al );

          // Create an array of same length as arraylist
          String[] s1 = new String[al.size()];

          // Copy arraylist elements to array
          String[] s2 = (String[]) al.toArray(s1);

          // Since s has enough space to copy all arraylist
          // elements, al.toArray(s) returns s itself after copying
          // all arraylist elements to it. Here, s1 == s2 is true
          System.out.println("s1 == s2:" + (s1 == s2));
          System.out.println("s1:" + getArrayContent(s1));
          System.out.println("s2:" + getArrayContent(s2));


          // Create an array of string with 1 elements, which is less
          // than size of arraylist, which is 3
          s1 = new String[1];
          s1[0] = "hello" ;  // Store hello in first element

          // Copy arraylist elements to array
          s2 = (String[]) al.toArray(s1);

          /* Since s doesn't have sufficient space to copy all
             arraylist elements, al.toArray(s) creates a new String
             array with 3 elements in it. All elements of arraylist
             are copied to new array. Finally, new array is returned.
             Here, s1 == s2 is false. s will be untouched by the
             method call
          */
          System.out.println("s1 == s2:" + (s1 == s2));
          System.out.println("s1:" + getArrayContent(s1));
          System.out.println("s2:" + getArrayContent(s2));
     }

     // Method to get the array content enclosed in square brackets
     public static String getArrayContent ( String[] s){
          StringBuffer  sb = new StringBuffer("[");
          for (int i = 0; i < s.length; i++ ){
               sb.append(s[i]);
```

```
                // Append a comma, if needed
                if (i != s.length - 1) {
                     sb.append (", ");
                }
           }

           sb.append("]");
           return sb.toString();
      }
}
```

Output

```
ArrayList:[cat, dog, rat]
s1 == s2:true
s1:[cat, dog, rat]
s2:[cat, dog, rat]
s1 == s2:false
s1:[hello]
s2:[cat, dog, rat]
```

References

Bloch, Joshua. *Effective Java: Programming Language Guide.* Addison-Wesley, 2001.

Cormen , Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2nd Edition. The MIT Press, 2001.

Cornell, Gary, and Cay Horstmann. *Core Java 1.2.* Vol. 1. Prentice Hall Computer Books, 1998.

—. *Core Java 1.2.* Vol. 2. Prentice Hall Computer Books, 1998.

Downing, Troy Bryan. *Java RMI: Remote Method Invocation.* Wiley Publishing, 1998.

Eckel, Bruce. *Thinking in Java.* 1st Edition. Prentice Hall, 1998.

Freeman, Elisabeth, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns.* O'Reilly Media, 2004.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* 3rd Edition. Addison-Wesley, 2005.

Grosso, William. *Java RMI.* 1st Edition. O'Reilly Media, 2001.

Hall, Marty. *Core Servlets and Javaserver Pages.* Prentice Hall, 2000.

Harold, Elliotte Rusty. *Java I/O.* 1st Edition. O'Reilly Media, 1999.

—. *Java Network Programming.* 2nd Edition. O'Reilly Media, 2000.

Horton, Ivor. *Beginning Java.* Wrox Press, 1997.

Hyde, Paul. *Java Thread Programming.* Sams, 1999.

"Java SE 7 Documentation." *Oracle Corporation Web site.* http://download.oracle.com/javase/7/docs/ (accessed 2011).

Lakshman, Bulusu. *Oracle and Java Development.* Sams, 2001.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns.* 2nd Edition. Addison-Wesley, 1999.

Liskov, Barbara, and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Professional, 2000.

Mano, M. Morris. *Computer Engineering: Hardware Design.* Prentice Hall, 198.

Oaks, Scott. *Java Security.* 2nd Edition. O'Reilly Media, 2001.

Oaks, Scott, and Henry Wong. *Java Threads.* 2nd Edition. O'Reilly Media, 1999.

Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* 2nd Edition. Morgan Kaufmann, 1997.

Pawlan, Monica. "Reference Objects and Garbage Collection." *Pawlan Communications.* 8 1998. http://www.pawlan.com/monica/articles/refobjs/ (accessed 7 2011).

Preiss, Bruno R. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java.* Wiley, 1999.

Shirazi, Jack. *Java Performance Tuning.* 1st Edition. O'Reilly Media, 2000.

Topley, Kim. *Core Java Foundation Classes.* Prentice Hall PTR, 1998.

Travis, Greg. "IBM DeveloperWorks." *Getting started with new I/O (NIO).* July 09, 2003. https://www.ibm.com/developerworks/java/tutorials/j-nio/ (accessed September 01, 2008).

Venners, Bill. *Inside the Java 2 Virtual Machine.* 2nd Edition. McGraw-Hill Companies, 2000.

Walsh, Aaron E, Justin Couch, and Daniel H. Steinberg. *Java 2 Bible.* Wiley Publishing, 2000.

Index

R

S

T

U

V